JAPAN SMARTPHONE SECURITY ASSOCIATION

Android アプリのセキュア設計
・
セキュアコーディングガイド

2018 年 9 月 1 日版

一般社団法人日本スマートフォンセキュリティ協会（JSSEC）

セキュアコーディング WG

文書管理番号：JSSEC-TECA-SC-GD20180901B

目次

2018 年 **9** 月 **1** 日版

一般社団法人日本スマートフォンセキュリティ協会（**JSSEC**）

セキュアコーディング **WG**

※ 本ガイドの内容は執筆時点のものです。サンプルコードを使用する場合はこの点にあらかじめご注意ください。

※ JSSEC ならびに執筆関係者は、このガイド文書に関するいかなる責任も負うものではありません。全ては自己責任にてご活用ください。

※ Android は、Google, Inc. の商標または登録商標です。また、本文書に登場する会社名、製品名、サービス名は、一般に各社の登録商標または商標です。本文中では ®、TM、© マークは明記していません。

※ この文書の内容の一部は、Google, Inc. が作成、提供しているコンテンツをベースに複製したもので、クリエイティブ・コモンズの表示 3.0 ライセンスに記載の条件に従って使用しています。

1 はじめに

1.1 スマートフォンを安心して利用出来る社会へ

本ガイドは Android アプリケーション開発者向けのセキュア設計、セキュアコーディングのノウハウをまとめた Tips 集です。できるだけ多くの Android アプリケーション開発者に活用していただきたく思い、ここに公開いたします。

昨今、スマートフォン市場は急拡大しており、さらにその勢いは増すばかりです。スマートフォン市場の急拡大は多種多彩なアプリケーション群によってもたらされています。従来の携帯電話ではセキュリティ制約によって利用できなかったさまざまな携帯電話の重要な機能がスマートフォンアプリケーションには開放され、従来の携帯電話では実現できなかった多種多彩なアプリケーション群がスマートフォンの魅力を引き立てています。

スマートフォンのアプリケーション開発者にはそれ相応の責任が生じています。従来の携帯電話ではあらかじめ課せられたセキュリティ制約によって、セキュリティについてあまり意識せずに開発したアプリケーションであっても比較的安全性が保たれていました。スマートフォンでは前述のとおり、携帯電話の重要な機能がアプリケーション開発者に開放されているため、アプリケーション開発者がセキュリティを意識して設計、コーディングをしなければ、スマートフォン利用者の個人情報が漏洩したり、料金の発生する携帯電話機能をマルウェアに悪用されたりといった被害が生じます。

Android スマートフォンは iPhone に比べると、アプリケーション開発者のセキュリティへの配慮がより多く求められます。iPhone に比べ Android スマートフォンはアプリケーション開発者に開放された携帯電話機能が多く、App Store に比べ Google Play（旧 Android Market）は無審査でアプリケーション公開ができるなど、アプリケーションのセキュリティがほぼ全面的にアプリケーション開発者に任されているためです。

スマートフォン市場の急拡大にともない、様々な分野のソフトウェア技術者が一気にスマートフォンアプリケーション開発市場に流れ込んできており、スマートフォン特有のセキュリティを考慮したセキュア設計、セキュアコーディングのノウハウ集約、共有が急務となっています。

このような状況を踏まえ、一般社団法人日本スマートフォンセキュリティ協会はセキュアコーディング WG を立ち上げ、Android アプリケーションのセキュア設計、セキュアコーディングのノウハウを集めて、公開することにいたしました。それがこのガイド文書です。多くの Android アプリケーション開発者にセキュア設計、セキュアコーディングのノウハウを知っていただき、アプリケーション開発に活かしていただくことで、市場にリリースされる多くの Android アプリケーションのセキュリティを高めることを狙っています。その結果、安心、安全なスマートフォン社会づくりに貢献したいと考えています。

1.2 常にベータ版でタイムリーなフィードバックを

私たち JSSEC セキュアコーディング WG はこのガイド文書の内容について、できるだけ間違いがないように心がけておりますが、その正しさを保証するものではありません。私たちはタイムリーにノウハウを公開し共有していくことが第一と考え、最新かつその時点で正しいと思われることをできるだけ記載・公開し、間違いがあればフィードバックを頂いて常に正しい情報に更新し、タイムリーに提供するよう心がける、いわゆる常にベータ版というアプローチをとっています。このアプローチはこのガイド文書をご利用いただく多くの Android アプリケーション開発者のみなさまにとって有意義であると私たちは信じています。

このガイド文書とサンプルコードの最新版はいつでも下記 URL から入手できます。

- https://www.jssec.org/dl/android_securecoding/ ガイド文書
- https://www.jssec.org/dl/android_securecoding.zip サンプルコード一式

1.3 本文書の利用許諾

このガイド文書のご利用に際しては次の 2 つの注意事項に同意いただく必要がございます。

1. このガイド文書には間違いが含まれている可能性があります。ご自身の責任のもとでご利用ください。
2. このガイド文書に含まれる間違いを見つけた場合には、下記連絡先までメールにてご連絡ください。ただしお返事することや修正をお約束するものではありませんのでご了承ください。

一般社団法人 日本スマートフォンセキュリティ協会

セキュアコーディング WG 問い合わせ

メール宛先：jssec-securecoding-qa@googlegroups.com

件名：【コメント応募】Android アプリのセキュア設計・セキュアコーディングガイド 2018 年 9 月 1 日版

内容：氏名 (任意)/所属 (任意)/連絡先 E-mail(任意)/ご意見 (必須)/その他ご希望 (任意)

1.4 2018 年 2 月 1 日版からの訂正記事について

本節では、前版の記事について事実関係と照らし合わせることで判明した訂正事項を一覧にして掲載しています。各訂正記事は、執筆者による継続的な調査結果だけでなく読者の方々の貴重なご指摘を広く取り入れたものです。特に、いただいたご指摘は、本改訂版をより実践に即したガイドとして高い完成度を得るための最も重要な糧となっています。

前版を元にアプリケーション開発を進めていた読者は、以下の訂正記事一覧に特に目を通していただきますようお願いいたします。なお、ここで掲げる項目には、誤植の修正、記事の追加、構成の変更、単なる表現上の改善は含みません。

本ガイドに対するコメントは、今後もお気軽にお寄せくださいますようよろしくお願いいたします。

訂正記事一覧

表 1.4.1: 訂正記事一覧

2018 年 2 月 1 日版の修正個所	本改訂版の訂正記事	訂正の要旨
2.5. サンプルコードの *Android Studio* への取り込み手順	2.5. サンプルコードの *Android Studio* への取り込み手順	AndroidStudio version 3.1 以降の挙動に合わせてサンプルプロジェクトの取り込み手順を改訂しました。
4.1.3.7. *Autofill* フレームワークについて	4.1.3.7. *Autofill* フレームワークについて	Android 9.0(API Level 28) で Autofill Service のパッケージ情報が取得できるようになり、これを利用した新たなセキュリティ対策を追記しました。
4.5.3.6. *[参考]SQLite* データベースを暗号化する *(SQLCipher for Android)*	4.5.3.6. *[参考]SQLite* データベースを暗号化する *(SQLCipher for Android)*	SQLCipher についての記載内容を 2018 年 8 月現在のものに合わせて最新にしました。
（該当なし）	4.9.3.4. *WebView* の *Safe Browsing* について	これまで記載のなかった WebView で利用できる SafeBrowsing 機能の利用環境や Android 8.0(API Level 26) および Android 8.1(API Level 27) で追加された API の説明を追記しました。
（該当なし）	4.11. 共有メモリを使用する	Android 8.1(API Level 27) で追加された SharedMemory API の安全な使用法について説明を追記しました。
5.2.1.2. 独自定義の *Signature Permission* で自社アプリ連携する方法	5.2.1.2. 独自定義の *Signature Permission* で自社アプリ連携する方法	Android 9.0(API Level 28) で導入された、アプリの署名検証 API の説明を追記しました。
5.4.1.2. *HTTPS* 通信する	5.4.1.2. *HTTPS* 通信する	サーバーの証明書の検証において RFC2818 にて CN の使用が非推奨となっているため Android 9.0(API Level 28) では検証に SAN のみを使用するようになったことを追記しました。
5.4.3.7. *Network Security Configuration*	5.4.3.7. *Network Security Configuration*	Android 9.0(API Level 28) 以降で cleartextTrafficPermitted のデフォルトが false となったことを反映し、内容を改訂しました。
5.7. 指紋認証機能を利用する	5.7. 指紋認証機能を利用する	Android 9.0(API Level 28) で導入された BiometricPrompt API を用いたサンプルコードを追加しました。
5.4.3.2. *Android OS* の証明書ストアにプライベート認証局のルート証明書をインストールする	5.4.3.2. *Android OS* の証明書ストアにプライベート認証局のルート証明書をインストールする	Android 7.0(API Level 24) 以降では、プライベート認証局のルート証明書をインストールしてもシステムがこれを無視する旨を追記しました。
5.4.3.8. （コラム） セキュア接続の *TLS1.2* への移行について	5.4.3.8. （コラム） セキュア接続の *TLS1.2* への移行について	Android OS のシェア情報を 2018 年 8 月時点のデータで更新しました。

2 ガイド文書の構成

2.1 開発者コンテキスト

セキュアコーディング系のガイド文書は「こういうコーディングは危ない、だからこのようにコーディングすべき」といった内容で構成されることが多いのですが、このような構成はすでにコーディングされたソースコードをレビューするときには役立つ反面、これから開発者がコーディングしようというときには、どの記事を読んだらよいのか分かりにくいという問題があります。

このガイド文書では、開発者がいま何をしようとしているか？ という開発者コンテキストに着目し、開発者コンテキストに合わせた切り口の記事を用意する方針をとっています。たとえば「Activity を作る・利用する」や「SQLite を使う」という開発者が行うであろう作業単位ごとに記事を用意しています。

開発者コンテキストに合わせて記事を用意することにより、開発者は必要な記事を見つけやすく、業務にすぐ役立つようになると考えています。

2.2 サンプルコード、ルールブック、アドバンスト

それぞれの記事はサンプルコード、ルールブック、アドバンストの 3 つのセクションで構成されています。お急ぎの方はサンプルコードとルールブックをご覧ください。ある程度再利用可能なパターンに落とし込んだ内容にしてあります。サンプルコードセクションとルールブックセクションに収まらない課題をお持ちの方はアドバンストをご覧ください。個別課題の解決方法を検討するための考慮材料を記載してあります。

なお、サンプルコードおよび記事の内容は特別な記述がない限り Android 4.0.3(API Level 15) 以降を対象にしています。Android 4.0.3(API Level 15) より前のバージョンにおいては動作確認をしておらず、対策として効果がない場合もありますのでご注意ください。また、対象範囲内のバージョンであっても、組み込んだ端末で動作をご確認の上、ご自身の責任のもとでご利用ください。

また、本書で紹介するサンプルコードは targetSdkVersion の API Lvel を 26 以上に設定しています。これは Google 社が定めている以下の要件に従ったものです：

- 2018 年 8 月以降に Google Play ストアから配信する新規リリースのアプリケーションは targetSdkVersion の API Level を 26(Android 8.0) 以上にしなければならない
- 2018 年 11 月以降に既存アプリをバージョンアップする場合も targetSdkVersion を 26 以上にしなければならない
- 2019 年以降は、毎年要求される targetSdkVersion が上げられる

2.2.1 サンプルコード

サンプルコードセクションでは、その記事がテーマとする開発者コンテキストにおいて基本的なお手本となるサンプルコードを掲載しています。複数のパターンがある場合はその分類方法とそれぞれのパターンのサンプルコードを用意しています。解説においては簡潔さを心がけており、セキュリティ上考慮すべきポイントを本文中で「ポイント：」部分に番号付き箇条書きで記載し、その箇条書き番号 N に対応するサンプルコードにも「★ポイント **N** ★」と記載しコメントで解説しています。一つのポイントがサンプルコード上では複数個所に対応する場合があることにご注意ください。このようにセキュリティを考慮すべき個所はソースコード全体に対して僅かな量ですが、それらの個所は点在します。セキュリティの考慮が必要な個所を見渡すことができるように、サンプルコードはクラス単位でまるごと掲載するようにしています。

このガイド文書で掲載しているサンプルコードは一部です。すべてのサンプルコードをまとめた圧縮ファイルも下記の URL に公開しています。Apache License, Version 2.0 で公開していますので、自由にサンプルコードをコピー&ペーストしてご利用いただけます。ただしエラー処理についてはサンプルコードが長くなり過ぎないように最小限にしていますのでご注意ください。

- https://www.jssec.org/dl/android_securecoding/ ガイド文書
- https://www.jssec.org/dl/android_securecoding.zip サンプルコード一式

サンプルコードに添付する Projects/keystore ファイルは APK 署名用の開発者鍵を含んだキーストアファイルです。パスワードは「android」です。自社限定系のサンプルコードを APK 署名する際にご利用ください。

デバッグ用にキーストアファイル debug.keystore を用意しているので、Android Studio で開発する場合は、Android Studio の個別のプロジェクトで設定しておくと、自社限定系のサンプルコードの動作確認に便利です。また、複数の APK から成るサンプルコードにおいて、各 APK 間の連携動作を確認するためには、各々の AndroidManifest.xml 内の android:debuggable の設定を合わせる必要があります。Android Studio から APK をインストールする場合は、明示的に設定が無ければ自動的に android:debuggable=”true” になります。

サンプルコードおよびキーストアファイルを Android Studio に取り込む方法については「2.5. サンプルコードの *Android Studio* への取り込み手順」をご参照ください。

2.2.2 ルールブック

ルールブックセクションでは、その記事がテーマとする開発者コンテキストにおいて、セキュリティ観点から守るべきルールや考慮事項を掲載しています。ルールブックセクションの冒頭にはそのセクションで扱っているルールを表形式で一覧表示し、「必須」または「推奨」のレベル分けをしています。ルールには肯定文または否定文の 2 種類がありますので、必須の肯定文は「やらなきゃだめ」、推奨の肯定文は「やったほうがよい」、必須の否定文は「やったらだめ」、推奨の否定文は「やらないほうがよい」といったレベル感で表現しています。もちろんこのレベル分けは執筆者の主観に基づくものですので、参考程度としてお取扱いください。

サンプルコードセクションに掲載されているサンプルコードはこれらのルールや考慮事項が反映されたものとなっていますが、その詳しい説明はルールブックセクションに記載されています。また、サンプルコードセクションでは扱っていないルールや考慮事項についてもルールブックセクションでは扱っています。

2.2.3 アドバンスト

アドバンストセクションでは、その記事がテーマとする開発者コンテキストにおいて、サンプルコードセクションやルールブックセクションで説明できなかった、しかし注意を要する事項について記載しています。その記事がテーマとする

開発者コンテキストにまつわる、コラム的な話題や Android OS の限界に関する話題など、サンプルコードセクションやルールブックセクションの内容で解決できなかった個別課題の解決方法を検討するための考慮材料として役立てることができます。

開発者のみなさんは常に多忙です。開発者の多くは、Android の深遠なるセキュリティの構造について深く理解することよりも、ある程度の Android セキュリティの知識を持って、迅速にかつ安全な Android アプリケーションをどんどん生産することが求められます。一方、セキュリティ設計が重要なアプリケーションもあります。このようなアプリケーションの開発者は Android のセキュリティについて深く理解している必要があります。

このようにスピード重視の開発者とセキュリティ重視の開発者の両方を支援するために、このガイド文書のすべての記事はサンプルコード、ルールブック、アドバンストの 3 つのセクションに分けて記述しています。サンプルコードとルールブックセクションは「そういうことがしたければ、これをしておけば安全ですよ」といった一般化できる内容が書いてあり、可能な限りソースコードのコピー&ペーストで自動的に安全なコーディングができることを狙っています。アドバンストセクションは「こんなときはこういう問題があって、こういう考え方をするとよい」といった考えるための材料が書いてあり、開発者が取り組んでいる個別のアプリケーションで最適なセキュア設計、セキュアコーディングを検討できることを狙っています。

2.3 ガイド文書のスコープ

このガイド文書は一般の Android アプリケーション開発者に必要なセキュリティ Tips を集めることを目的としています。そのため主にマーケットで配布される Android アプリケーションの開発におけるセキュリティ Tips（下図の「アプリのセキュリティ」）が主なスコープとなっています。

図 2.3.1　Android プラットフォームの主なコンポーネント

Android OS 層以下の Android 端末実装に関するセキュリティ（上図の「端末のセキュリティ」）はスコープ外です。また Android 端末にユーザーがインストールする一般の Android アプリケーションと、Android 端末メーカーがプレインストールする Android アプリケーションでは気を付けるべきセキュリティの観点で異なるところがありますが、特に現行版においては前者のみを扱っており、後者については扱っていません。現行版では Java により実装する Tips だけを記

載しておりますが、JNI 実装についても今後の版で記載していく予定です。

root 権限が奪取される脅威についても今のところ扱っていません。基本的には root 権限が奪われていないセキュアな Android 端末を前提とし、Android OS のセキュリティモデルを活用したセキュリティ Tips をまとめています。なお、資産と脅威の扱いについては「3.1.3. 資産分類と保護施策」にて詳しく説明しておりますので、合わせてご確認ください。

2.4 Android セキュアコーディング関連書籍の紹介

このガイド文書では Android セキュアコーディングのすべてを扱うことはとてもできないので、下記で紹介する書籍を併用することをお勧めします。

- Android Security 安全なアプリケーションを作成するために著者：タオソフトウェア株式会社　ISBN978-4-8443-3134-6 https://www.amazon.co.jp/dp/4844331345/
- Java セキュアコーディングスタンダード CERT/ Oracle 版著者：Fred Long, Dhruv Mohindra, Robert C. Seacord, Dean F. Sutherland, David Svoboda 監修：歌代和正　翻訳：久保正樹, 戸田洋三　ISBN978-4-04-886070-3 https://www.amazon.co.jp/dp/4048860704/

2.5 サンプルコードの Android Studio への取り込み手順

サンプルコードの Android Studio への取り込み手順を説明します。サンプルコードは目的ごとに複数のプロジェクトにわかれています。これらのプロジェクトを取り込む方法を「2.5.1. サンプルプロジェクトを取り込む」に示します。プロジェクトの取り込みが終わったら「2.5.2. サンプルコード動作確認用 *debug.keystore* を設定する」を参照して debug.keystore ファイルを Android Studio に設定してください。なお、確認は下記の環境で行っております。

- OS
  - Windows 10 Pro
- Android Studio
  - 3.1.4
- Android SDK
  - Android 9.0(API 28)

    特に注意のないサンプルプロジェクトは Android 9.0(API28) でビルドできます。

2.5.1 サンプルプロジェクトを取り込む

2.5.1.1 サンプルコードをダウンロードする

「2.2.1. サンプルコード」で紹介した URL よりサンプルコードを取得します。

2.5.1.2 サンプルコードを展開する

Zip で圧縮されたサンプルコードを右クリックし、表示されたメニューの” すべて展開” をクリックします。

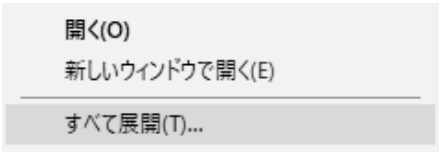

図 2.5.1　サンプルコードを展開する

2.5.1.3 展開先を指定する

ここでは”C:\android_securecoding” という名前でワークスペースを作成します。そのため、”C:\ ” を指定し” 展開” ボタンをクリックします。

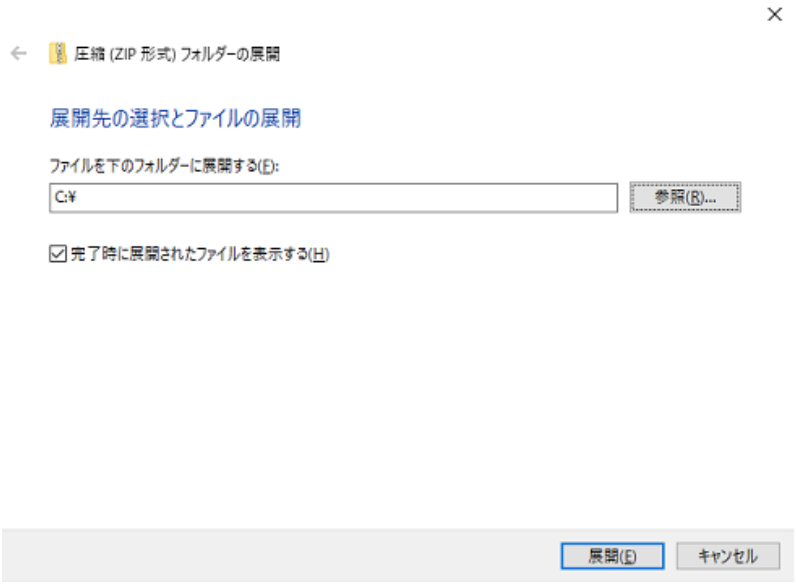

図 2.5.2　展開先を指定する

” 展開” ボタンをクリックすると”C:\” 直下に”android_securecoding” というフォルダが作成されます。

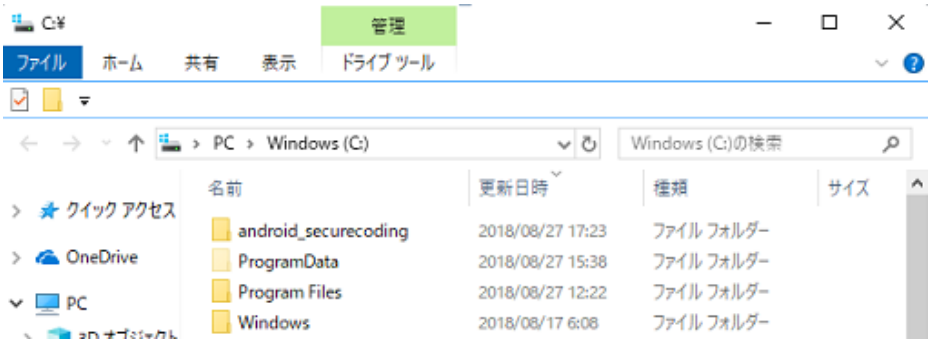

図 2.5.3　android_securecoding フォルダ

”android_securecoding” フォルダの中にはサンプルコードが含まれています。

例えば、「4.1. *Activity* を作る・利用する」の「4.1.1.3. パートナー限定 *Activity* を作る・利用する」においてサンプルコードを参照したい場合は以下をご覧ください。

```
android_securecoding
    ┗ Create Use Activity
        ┗ Activity PartnerActivity
```

以上のように、”android_securecoding” フォルダ配下は、節ごとに「サンプルコードのプロジェクト」が配置された構成となります。

2.5.1.4 Android Studio を起動しワークスペースを指定する

スタートメニューやデスクトップアイコンなどから Android Studio を起動します。

図 2.5.4　Android Studio を起動する

起動後、表示されたダイアログからプロジェクトをオープンします。

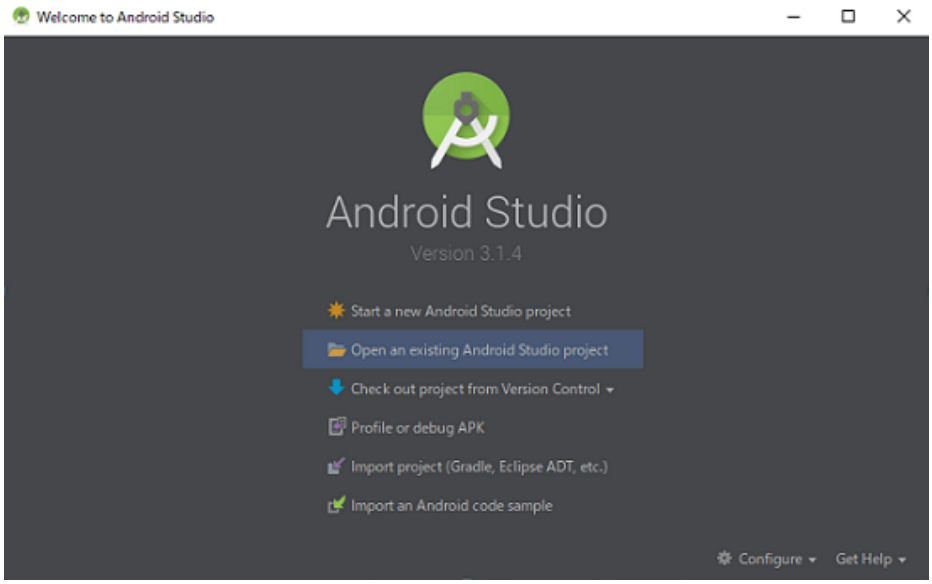

図 2.5.5　Android Studio ダイアログ

また、既にプロジェクトを読み込んでいる場合は、その Window が表示されるため、メニューより”File -> Close Project” で表示しているプロジェクトをクローズします。

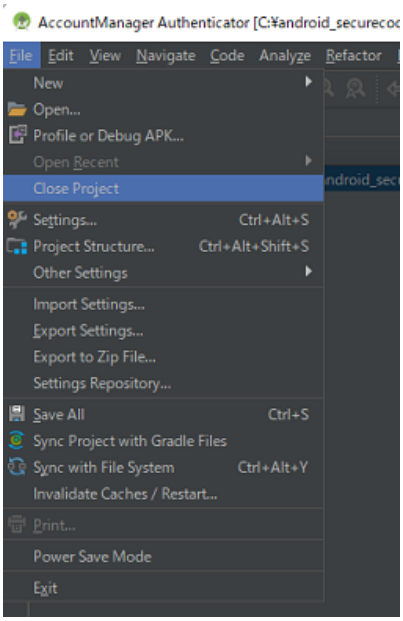

図 2.5.6　File -> Close Project

2.5.1.5 プロジェクトをオープンする

表示されているダイアログの”Open an existing Android Studio project” をクリックします。

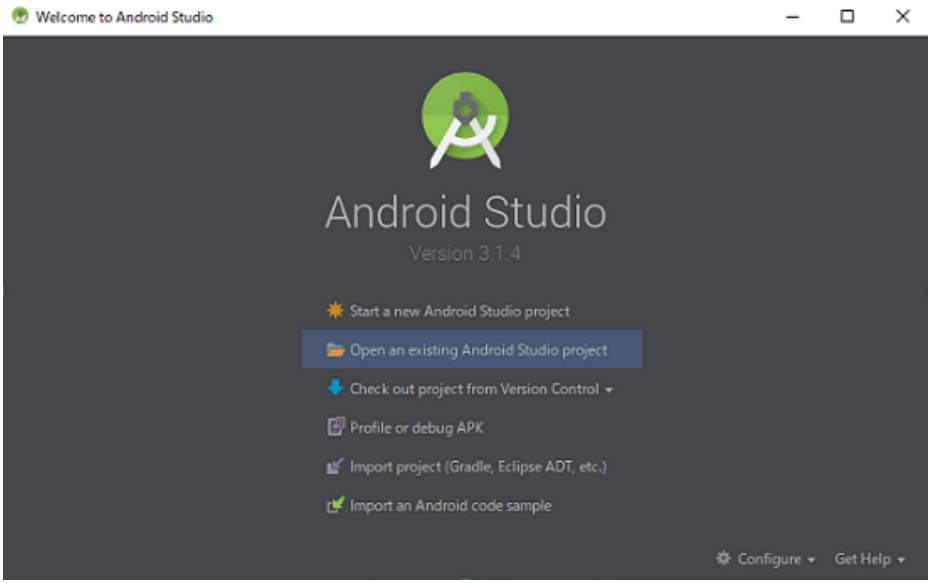

図 2.5.7　Open project

2.5.1.6 プロジェクトを選択する

オープンするプロジェクトフォルダを選択します。

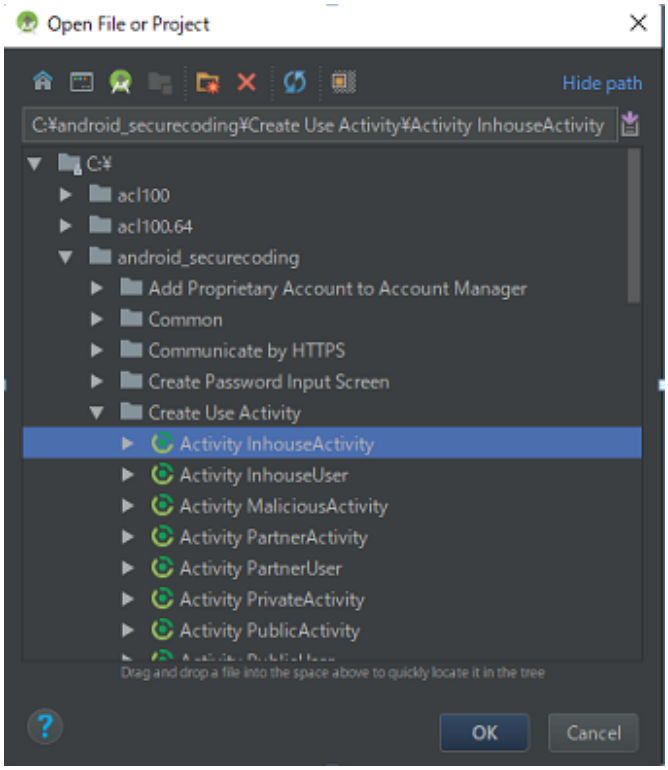

図 2.5.8　オープンするプロジェクトを選択

本ガイドのサンプルコードプロジェクトと使用している Android Studio の Gradle バージョンが異なる場合、Gradle が最適化されます。

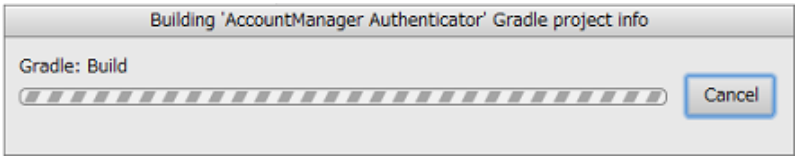

図 2.5.9　Gradle の最適化

画面に従い、”Update” をクリックし、Android Gradle Plugin のアップデートを開始してください。

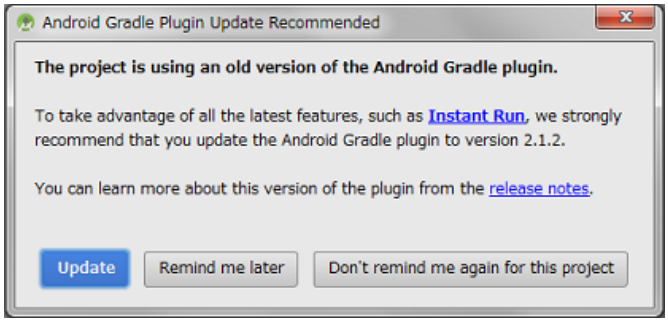

図 2.5.10　Android Gradle Plugin のアップデート

以下のメッセージが表示されるので”Fix Gradle wapper and re-import project Gradle setting” をクリックし、Gradle Wrapper の更新を行ってください。

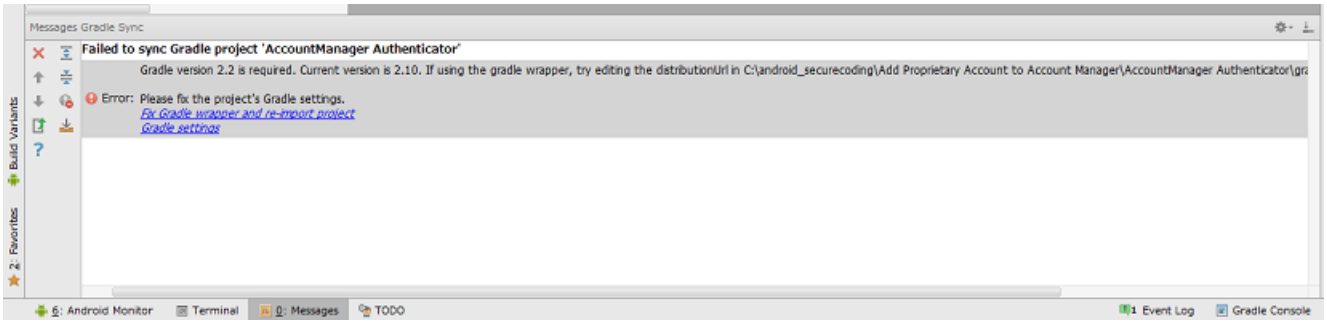

図 2.5.11　Gradle Wrapper の更新

2.5.1.7 オープンの完了

プロジェクトがオープンされ完了します。

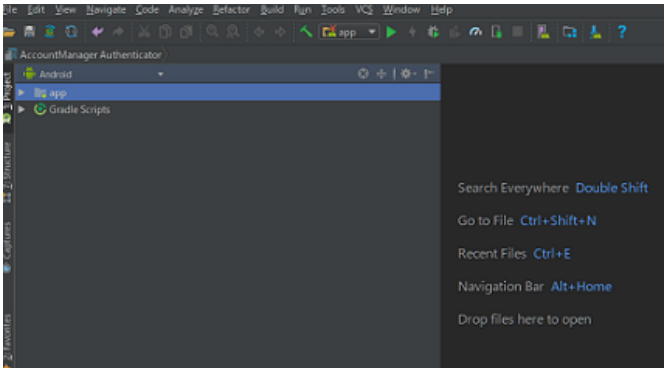

図 2.5.12　オープンの完了

Android Studio は、Eclipse とは違い、１つのプロジェクトに対して１つの Window で表示されます。違うプロジェクトをオープンする場合は、”File-> Open...” をクリックオープンします。

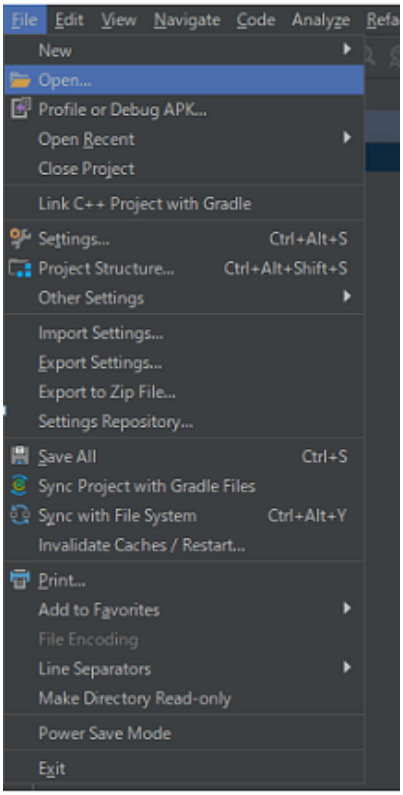

図 2.5.13　File -> Open...

2.5.2 サンプルコード動作確認用 debug.keystore を設定する

サンプルコードから作成したアプリを Android 端末やエミュレーターで動作させるためには署名が必要です。この署名に使うデバッグ用の鍵ファイル”debug.keystore” を Android Studio のプロジェクトに設定します。

2.5.2.1 File -> Project Structure... をクリックする

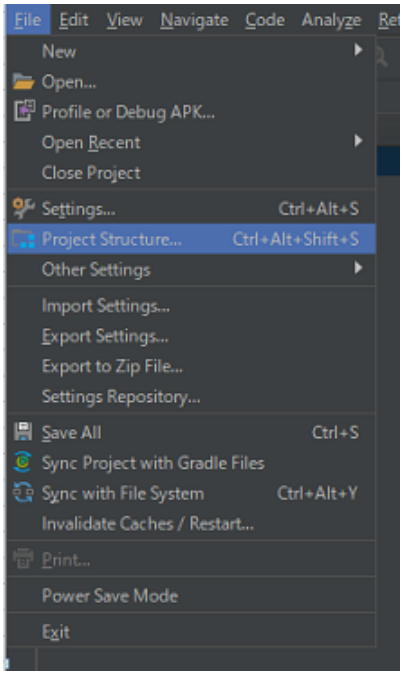

図 2.5.14　File -> Project Structure...

2.5.2.2 Signing を追加する

左欄の Modules からプロジェクト名を選択し、Signing タブを選択後、「＋」ボタンをクリックし、デフォルトの名前”config” を”debug” に変更します。

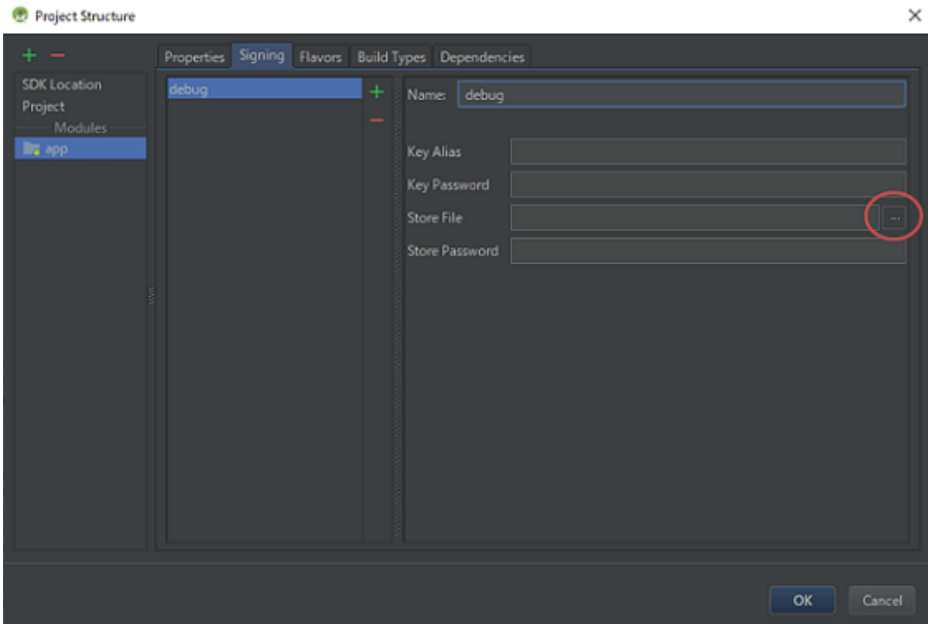

図 2.5.15　Signing を追加する

2.5.2.3 Store File として”debug.keystore” を選択する

図 2.5.15 で赤丸で囲んだボタンをクリックし、”Store File” を設定します。debug.keystore はサンプルコードに含まれています。(android_securecoding フォルダ直下)

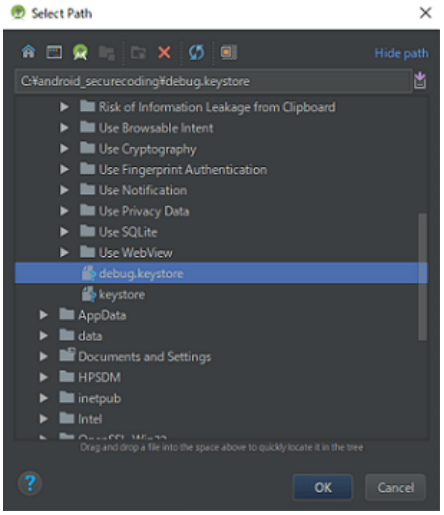

図 2.5.16　”debug.keystore” を選択する

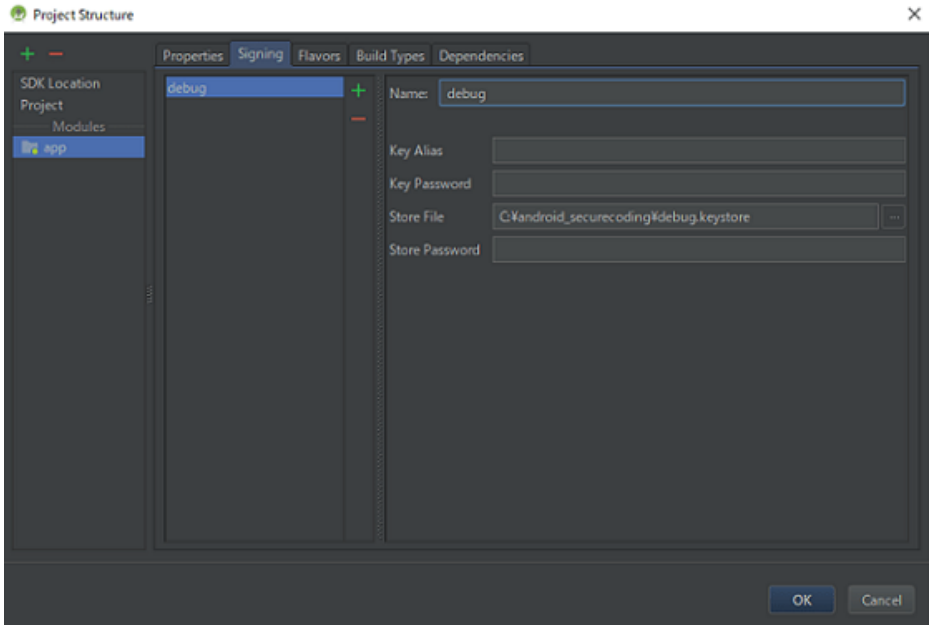

図 2.5.17　debug.keystore を選択した結果

2.5.2.4 Build Types で Signing Config を設定

Build Types タブを選択し、debug ビルド用の Signing Config を Singning で追加した”debug” を選択し、OK をクリックします。

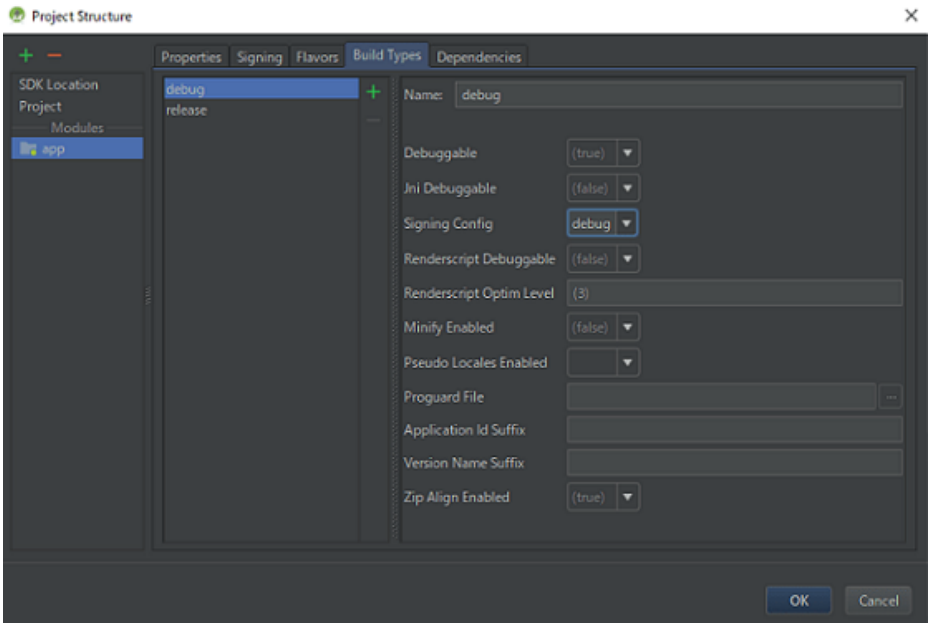

図 2.5.18　Build Types で Signing Config を設定

2.5.2.5 build.gradle ファイルで確認

signingConfigs に選択した debug.keystore のパスが表示され、buildTypes の debug に signingConfig が表示されます。

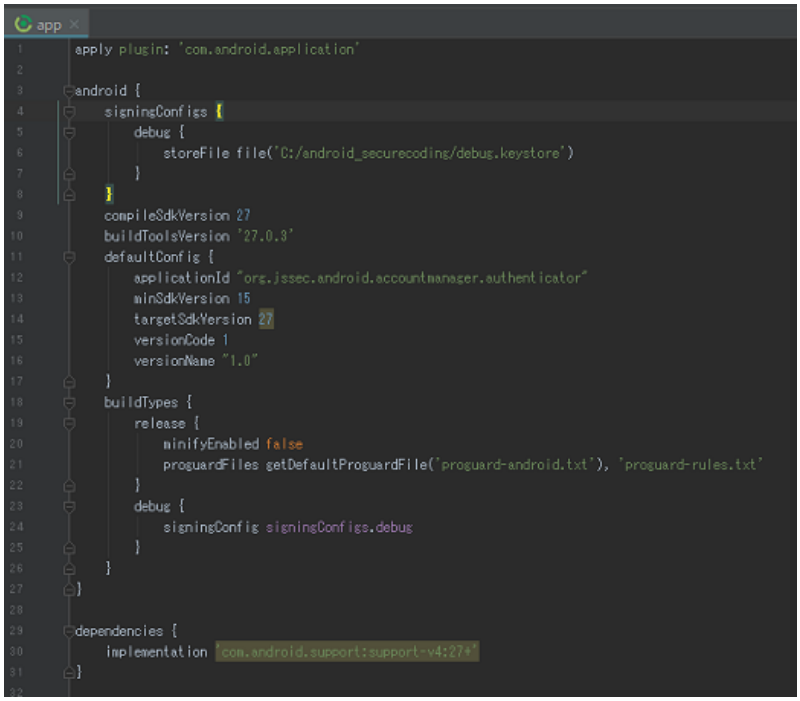

図 2.5.19　build.gradle ファイルで確認

3 セキュア設計・セキュアコーディングの基礎知識

このガイド文書は Android アプリ開発におけるセキュリティ Tips をまとめるものであるが、この章では Android スマートフォン／タブレットを例に一般的なセキュア設計・セキュアコーディングの基礎知識を扱う。後続の章において一般的なセキュア設計・セキュアコーディングの解説が必要なときに、本章の記事を参照するため、後続の章を読み進める前に本章の内容に一通り目を通しておくことをお勧めする。

3.1 Android アプリのセキュリティ

システムやアプリのセキュリティについて検討するとき、定番の考え方のフレームワークがある。まずそのシステムやアプリにおいて守るべき対象を把握する。これを「資産」と呼ぶ。次にその資産を脅かす攻撃を把握する。これを「脅威」と呼ぶ。最後に「資産」を「脅威」から守るための施策を検討・実施する。この施策を「対策」と呼ぶ。

ここで「対策」とは、システムやアプリに適切なセキュア設計・セキュアコーディングを施すことであり、このガイド文書では 4 章以降でこれを扱っている。本節では「資産」および「脅威」について焦点を当てる。

3.1.1 「資産」　守るべき対象

システムやアプリにおける「守るべき対象」には「情報」と「機能」の 2 つがある。これらをそれぞれ「情報資産」と「機能資産」と呼ぶ。「情報資産」とは、許可された人だけが参照や変更ができる情報のことであり、それ以外の人には一切参照や変更ができてはならない情報のことである。「機能資産」とは許可された人だけが利用できる機能のことであり、それ以外の人には一切利用できてはならない機能のことである。

以下、Android スマートフォン／タブレットにおける情報資産と機能資産にどのようなものがあるかを紹介する。Android アプリや Android スマートフォン／タブレットを活用したシステムを開発するときの資産の洗い出しの参考にしてほしい。以降では、Android スマートフォン／タブレットを総称して Android スマートフォンと呼ぶ。

3.1.1.1 Android スマートフォンにおける情報資産

表 3.1.1 および 表 3.1.2 は Android スマートフォンに入っている情報の一例である。これらの情報はスマートフォンユーザーに関する個人情報、プライバシー情報またはそれらに類する情報に該当するため適切な保護が必要である。

表 3.1.1: Android スマートフォンが管理する情報の例

情報	備考
電話番号	スマートフォン自身の電話番号
通話履歴	受発信の日時や相手番号
IMEI	スマートフォンの端末 ID
IMSI	回線契約者 ID
センサー情報	GPS、地磁気、加速度...
各種設定情報	WiFi 設定値...
アカウント情報	各種アカウント情報、認証情報...
メディアデータ	写真、動画、音楽、録音...
...	

表 3.1.2: アプリが管理する情報の例

情報	備考
電話帳	知人の連絡先
E メールアドレス	ユーザーのメールアドレス
Web ブックマーク	ブックマーク
Web 閲覧履歴	閲覧履歴
カレンダー	予定、ToDo、イベント...
Facebook	SNS コンテンツ...
Twitter	SNS コンテンツ...
...	

表 3.1.1 の情報は主に Android スマートフォン本体または SD カードに含まれる情報であり、表 3.1.2 の情報は主にアプリが管理する情報である。特に 表 3.1.2 の情報については、アプリがインストールされればされるほど、どんどん本体の中に増えていくことになるのである。

表 3.1.3 は電話帳の 1 件のエントリに含まれる情報である。この情報はスマートフォンユーザーに関する情報ではなく、スマートフォンユーザーの知人、友人等に関する情報である。つまりスマートフォンにはその利用者であるユーザーのみならず、ほかの人々の情報も含まれていることに注意が必要だ。

表 3.1.3　電話帳（Contacts）の 1 件のエントリに含まれる情報の例

情報	内容
電話番号	自宅、携帯電話、仕事、FAX、MMS...
E メールアドレス	自宅、仕事、携帯電話...
プロフィール画像	サムネール画像、大きな画像...
インスタントメッセンジャー	AIM、MSN、Yahoo、Skype、QQ、Google Talk、ICQ、Jabber、Netmeeting...
ニックネーム	略称、イニシャル、旧姓、別名...
住所	国、郵便番号、地域、地方、町、通り...
グループ	お気に入り、家族、友達、同僚...
ウェブサイト	ブログ、プロフィールサイト、ホームページ、FTP サーバー、自宅、会社...
イベント	誕生日、記念日、その他...
関係する人物	配偶者、子供、父、母、マネージャー、助手、 同棲関係、パートナー...
SIP アドレス	自宅、仕事、その他...
...	...

これまでの説明では主にスマートフォンユーザーの情報を紹介してきたが、アプリはユーザー以外の情報も扱っている。図 3.1.1 は 1 つのアプリが管理している情報を表しており、大きく分けるとプログラム部分とデータ部分に分かれる。プログラム部分は主にアプリメーカーの情報であり、データ部分は主にユーザーの情報である。アプリメーカーの情報の中には、勝手にユーザーに利用されたくない情報もあり得るため、そうした情報についてはユーザーが参照・変更できないような保護が必要である。

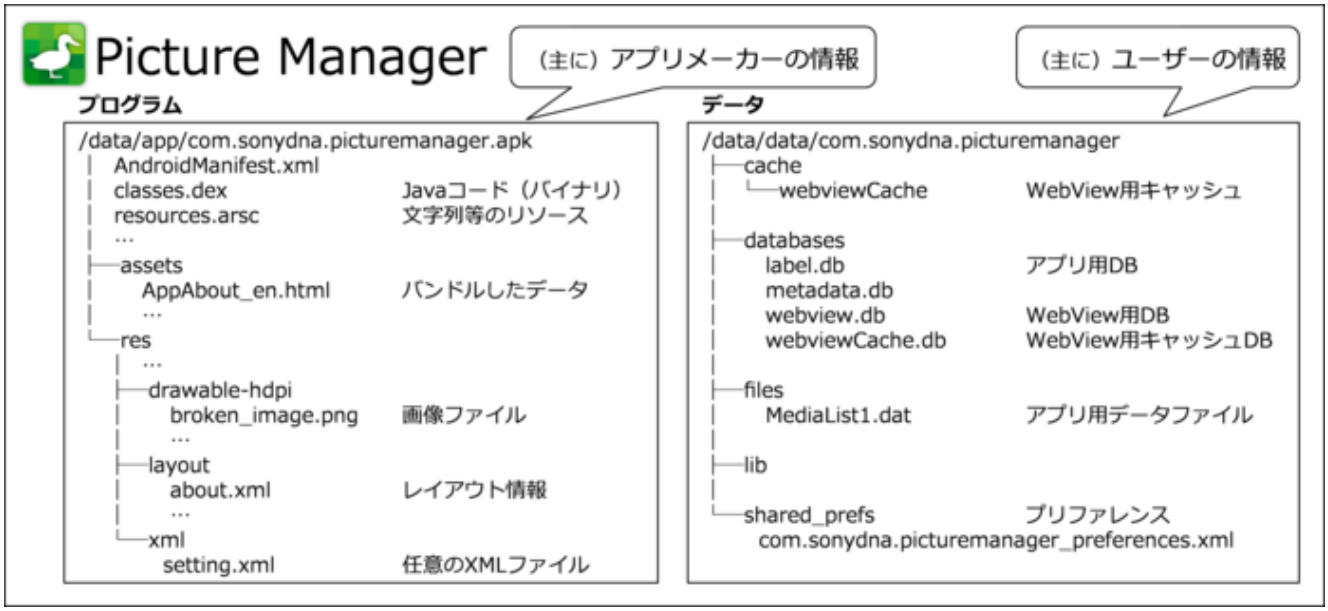

図 3.1.1　アプリが抱えている情報

Android アプリを作る場合には 図 3.1.1 のようなアプリ自身が管理する情報のみならず、表 3.1.1 、表 3.1.2 、表 3.1.3 のような Android スマートフォン本体や他のアプリから取得した情報に関しても適切に保護する必要があることにも注意が必要だ。

3.1.1.2 Android スマートフォンにおける機能資産

表 3.1.4 は Android OS がアプリに提供する機能の一例である。これらの機能がマルウェア等に勝手に利用されてしまうとユーザーの意図しない課金が生じたり、プライバシーが損なわれるなどの被害が生じたりする。そのため情報資産と同様にこうした機能資産も適切に保護されなければならない。

表 3.1.4　Android OS がアプリに提供する機能の一例

機能	機能
SMS メッセージを送受する機能	カメラ撮影機能
電話を掛ける機能	音量変更機能
ネットワーク通信機能	電話番号、携帯状態の読み取り機能
GPS 等で現在位置を得る機能	SD カード書き込み機能
Bluetooth 通信機能	システム設定変更機能
NFC 通信機能	ログデータの読み取り機能
インターネット通話（SIP）機能	実行中アプリ情報の取得機能
...	...

Android OS がアプリに提供する機能に加え、Android アプリのアプリ間連携機能も機能資産に含まれる。Android アプリはそのアプリ内で実現している機能を他のアプリから利用できるように提供することができ、このような仕組みをアプリ間連携と呼んでいる。この機能は便利である反面、Android アプリの開発者がセキュアコーディングの知識がないために、アプリ内部だけで利用する機能を誤って他のアプリから利用できるようにしてしまっているケースもある。他のアプリから利用できる機能の内容によっては、マルウェアから利用されては困ることもあるため、意図したアプリだけから利用できるように適切な保護が必要となることがある。

3.1.2 「脅威」　資産を脅かす攻撃

前節では Android スマートフォンにおける資産について解説した。ここではそれらの脅威、つまり資産を脅かす攻撃について解説する。資産が脅かされるとは簡単に言えば、情報資産が他人に勝手に参照・変更・削除・作成されることを言い、機能資産が他人に勝手に利用されることを言う、といった具合だ。こうした資産を直接的および間接的に操作する攻撃行為を「脅威」と呼ぶ。また攻撃行為を行う人や物のことを「脅威源」と呼ぶ。攻撃者やマルウェアは脅威源であって脅威ではない。攻撃者やマルウェアが行う攻撃行為のことを脅威と呼ぶのである。これら用語間の関係を 図 3.1.2 に示す。

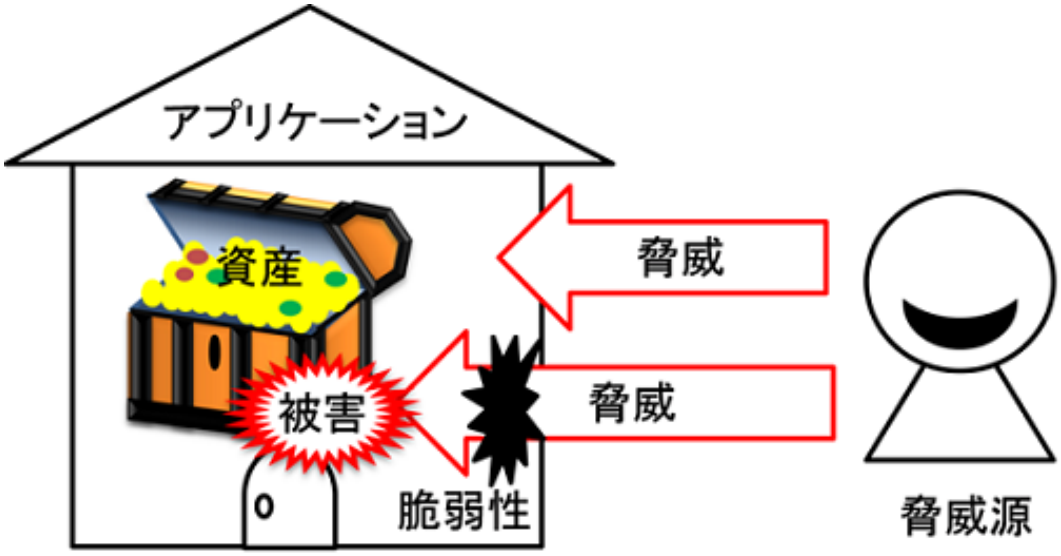

図 3.1.2　資産、脅威、脅威源、脆弱性、被害の関係

図 3.1.3 は Android アプリが動作する一般的な環境を表現したものだ。以降ではこの図をベースにして Android アプリにおける脅威の説明を展開するため、初めにこの図の見方を解説する。

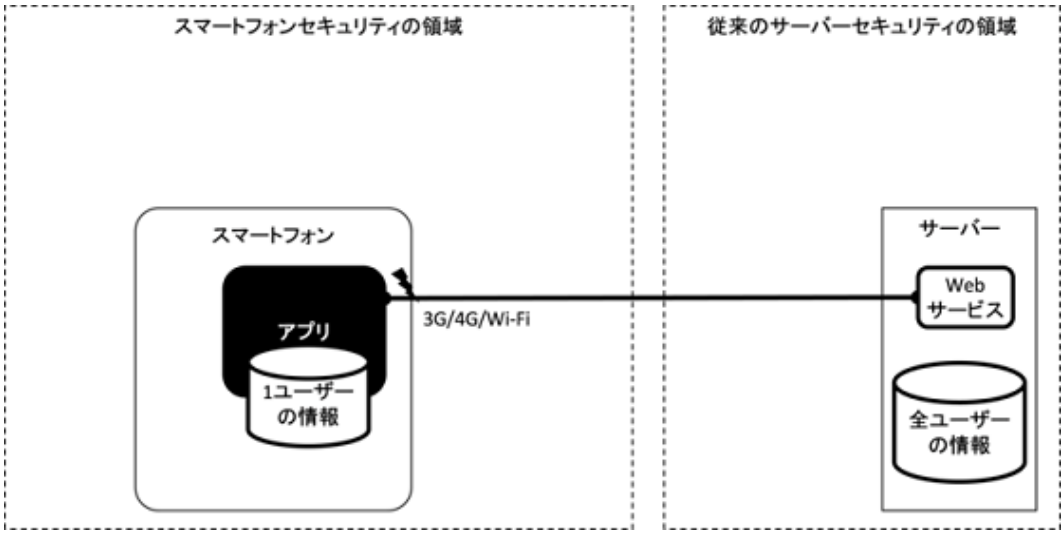

図 3.1.3　Android アプリが動作する一般的な環境

図の左右にスマートフォンとサーバーを配置している。スマートフォンやサーバーは 3G/4G/Wi-Fi およびインターネットを経由して通信している。スマートフォンの中には複数のアプリが存在するが、以降の説明で 1 つのアプリに関する脅威を説明するため、この図では 1 つのアプリに絞って説明している。スマートフォン上のアプリはそのユーザーの情報を主に扱うが、サーバー上の Web サービスは全ユーザーの情報を集中管理することを表現している。そのため従来同様にサーバーセキュリティの重要性は変わらない。サーバーセキュリティについては、このガイド文書ではスコープ外であるため言及しない。

以降ではこの図を使って Android アプリにおける脅威を説明していく。

3.1.2.1 ネットワーク上の第三者による脅威

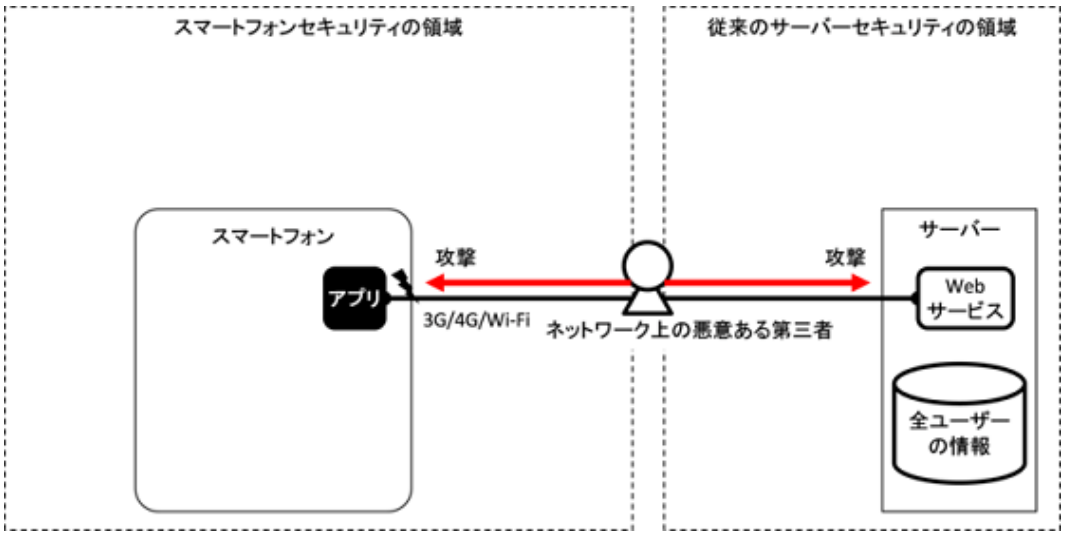

図 3.1.4　ネットワーク上の悪意ある第三者がアプリを攻撃する

スマートフォンアプリはユーザーの情報をサーバーで管理する形態が一般的である。そのため情報資産がネットワーク上を移動することになる。図 3.1.4 に示すように、ネットワーク上の悪意ある第三者は通信中の情報を参照（盗聴）したり、情報を変更（改ざん）したりしようとする。また本物のサーバーになりすまして、アプリの通信相手になろうとする。もちろん従来同様、ネットワーク上の悪意ある第三者はサーバーも攻撃する。

3.1.2.2 ユーザーがインストールしたマルウェアによる脅威

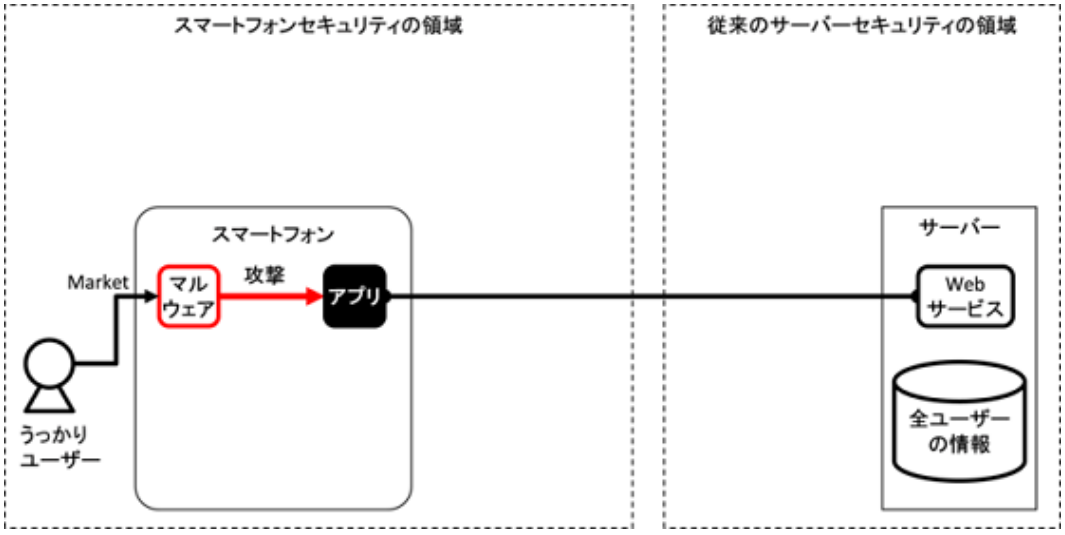

図 3.1.5　ユーザーがインストールしてしまったマルウェアがアプリを攻撃する

スマートフォンは多種多様なアプリをマーケットから入手しインストールすることで機能拡張できることがその最大の特徴である。ユーザーがうっかりマルウェアアプリをインストールしてしまうこともある。図 3.1.5 が示すように、マルウェアはアプリ間連携機能やアプリの脆弱性を悪用してアプリの情報資産や機能資産にアクセスしようとする。

3.1.2.3 アプリの脆弱性を悪用する攻撃ファイルによる脅威

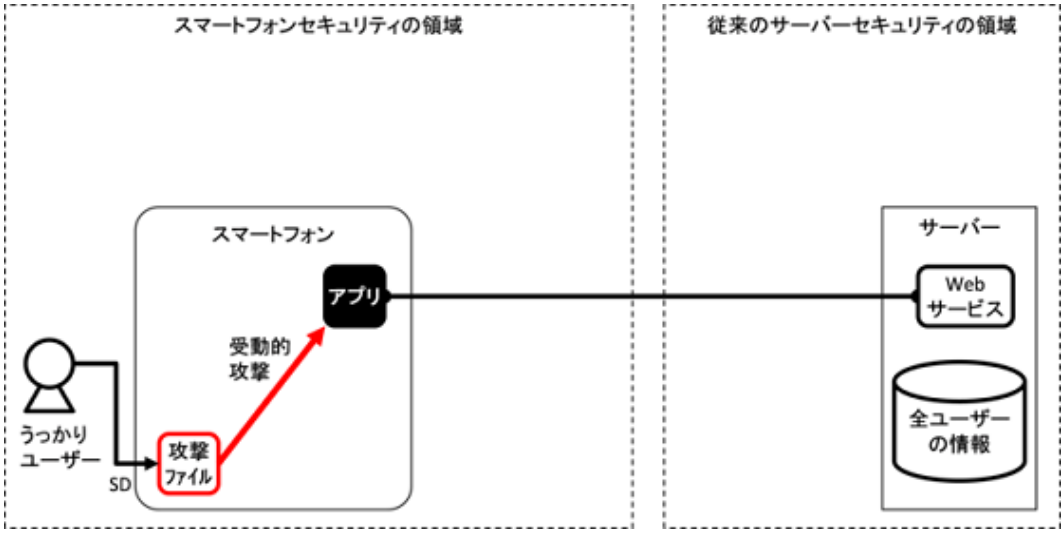

図 3.1.6　アプリの脆弱性を悪用する攻撃ファイルがアプリを攻撃する

インターネット上には音楽や写真、動画、文書など、様々なタイプのファイルが大量に公開されており、ユーザーがそれらのファイルを SD カードにダウンロードし、スマートフォンで利用する形態が一般的である。またスマートフォンで受信したメールに添付されるファイルを利用する形態も一般的である。これらのファイルは閲覧用や編集用のアプリでオープンされ利用される。

こうしたファイルを処理するアプリの機能に脆弱性があると、攻撃ファイルにより、そのアプリの情報資産や機能資産が悪用されてしまう。特に複雑なデータ構造を持ったファイル形式の処理においては脆弱性が入り込みやすい。攻撃ファイルは巧みに脆弱性を悪用してアプリを操作し、攻撃ファイルの作成者の目的を達成する。

図 3.1.6 に示すように、攻撃ファイルは脆弱なアプリによってオープンされるまでは何もせず、いったんオープンされるとアプリの脆弱性を悪用した攻撃を始める。攻撃者が能動的に行う攻撃行為と比較して、このような攻撃手法を受動的攻撃（Passive Attack）と呼ぶ。

3.1.2.4 悪意あるスマートフォンユーザーによる脅威

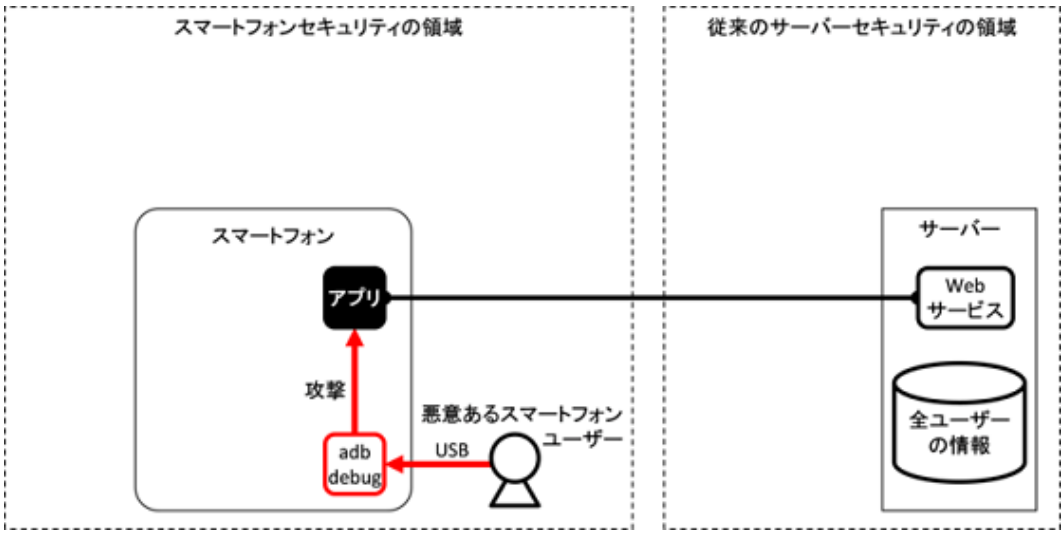

図 3.1.7　悪意あるスマートフォンユーザーがアプリを攻撃する

Android スマートフォンのアプリ開発においては、一般ユーザーに対してアプリを開発、解析する環境や機能が公式に提供されている。提供されている機能の中でも、特に ADB と呼ばれる充実したデバッグ機能は、誰でも何の登録・審査もなく利用可能であり、この機能により Android スマートフォンユーザーは OS 解析行為やアプリ解析行為を容易に行うことができる。

図 3.1.7 に示すように、悪意あるスマートフォンユーザーは ADB 等のデバッグ機能を利用してアプリを解析し、アプリが抱える情報資産や機能資産にアクセスしようとする。アプリが抱える資産がユーザー自身のものである場合には問題とならないが、アプリメーカー等ユーザー以外のステークホルダーの資産である場合に問題となる。このようにスマートフォンのユーザー自身が悪意を持ってアプリ内の資産を狙うことがあることにも注意が必要だ。

3.1.2.5 スマートフォンの近くにいる第三者による脅威

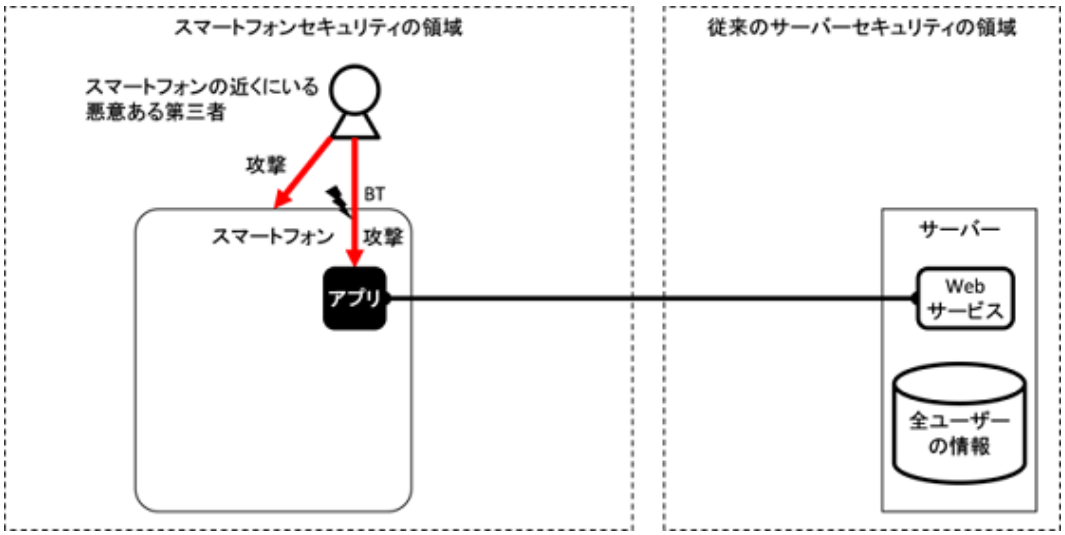

図 3.1.8　スマートフォンの近くにいる悪意ある第三者がアプリを攻撃する

スマートフォンはその携帯性の良さや、Bluetooth 等の近距離無線通信機能を標準搭載していることから、物理的にスマートフォンの近くにいる悪意ある第三者から攻撃され得ることも忘れてはならない。攻撃者はユーザーが入力中のパスワードを肩越しに覗き見したり、図 3.1.8 に示すように、Bluetooth 通信機能を持つアプリに対して Bluetooth でアクセスしたり、スマートフォン自体を盗んだり破壊したりする。特にスマートフォン自体の窃盗や破壊については、機密度の高さからスマートフォン内から一切外に出さない運用としている情報資産が失われてしまう脅威であり、アプリ設計の段階で見過ごされてしまうこともあるので注意が必要だ。

3.1.2.6 様々な脅威

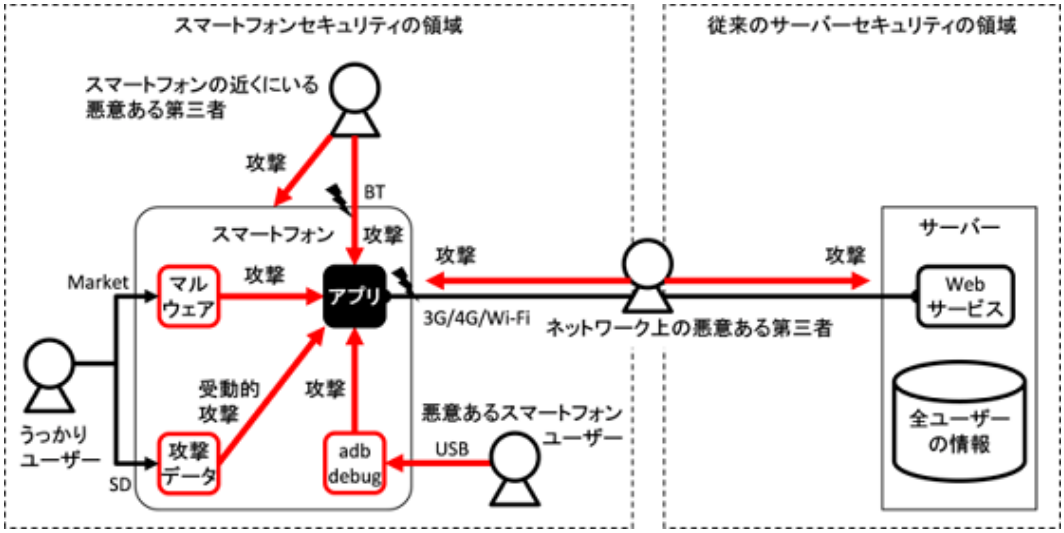

図 3.1.9　スマートフォンアプリは様々な攻撃にさらされている

図 3.1.9 はこれまで説明した脅威をひとまとめにした図である。このようにスマートフォンアプリを取り巻く脅威には様々なものがあり、この図はそのすべてを書き出しているものではない。日々の情報収集により、Android アプリを取り巻く脅威について認識を広め、アプリのセキュア設計・セキュアコーディングに活かしていく努力が必要である。一般社団法人日本スマートフォンセキュリティ協会が作成した次の文書もスマートフォンの脅威について役立つ情報を提供しているので参考にしていただきたい。

- 『スマートフォン＆タブレットの業務利用に関するセキュリティガイドライン』【第二版】https://www.jssec.org/dl/guidelines_v2.pdf https://www.jssec.org/dl/guidelines2012Enew_v1.0.pdf (English)
- 『スマートフォンネットワークセキュリティ実装ガイド』【第一版】https://www.jssec.org/dl/NetworkSecurityGuide1.pdf
- 『スマートフォンの業務利用におけるクラウド活用ガイド』【ベータ版】https://www.jssec.org/dl/cloudguide2012_beta.pdf
- 『MDM 導入・運用検討ガイド』【第一版】https://www.jssec.org/dl/MDMGuideV1.pdf

3.1.3 資産分類と保護施策

前節までで解説した通り Android スマートフォンには様々な脅威が存在する。それらの脅威から、アプリが扱うすべての資産を保護することは、開発にかかる時間や技術的限界などから困難な場合がある。そのため、Android アプリ開発者は、アプリが扱う資産の重要度や容認される被害レベルを判断基準とした優先度に応じて、資産に対する妥当な対策を検討することが必要になる。

資産毎の保護施策を決めるため、アプリが扱うそれぞれの資産について、その重要度・保護に関する法的根拠・被害発生時の影響・開発者（または組織）の社会的責任などを検討した上で、資産を分類し、分類毎に保護施策のレベルを定める。これが、それぞれの資産をどのように扱い、どのような対策を施すかを決める判断基準となる。この分類はアプリ開発者（または組織）として資産をどのように扱い保護するかを定める基準そのものであるため、アプリ開発者（または組織）がそれぞれの事情に合わせて、分類方法や対策内容を定める必要がある。

参考までに、本ガイドにおける資産分類と保護施策のレベルを以下に示す。

表 3.1.5: 資産分類と保護施策のレベル

資産分類	資産のレベル	保護施策のレベル
高位[*1]	資産が被害にあった場合、組織または個人の活動に致命的または壊滅的な影響をあたえるもの。例）資産が被害にあった場合、当該組織が事業を継続できなくなるレベル。	Android OS のセキュリティモデルを破る、root 権限を奪取した状態からの攻撃や APK の dex 部分を改造するといった、高度な攻撃に対しても保護する。UX 等の他要素よりもセキュリティ確保を優先する。
中位	資産が被害にあった場合、組織または個人の活動に重大な影響をあたえるもの。例）資産が被害にあった場合、当該組織の利益が悪化し、事業に影響を及ぼすレベル。	Android OS のセキュリティモデルを活用し、その範囲内で保護する。UX 等の他要素よりもセキュリティ確保を優先する。
低位	資産が被害にあった場合、組織または個人の活動に限定的な影響をあたえるもの。例）資産が被害にあった場合、当該組織の利益に影響を与え得るが、他の要素により利益の補てんが可能なレベル。	UX 等の他要素とセキュリティを比較し、他要素を優先しても良い。

本ガイドにおける資産分類と保護施策は、基本的には root 権限が奪われていないセキュアな Android 端末を前提とし、Android OS のセキュリティモデルを活用したセキュリティ施策を基準にしている。具体的には、資産分類で中位レベル以下の資産に対して Android OS のセキュリティモデルが機能していることを前提に、Android OS のセキュリティモデルを活用した保護施策を想定している。

3.1.4 センシティブな情報

本ガイドのこれ以降の文章において情報資産を「センシティブな情報」と表現している。前節で述べたようにアプリが扱う個々の情報資産ごとに資産のレベルや保護施策のレベルを判断しなければならない。

3.2 入力データの安全性を確認する

入力データの安全性確認はもっとも基礎的で効果の高いセキュアコーディング作法である。プログラムに入力されるデータのうち、攻撃者が直接的、間接的にそのデータの値を操作可能であるものはすべて、入力データの安全性確認が必要である。以下、Activity をプログラムに見立て、Intent を入力データに見立てた場合を例にして、入力データの安全性確認の在り方について解説する。

Activity が受け取った Intent には攻撃者が細工した　データが含まれている可能性がある。攻撃者はプログラマが想定していない形式・値のデータを送り付けることで、アプリの誤動作を誘発し、結果として何らかのセキュリティ被害を生じさせるのである。ユーザーも攻撃者の一人となり得ることも忘れてはならない。

Intent は action や data、extras などのデータで構成されるが、攻撃者が制御可能なデータはすべて気を付けなければならない。攻撃者が制御可能なデータを扱うコードでは、必ず次の事項を確認しなければならない。

[*1] 高位レベルの資産は、root 権限が奪取された状態からの攻撃や、APK 解析・改造による攻撃といった、Android OS のセキュリティモデルが破られた状態での攻撃からも、保護が必要な資産であると想定している。このような資産を守るためには、Android OS のセキュリティモデルが活用できないため、暗号化、難読化、ハードウェア支援、サーバー支援など複数の手段を組み合わせて高度な防御設計をする。これはガイド文書に簡潔に書けるようなノウハウではないし、個別の状況に応じて適切な防御設計は異なるため本ガイドの対象外としている。root 権限奪取からの攻撃や APK 解析・改造などの高度な攻撃に対する保護が必要な場合は、Android の耐タンパ設計に詳しいセキュリティ専門家に相談することをお勧めする。

(a) 受け取ったデータは、プログラマが想定した形式であって、値は想定の範囲内に収まっているか？

(b) 想定している形式、値のあらゆるデータを受け取っても、そのデータを扱うコードが想定外の動作をしないと保証できるか？

次の例は指定 URL の Internet 上の Web ページの HTML を取得し、画面上の TextView に表示するだけの簡単なサンプルである。しかしこれには不具合がある。

Internet 上の Web ページの HTML を TextView に表示するサンプルコード

```
TextView tv = (TextView) findViewById(R.id.textview);
InputStreamReader isr = null;
char[] text = new char[1024];
int read;
try {
    String urlstr = getIntent().getStringExtra("WEBPAGE_URL");
    URL url = new URL(urlstr);
    isr = new InputStreamReader(url.openConnection().getInputStream());
    while ((read=isr.read(text)) != -1) {
        tv.append(new String(text, 0, read));
    }
} catch (MalformedURLException e) { //...
```

(a) の観点で「urlstr が正しい URL である」ことを new URL() で MalformedURLException が発生しないことに　より確認している。しかしこれは不十分であり、urlstr に「file://〜」形式の URL が指定されると Internet 上の Web ページではなく、内部ファイルシステム上のファイルを開いて TextView に表示してしまう。プログラマが想定した動作を保証していないため、(b) の観点を満たしていない。

次は改善例である。(a) の観点で「urlstr は正規の URL であって、protocol は http または https に限定される」ことを確認している。これにより (b) の観点でも url.openConnection().getInputStream() で Internet 経由の InputStream を取得することが保証される。

Internet 上の Web ページの HTML を TextView に表示するサンプルコードの修正版

```
TextView tv = (TextView) findViewById(R.id.textview);
InputStreamReader isr = null;
char[] text = new char[1024];
int read;
try {
    String urlstr = getIntent().getStringExtra("WEBPAGE_URL");
    URL url = new URL(urlstr);
    String prot = url.getProtocol();
    if (!"http".equals(prot) && !"https".equals(prot)) {
        throw new MalformedURLException("invalid protocol");
    }
    isr = new InputStreamReader(url.openConnection().getInputStream());
    while ((read=isr.read(text)) != -1) {
        tv.append(new String(text, 0, read));
    }
} catch (MalformedURLException e) { //...
```

入力データの安全性確認は Input Validation と呼ばれる基礎的なセキュアコーディング作法である。Input Validation という言葉の語感から (a) の観点のみ気を付けて (b) の観点を忘れてしまいがちである。データはプログラムに入ってきたときではなく、プログラムがそのデータを「使う」ときに被害が発生することに気を付けていただきたい。下記 URL もぜひ参考にしていただきたい。

- IPA 「セキュア・プログラミング講座」https://www.ipa.go.jp/security/awareness/vendor/programming/index.html
- JPCERT CC 「Java セキュアコーディングスタンダード CERT/Oracle 版」https://www.jpcert.or.jp/java-rules/
- JPCERT CC「Java Android アプリケーション開発へのルールの適用」https://www.jpcert.or.jp/java-rules/android-j.html

4 安全にテクノロジーを活用する

Android で言えば Activity や SQLite など、テクノロジーごとにセキュリティ観点の癖というものがある。そうしたセキュリティの癖を知らずに設計、コーディングしていると思わぬ脆弱性をつくりこんでしまうことがある。この章では開発者が Android のテクノロジーを活用するシーンを想定した記事を扱う。

4.1 Activity を作る・利用する

4.1.1 サンプルコード

Activity がどのように利用されるかによって、Activity が抱えるリスクや適切な防御手段が異なる。ここでは、Activity がどのように利用されるかという観点で、Activity を 4 つのタイプに分類した。次の判定フローによって作成する Activity がどのタイプであるかを判断できる。なお、どのような相手を利用するかによって適切な防御手段が決まるため、Activity の利用側の実装についても合わせて説明する。

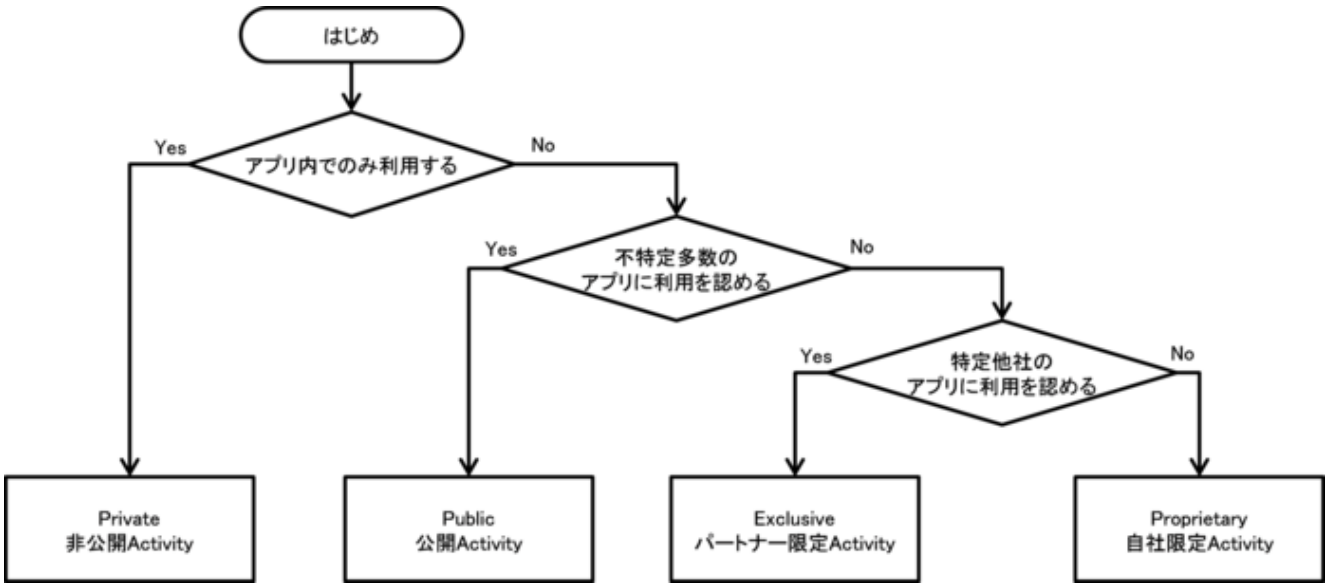

図 4.1.1　Activity タイプ選択フロー

4.1.1.1 非公開 Activity を作る・利用する

非公開 Activity は、同一アプリ内でのみ利用される Activity であり、もっとも安全性の高い Activity である。

同一アプリ内だけで利用される Activity（非公開 Activity）を利用する際は、クラスを指定する明示的 Intent を使えば誤って外部アプリに Intent を送信してしまうことがない。ただし、Activity を呼び出す際に使用する Intent は第三者によって読み取られる恐れがある。そのため、Activity に送信する Intent にセンシティブな情報を格納する場合には、その情報が悪意のある第三者に読み取られることのないように、適切な対応を実施する必要がある。

以下に非公開 Activity を作る側のサンプルコードを示す。

ポイント (Activity を作る)：

1. taskAffinity を指定しない
2. launchMode を指定しない
3. exported=”false” により、明示的に非公開設定する
4. 同一アプリからの Intent であっても、受信 Intent の安全性を確認する
5. 利用元アプリは同一アプリであるから、センシティブな情報を返送してよい

Activity を非公開設定するには、AndroidManifest.xml の activity 要素の exported 属性を false と指定する。

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="org.jssec.android.activity.privateactivity" >

    <application
        android:allowBackup="false"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name" >

        <!-- 非公開 Activity -->
        <!-- ★ポイント 1★ taskAffinity を指定しない -->
        <!-- ★ポイント 2★ launchMode を指定しない -->
        <!-- ★ポイント 3★ exported="false"により、明示的に非公開設定する -->
        <activity
            android:name=".PrivateActivity"
            android:label="@string/app_name"
            android:exported="false" />

        <!-- ランチャーから起動する公開 Activity -->
        <activity
            android:name=".PrivateUserActivity"
            android:label="@string/app_name"
            android:exported="true" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

PrivateActivity.java

```
package org.jssec.android.activity.privateactivity;
```

(continues on next page)

(continued from previous page)

```
import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.widget.Toast;

public class PrivateActivity extends Activity {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.private_activity);

        // ★ポイント 4★ 同一アプリからの Intent であっても、受信 Intent の安全性を確認する
        // サンプルにつき割愛。「3.2 入力データの安全性を確認する」を参照。
        String param = getIntent().getStringExtra("PARAM");
        Toast.makeText(this, String.format("パラメータ「%s」を受け取った。", param), Toast.LENGTH_LONG).
↪show();
    }

    public void onReturnResultClick(View view) {

        // ★ポイント 5★ 利用元アプリは同一アプリであるから、センシティブな情報を返送してよい
        Intent intent = new Intent();
        intent.putExtra("RESULT", "センシティブな情報");
        setResult(RESULT_OK, intent);
        finish();
    }
}
```

次に非公開 Activity を利用する側のサンプルコードを示す。

ポイント (Activity を利用する) :

6. Activity に送信する Intent には、フラグ FLAG_ACTIVITY_NEW_TASK を設定しない
7. 同一アプリ内 Activity はクラス指定の明示的 Intent で呼び出す
8. 利用先アプリは同一アプリであるから、センシティブな情報を putExtra() を使う場合に限り送信してもよい[*1]
9. 同一アプリ内 Activity からの結果情報であっても、受信データの安全性を確認する

PrivateUserActivity.java

```
package org.jssec.android.activity.privateactivity;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.widget.Toast;

public class PrivateUserActivity extends Activity {
```

(continues on next page)

[*1] ただし、ポイント 1, 2, 6 を遵守している場合を除いては Intent が第三者に読み取られるおそれがあることに注意する必要がある。詳細はルールブックセクションの 4.1.2.2.、4.1.2.3. を参照すること。

(continued from previous page)

```
    private static final int REQUEST_CODE = 1;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.user_activity);
    }

    public void onUseActivityClick(View view) {

        // ★ポイント 6★ Activity に送信する Intent には、フラグ FLAG_ACTIVITY_NEW_TASK を設定しない
        // ★ポイント 7★ 同一アプリ内 Activity はクラス指定の明示的 Intent で呼び出す
        Intent intent = new Intent(this, PrivateActivity.class);

        // ★ポイント 8★ 利用先アプリは同一アプリであるから、センシティブな情報を putExtra() を使う場合に限り送信してもよい
        intent.putExtra("PARAM", "センシティブな情報");

        startActivityForResult(intent, REQUEST_CODE);
    }

    @Override
    public void onActivityResult(int requestCode, int resultCode, Intent data) {
        super.onActivityResult(requestCode, resultCode, data);

        if (resultCode != RESULT_OK) return;

        switch (requestCode) {
        case REQUEST_CODE:
            String result = data.getStringExtra("RESULT");

            // ★ポイント 9★ 同一アプリ内 Activity からの結果情報であっても、受信データの安全性を確認する
            // サンプルにつき割愛。「3.2 入力データの安全性を確認する」を参照。
            Toast.makeText(this, String.format("結果「%s」を受け取った。", result), Toast.LENGTH_LONG).
↪show();
            break;
        }
    }
}
```

4.1.1.2 公開 Activity を作る・利用する

公開 Activity は、不特定多数のアプリに利用されることを想定した Activity である。マルウェアが送信した Intent を受信することがあることに注意が必要である。また、公開 Activity を利用する場合には、送信する Intent がマルウェアに受信される、あるいは読み取られることがあることに注意が必要である。

以下に公開 Activity を作る側のサンプルコードを示す。

ポイント (Activity を作る) :

1. exported=”true” により、明示的に公開設定する
2. 受信 Intent の安全性を確認する
3. 結果を返す場合、センシティブな情報を含めない

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="org.jssec.android.activity.publicactivity" >


    <application
        android:allowBackup="false"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name" >

        <!-- 公開 Activity -->
        <!-- ★ポイント 1★ exported="true"により、明示的に公開設定する -->
        <activity
            android:name=".PublicActivity"
            android:label="@string/app_name"
            android:exported="true" >

            <!-- Action 指定による暗黙的 Intent を受信するように Intent Filter を定義 -->
            <intent-filter>
                <action android:name="org.jssec.android.activity.MY_ACTION" />
                <category android:name="android.intent.category.DEFAULT" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

PublicActivity.java

```
package org.jssec.android.activity.publicactivity;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.widget.Toast;

public class PublicActivity extends Activity {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        // ★ポイント 2★ 受信 Intent の安全性を確認する
        // 公開 Activity であるため利用元アプリがマルウェアである可能性がある。
        // サンプルにつき割愛。「3.2 入力データの安全性を確認する」を参照。
        String param = getIntent().getStringExtra("PARAM");
        Toast.makeText(this, String.format("パラメータ「%s」を受け取った。", param), Toast.LENGTH_LONG).
↪show();
    }

    public void onReturnResultClick(View view) {

        // ★ポイント 3★ 結果を返す場合、センシティブな情報を含めない
        // 公開 Activity であるため利用元アプリがマルウェアである可能性がある。
```

(continues on next page)

(continued from previous page)

```
        // マルウェアに取得されても問題のない情報であれば結果として返してもよい。

        Intent intent = new Intent();
        intent.putExtra("RESULT", "センシティブではない情報");
        setResult(RESULT_OK, intent);
        finish();
    }
}
```

次に公開 Activity を利用する側のサンプルコードを示す。

ポイント (Activity を利用する) :

4. センシティブな情報を送信してはならない

5. 結果を受け取る場合、結果データの安全性を確認する

PublicUserActivity.java

```
package org.jssec.android.activity.publicuser;

import android.app.Activity;
import android.content.ActivityNotFoundException;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.widget.Toast;

public class PublicUserActivity extends Activity {

    private static final int REQUEST_CODE = 1;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }

    public void onUseActivityClick(View view) {

        try {
            // ★ポイント 4★ センシティブな情報を送信してはならない
            Intent intent = new Intent("org.jssec.android.activity.MY_ACTION");
            intent.putExtra("PARAM", "センシティブではない情報");
            startActivityForResult(intent, REQUEST_CODE);
        } catch (ActivityNotFoundException e) {
            Toast.makeText(this, "利用先 Activity が見つからない。", Toast.LENGTH_LONG).show();
        }
    }

    @Override
    public void onActivityResult(int requestCode, int resultCode, Intent data) {
        super.onActivityResult(requestCode, resultCode, data);

        // ★ポイント 5★ 結果を受け取る場合、結果データの安全性を確認する
        // サンプルにつき割愛。「3.2 入力データの安全性を確認する」を参照。
```

(continues on next page)

(continued from previous page)

```
        if (resultCode != RESULT_OK) return;
        switch (requestCode) {
        case REQUEST_CODE:
            String result = data.getStringExtra("RESULT");
            Toast.makeText(this, String.format("結果「%s」を受け取った。", result), Toast.LENGTH_LONG).
↪show();
            break;
        }
    }
}
```

4.1.1.3 パートナー限定 **Activity** を作る・利用する

パートナー限定 Activity は、特定のアプリだけから利用できる Activity である。パートナー企業のアプリと自社アプリが連携してシステムを構成し、パートナーアプリとの間で扱う情報や機能を守るために利用される。

Activity を呼び出す際に使用する Intent は第三者によって読み取られる恐れがある。そのため、Activity に送信する Intent にセンシティブな情報を格納する場合には、その情報が悪意のある第三者に読み取られることのないように、適切な対応を実施する必要がある。

以下にパートナー限定 Activity を作る側のサンプルコードを示す。

ポイント (Activity を作る) :

1. taskAffinity を指定しない
2. launchMode を指定しない
3. Intent Filter を定義せず、exported=”true” を明示的に設定する
4. 利用元アプリの証明書がホワイトリストに登録されていることを確認する
5. パートナーアプリからの Intent であっても、受信 Intent の安全性を確認する
6. パートナーアプリに開示してよい情報に限り返送してよい

ホワイトリストを用いたアプリの確認方法については、「4.1.3.2. 利用元アプリを確認する」を参照すること。また、ホワイトリストに指定する利用先アプリの証明書ハッシュ値の確認方法は「5.2.1.3. アプリの証明書のハッシュ値を確認する方法」を参照すること。

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="org.jssec.android.activity.partneractivity" >

    <application
        android:allowBackup="false"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name" >

        <!-- パートナー限定 Activity -->
        <!-- ★ポイント 1★ taskAffinity を指定しない -->
        <!-- ★ポイント 2★ launchMode を指定しない -->
        <!-- ★ポイント 3★ Intent Filter を定義せず、exported="true"を明示的に設定する -->
```

(continues on next page)

(continued from previous page)

```
        <activity
            android:name=".PartnerActivity"
            android:exported="true" />

    </application>
</manifest>
```

PartnerActivity.java

```
package org.jssec.android.activity.partneractivity;

import org.jssec.android.shared.PkgCertWhitelists;
import org.jssec.android.shared.Utils;

import android.app.Activity;
import android.content.Context;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.widget.Toast;

public class PartnerActivity extends Activity {

    // ★ポイント 4★ 利用元アプリの証明書がホワイトリストに登録されていることを確認する
    private static PkgCertWhitelists sWhitelists = null;
    private static void buildWhitelists(Context context) {
        boolean isdebug = Utils.isDebuggable(context);
        sWhitelists = new PkgCertWhitelists();

        // パートナーアプリ org.jssec.android.activity.partneruser の証明書ハッシュ値を登録
        sWhitelists.add("org.jssec.android.activity.partneruser", isdebug ?
                // debug.keystore の"androiddebugkey"の証明書ハッシュ値
                "0EFB7236 328348A9 89718BAD DF57F544 D5CCB4AE B9DB34BC 1E29DD26 F77C8255" :
                // keystore の"partner key"の証明書ハッシュ値
                "1F039BB5 7861C27A 3916C778 8E78CE00 690B3974 3EB8259F E2627B8D 4C0EC35A");

        // 以下同様に他のパートナーアプリを登録...
    }
    private static boolean checkPartner(Context context, String pkgname) {
        if (sWhitelists == null) buildWhitelists(context);
        return sWhitelists.test(context, pkgname);
    }

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        // ★ポイント 4★ 利用元アプリの証明書がホワイトリストに登録されていることを確認する
        if (!checkPartner(this, getCallingActivity().getPackageName())) {
            Toast.makeText(this, "利用元アプリはパートナーアプリではない。", Toast.LENGTH_LONG).show();
            finish();
            return;
        }
```

(continues on next page)

(continued from previous page)

```
        // ★ポイント 5★ パートナーアプリからの Intent であっても、受信 Intent の安全性を確認する
        // サンプルにつき割愛。「3.2 入力データの安全性を確認する」を参照。
        Toast.makeText(this, "パートナーアプリからアクセスあり", Toast.LENGTH_LONG).show();
    }

    public void onReturnResultClick(View view) {

        // ★ポイント 6★ パートナーアプリに開示してよい情報に限り返送してよい
        Intent intent = new Intent();
        intent.putExtra("RESULT", "パートナーアプリに開示してよい情報");
        setResult(RESULT_OK, intent);
        finish();
    }
}
```

PkgCertWhitelists.java

```
package org.jssec.android.shared;

import android.content.pm.PackageManager;
import java.util.HashMap;
import java.util.Map;
import android.content.Context;
import android.os.Build;

import static android.content.pm.PackageManager.CERT_INPUT_SHA256;

public class PkgCertWhitelists {
    private Map<String, String> mWhitelists = new HashMap<String, String>();

    public boolean add(String pkgname, String sha256) {
        if (pkgname == null) return false;
        if (sha256 == null) return false;

        sha256 = sha256.replaceAll(" ", "");
        if (sha256.length() != 64) return false;    // SHA-256 は 32 バイト
        sha256 = sha256.toUpperCase();
        if (sha256.replaceAll("[0-9A-F]+", "").length() != 0) return false; // 0-9A-F 以外の文字がある

        mWhitelists.put(pkgname, sha256);
        return true;
    }

    public boolean test(Context ctx, String pkgname) {
        // pkgname に対応する正解のハッシュ値を取得する
        String correctHash = mWhitelists.get(pkgname);
        android.util.Log.d("Partner", "hash=" + correctHash);
        // pkgname の実際のハッシュ値と正解のハッシュ値を比較する
        if (Build.VERSION.SDK_INT >= 28) {
            // ★ API Level >= 28 では Package Manager の API で直接検証が可能
            PackageManager pm = ctx.getPackageManager();
            return pm.hasSigningCertificate(pkgname, hex2Bytes(correctHash), CERT_INPUT_SHA256);
        } else {
            // API Level < 28 の場合は PkgCert の機能を利用する
            return PkgCert.test(ctx, pkgname, correctHash);
```

(continues on next page)

(continued from previous page)

```
        }
    }

    private byte[] hex2Bytes(String s) {
        int len = s.length();
        byte[] data = new byte[len / 2];
        for (int i = 0; i < len; i += 2) {
            data[i / 2] = (byte) ((Character.digit(s.charAt(i), 16) << 4)
                    + Character.digit(s.charAt(i+1), 16));
        }
        return data;
    }
}
```

PkgCert.java

```
package org.jssec.android.shared;

import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

import android.content.Context;
import android.content.pm.PackageInfo;
import android.content.pm.PackageManager;
import android.content.pm.PackageManager.NameNotFoundException;
import android.content.pm.Signature;

public class PkgCert {

    public static boolean test(Context ctx, String pkgname, String correctHash) {
        if (correctHash == null) return false;
        correctHash = correctHash.replaceAll(" ", "");
        return correctHash.equals(hash(ctx, pkgname));
    }

    public static String hash(Context ctx, String pkgname) {
        if (pkgname == null) return null;
        try {
            PackageManager pm = ctx.getPackageManager();
            PackageInfo pkginfo = pm.getPackageInfo(pkgname, PackageManager.GET_SIGNATURES);
            if (pkginfo.signatures.length != 1) return null;    // 複数署名は扱わない
            Signature sig = pkginfo.signatures[0];
            byte[] cert = sig.toByteArray();
            byte[] sha256 = computeSha256(cert);
            return byte2hex(sha256);
        } catch (NameNotFoundException e) {
            return null;
        }
    }

    private static byte[] computeSha256(byte[] data) {
        try {
            return MessageDigest.getInstance("SHA-256").digest(data);
        } catch (NoSuchAlgorithmException e) {
            return null;
```

(continues on next page)

(continued from previous page)

```
        }
    }

    private static String byte2hex(byte[] data) {
        if (data == null) return null;
        final StringBuilder hexadecimal = new StringBuilder();
        for (final byte b : data) {
            hexadecimal.append(String.format("%02X", b));
        }
        return hexadecimal.toString();
    }
}
```

次にパートナー限定 Activity を利用する側のサンプルコードを示す。ここではパートナー限定 Activity を呼び出す方法を説明する。

ポイント (Activity を利用する) :

7. 利用先パートナー限定 Activity アプリの証明書がホワイトリストに登録されていることを確認する
8. Activity に送信する Intent には、フラグ FLAG_ACTIVITY_NEW_TASK を設定しない
9. 利用先パートナー限定アプリに開示してよい情報は putExtra() を使う場合に限り送信してよい
10. 明示的 Intent によりパートナー限定 Activity を呼び出す
11. startActivityForResult() によりパートナー限定 Activity を呼び出す
12. パートナー限定アプリからの結果情報であっても、受信 Intent の安全性を確認する

ホワイトリストを用いたアプリの確認方法については、「4.1.3.2. 利用元アプリを確認する」を参照すること。また、ホワイトリストに指定する利用先アプリの証明書ハッシュ値の確認方法は「5.2.1.3. アプリの証明書のハッシュ値を確認する方法」を参照すること。

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="org.jssec.android.activity.partneruser" >

    <application
        android:allowBackup="false"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name" >

        <activity
            android:name=".PartnerUserActivity"
            android:label="@string/app_name"
            android:exported="true" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

PartnerUserActivity.java

```
package org.jssec.android.activity.partneruser;

import org.jssec.android.shared.PkgCertWhitelists;
import org.jssec.android.shared.Utils;

import android.app.Activity;
import android.content.ActivityNotFoundException;
import android.content.Context;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.widget.Toast;

public class PartnerUserActivity extends Activity {

    // ★ポイント 7★ 利用先パートナー限定 Activity アプリの証明書がホワイトリストに登録されていることを確認する
    private static PkgCertWhitelists sWhitelists = null;
    private static void buildWhitelists(Context context) {
        boolean isdebug = Utils.isDebuggable(context);
        sWhitelists = new PkgCertWhitelists();

        // パートナー限定 Activity アプリ org.jssec.android.activity.partneractivity の証明書ハッシュ値を登録
        sWhitelists.add("org.jssec.android.activity.partneractivity", isdebug ?
                // debug.keystore の"androiddebugkey"の証明書ハッシュ値
                "0EFB7236 328348A9 89718BAD DF57F544 D5CCB4AE B9DB34BC 1E29DD26 F77C8255" :
                // keystore の"my company key"の証明書ハッシュ値
                "D397D343 A5CBC10F 4EDDEB7C A10062DE 5690984F 1FB9E88B D7B3A7C2 42E142CA");

        // 以下同様に他のパートナー限定 Activity アプリを登録...
    }
    private static boolean checkPartner(Context context, String pkgname) {
        if (sWhitelists == null) buildWhitelists(context);
        return sWhitelists.test(context, pkgname);
    }

    private static final int REQUEST_CODE = 1;

    // 利用先のパートナー限定 Activity に関する情報
    private static final String TARGET_PACKAGE =  "org.jssec.android.activity.partneractivity";
    private static final String TARGET_ACTIVITY = "org.jssec.android.activity.partneractivity.
↪PartnerActivity";

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }

    public void onUseActivityClick(View view) {

        // ★ポイント 7★ 利用先パートナー限定 Activity アプリの証明書がホワイトリストに登録されていることを確認する
        if (!checkPartner(this, TARGET_PACKAGE)) {
            Toast.makeText(this, "利用先 Activity アプリはホワイトリストに登録されていない。", Toast.LENGTH_
↪LONG).show();
            return;
```

(continues on next page)

(continued from previous page)

```
        }

        try {
            Intent intent = new Intent();

            // ★ポイント 8★ Activity に送信する Intent には、フラグ FLAG_ACTIVITY_NEW_TASK を設定しない

            // ★ポイント 9★ 利用先パートナー限定アプリに開示してよい情報を putExtra() を使う場合に限り送信してよい
            intent.putExtra("PARAM", "パートナーアプリに開示してよい情報");

            // ★ポイント 10★ 明示的 Intent によりパートナー限定 Activity を呼び出す
            intent.setClassName(TARGET_PACKAGE, TARGET_ACTIVITY);

            // ★ポイント 11★ startActivityForResult() によりパートナー限定 Activity を呼び出す
            startActivityForResult(intent, REQUEST_CODE);
        }
        catch (ActivityNotFoundException e) {
            Toast.makeText(this, "利用先 Activity が見つからない。", Toast.LENGTH_LONG).show();
        }
    }

    @Override
    public void onActivityResult(int requestCode, int resultCode, Intent data) {
        super.onActivityResult(requestCode, resultCode, data);

        if (resultCode != RESULT_OK) return;

        switch (requestCode) {
        case REQUEST_CODE:
            String result = data.getStringExtra("RESULT");

            // ★ポイント 12★ パートナー限定アプリからの結果情報であっても、受信 Intent の安全性を確認する
            // サンプルにつき割愛。「3.2 入力データの安全性を確認する」を参照。
            Toast.makeText(this,
                    String.format("結果「%s」を受け取った。", result), Toast.LENGTH_LONG).show();
            break;
        }
    }
}
```

PkgCertWhitelists.java

```
package org.jssec.android.shared;

import android.content.pm.PackageManager;
import java.util.HashMap;
import java.util.Map;
import android.content.Context;
import android.os.Build;

import static android.content.pm.PackageManager.CERT_INPUT_SHA256;

public class PkgCertWhitelists {
    private Map<String, String> mWhitelists = new HashMap<String, String>();
```

(continues on next page)

(continued from previous page)

```
    public boolean add(String pkgname, String sha256) {
        if (pkgname == null) return false;
        if (sha256 == null) return false;

        sha256 = sha256.replaceAll(" ", "");
        if (sha256.length() != 64) return false;    // SHA-256 は 32 バイト
        sha256 = sha256.toUpperCase();
        if (sha256.replaceAll("[0-9A-F]+", "").length() != 0) return false; // 0-9A-F 以外の文字がある

        mWhitelists.put(pkgname, sha256);
        return true;
    }

    public boolean test(Context ctx, String pkgname) {
        // pkgname に対応する正解のハッシュ値を取得する
        String correctHash = mWhitelists.get(pkgname);
        android.util.Log.d("Partner", "hash=" + correctHash);
        // pkgname の実際のハッシュ値と正解のハッシュ値を比較する
        if (Build.VERSION.SDK_INT >= 28) {
            // ★ API Level >= 28 では Package Manager の API で直接検証が可能
            PackageManager pm = ctx.getPackageManager();
            return pm.hasSigningCertificate(pkgname, hex2Bytes(correctHash), CERT_INPUT_SHA256);
        } else {
            // API Level < 28 の場合は PkgCert の機能を利用する
            return PkgCert.test(ctx, pkgname, correctHash);
        }
    }

    private byte[] hex2Bytes(String s) {
        int len = s.length();
        byte[] data = new byte[len / 2];
        for (int i = 0; i < len; i += 2) {
            data[i / 2] = (byte) ((Character.digit(s.charAt(i), 16) << 4)
                    + Character.digit(s.charAt(i+1), 16));
        }
        return data;
    }
}
```

PkgCert.java

```
package org.jssec.android.shared;

import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

import android.content.Context;
import android.content.pm.PackageInfo;
import android.content.pm.PackageManager;
import android.content.pm.PackageManager.NameNotFoundException;
import android.content.pm.Signature;

public class PkgCert {

    public static boolean test(Context ctx, String pkgname, String correctHash) {
```

(continues on next page)

(continued from previous page)

```
        if (correctHash == null) return false;
        correctHash = correctHash.replaceAll(" ", "");
        return correctHash.equals(hash(ctx, pkgname));
    }

    public static String hash(Context ctx, String pkgname) {
        if (pkgname == null) return null;
        try {
            PackageManager pm = ctx.getPackageManager();
            PackageInfo pkginfo = pm.getPackageInfo(pkgname, PackageManager.GET_SIGNATURES);
            if (pkginfo.signatures.length != 1) return null;    // 複数署名は扱わない
            Signature sig = pkginfo.signatures[0];
            byte[] cert = sig.toByteArray();
            byte[] sha256 = computeSha256(cert);
            return byte2hex(sha256);
        } catch (NameNotFoundException e) {
            return null;
        }
    }

    private static byte[] computeSha256(byte[] data) {
        try {
            return MessageDigest.getInstance("SHA-256").digest(data);
        } catch (NoSuchAlgorithmException e) {
            return null;
        }
    }

    private static String byte2hex(byte[] data) {
        if (data == null) return null;
        final StringBuilder hexadecimal = new StringBuilder();
        for (final byte b : data) {
            hexadecimal.append(String.format("%02X", b));
        }
        return hexadecimal.toString();
    }
}
```

4.1.1.4 自社限定 **Activity** を作る・利用する

自社限定 Activity は、自社以外のアプリから利用されることを禁止する Activity である。複数の自社製アプリでシステムを構成し、自社アプリが扱う情報や機能を守るために利用される。

Activity を呼び出す際に使用する Intent は第三者によって読み取られる恐れがある。そのため、Activity に送信する Intent にセンシティブな情報を格納する場合には、その情報が悪意のある第三者に読み取られることのないように、適切な対応を実施する必要がある。

以下に自社限定 Activity を作る側のサンプルコードを示す。

ポイント (Activity を作る) :

1. 独自定義 Signature Permission を定義する
2. taskAffinity を指定しない

3. launchMode を指定しない
4. 独自定義 Signature Permission を要求宣言する
5. Intent Filter を定義せず、exported=”true” を明示的に設定する
6. 独自定義 Signature Permission が自社アプリにより定義されていることを確認する
7. 自社アプリからの Intent であっても、受信 Intent の安全性を確認する
8. 利用元アプリは自社アプリであるから、センシティブな情報を返送してよい
9. 利用元アプリと同じ開発者鍵で APK を署名する

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="org.jssec.android.activity.inhouseactivity" >

    <!-- ★ポイント 1★ 独自定義 Signature Permission を定義する -->
    <permission
        android:name="org.jssec.android.activity.inhouseactivity.MY_PERMISSION"
        android:protectionLevel="signature" />

    <application
        android:allowBackup="false"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name" >

        <!-- 自社限定 Activity -->
        <!-- ★ポイント 2★ taskAffinity を指定しない -->
        <!-- ★ポイント 3★ launchMode を指定しない -->
        <!-- ★ポイント 4★ 独自定義 Signature Permission を要求宣言する -->
        <!-- ★ポイント 5★ Intent Filter を定義せず、exported="true"を明示的に設定する -->
        <activity
            android:name=".InhouseActivity"
            android:exported="true"
            android:permission="org.jssec.android.activity.inhouseactivity.MY_PERMISSION" >
        </activity>
    </application>

</manifest>
```

InhouseActivity.java

```
package org.jssec.android.activity.inhouseactivity;

import org.jssec.android.shared.SigPerm;
import org.jssec.android.shared.Utils;

import android.app.Activity;
import android.content.Context;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.widget.Toast;

public class InhouseActivity extends Activity {
```

(continues on next page)

(continued from previous page)

```
    // 自社の Signature Permission
    private static final String MY_PERMISSION = "org.jssec.android.activity.inhouseactivity.MY_PERMISSION
↪";

    // 自社の証明書のハッシュ値
    private static String sMyCertHash = null;
    private static String myCertHash(Context context) {
        if (sMyCertHash == null) {
            if (Utils.isDebuggable(context)) {
                // debug.keystore の"androiddebugkey"の証明書ハッシュ値
                sMyCertHash = "0EFB7236 328348A9 89718BAD DF57F544 D5CCB4AE B9DB34BC 1E29DD26 F77C8255";
            } else {
                // keystore の"my company key"の証明書ハッシュ値
                sMyCertHash = "D397D343 A5CBC10F 4EDDEB7C A10062DE 5690984F 1FB9E88B D7B3A7C2 42E142CA";
            }
        }
        return sMyCertHash;
    }

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        // ★ポイント 6★ 独自定義 Signature Permission が自社アプリにより定義されていることを確認する
        if (!SigPerm.test(this, MY_PERMISSION, myCertHash(this))) {
            Toast.makeText(this, "独自定義 Signature Permission が自社アプリにより定義されていない。", Toast.
↪LENGTH_LONG).show();
            finish();
            return;
        }

        // ★ポイント 7★ 自社アプリからの Intent であっても、受信 Intent の安全性を確認する
        // サンプルにつき割愛。「3.2 入力データの安全性を確認する」を参照
        String param = getIntent().getStringExtra("PARAM");
        Toast.makeText(this, String.format("パラメータ「%s」を受け取った。", param), Toast.LENGTH_LONG).
↪show();
    }

    public void onReturnResultClick(View view) {

        // ★ポイント 8★ 利用元アプリは自社アプリであるから、センシティブな情報を返送してよい
        Intent intent = new Intent();
        intent.putExtra("RESULT", "センシティブな情報");
        setResult(RESULT_OK, intent);
        finish();
    }
}
```

SigPerm.java

```
package org.jssec.android.shared;

import android.content.Context;
```

(continues on next page)

(continued from previous page)

```
import android.content.pm.PackageManager;
import android.content.pm.PackageManager.NameNotFoundException;
import android.content.pm.PermissionInfo;
import android.os.Build;

import static android.content.pm.PackageManager.CERT_INPUT_SHA256;

public class SigPerm {

    public static boolean test(Context ctx, String sigPermName, String correctHash) {
        if (correctHash == null) return false;
        correctHash = correctHash.replaceAll(" ", "");
        try{
            // sigPermName を定義したアプリのパッケージ名を取得する
            PackageManager pm = ctx.getPackageManager();
            PermissionInfo pi = pm.getPermissionInfo(sigPermName, PackageManager.GET_META_DATA);
            String pkgname = pi.packageName;
            // 非 Signature Permission の場合は失敗扱い
            if (pi.protectionLevel != PermissionInfo.PROTECTION_SIGNATURE) return false;
            // pkgname の実際のハッシュ値と正解のハッシュ値を比較する
            if (Build.VERSION.SDK_INT >= 28) {
                // ★ API Level >= 28 では Package Manager の API で直接検証が可能
                return pm.hasSigningCertificate(pkgname, Utils.hex2Bytes(correctHash), CERT_INPUT_
↪SHA256);
            } else {
                // API Level < 28 の場合は PkgCert を利用し、ハッシュ値を取得して比較する
                return correctHash.equals(PkgCert.hash(ctx, pkgname));
            }
        } catch (NameNotFoundException e){
            return false;
        }
    }
}
```

PkgCert.java

```
package org.jssec.android.shared;

import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

import android.content.Context;
import android.content.pm.PackageInfo;
import android.content.pm.PackageManager;
import android.content.pm.PackageManager.NameNotFoundException;
import android.content.pm.Signature;

public class PkgCert {

    public static boolean test(Context ctx, String pkgname, String correctHash) {
        if (correctHash == null) return false;
        correctHash = correctHash.replaceAll(" ", "");
        return correctHash.equals(hash(ctx, pkgname));
    }
```

(continues on next page)

(continued from previous page)

```
    public static String hash(Context ctx, String pkgname) {
        if (pkgname == null) return null;
        try {
            PackageManager pm = ctx.getPackageManager();
            PackageInfo pkginfo = pm.getPackageInfo(pkgname, PackageManager.GET_SIGNATURES);
            if (pkginfo.signatures.length != 1) return null;    // 複数署名は扱わない
            Signature sig = pkginfo.signatures[0];
            byte[] cert = sig.toByteArray();
            byte[] sha256 = computeSha256(cert);
            return byte2hex(sha256);
        } catch (NameNotFoundException e) {
            return null;
        }
    }

    private static byte[] computeSha256(byte[] data) {
        try {
            return MessageDigest.getInstance("SHA-256").digest(data);
        } catch (NoSuchAlgorithmException e) {
            return null;
        }
    }

    private static String byte2hex(byte[] data) {
        if (data == null) return null;
        final StringBuilder hexadecimal = new StringBuilder();
        for (final byte b : data) {
            hexadecimal.append(String.format("%02X", b));
        }
        return hexadecimal.toString();
    }
}
```

★ポイント 9 ★ APK を Export するときに、利用元アプリと同じ開発者鍵で APK を署名する。

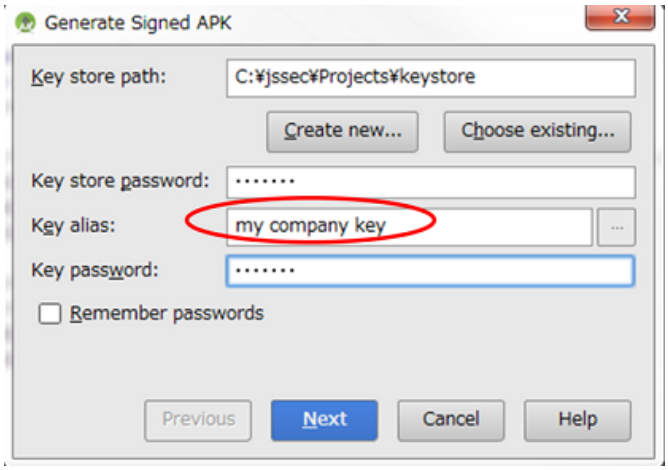

図 4.1.2　利用元アプリと同じ開発者鍵で APK を署名する

次に自社限定 Activity を利用する側のサンプルコードを示す。ここでは自社限定 Activity を呼び出す方法を説明する。

ポイント (Activity を利用する) :

10. 独自定義 Signature Permission を利用宣言する
11. 独自定義 Signature Permission が自社アプリにより定義されていることを確認する
12. 利用先アプリの証明書が自社の証明書であることを確認する
13. 利用先アプリは自社アプリであるから、センシティブな情報を putExtra() を使う場合に限り送信してもよい
14. 明示的 Intent により自社限定 Activity を呼び出す
15. 自社アプリからの結果情報であっても、受信 Intent の安全性を確認する
16. 利用先アプリと同じ開発者鍵で APK を署名する

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="org.jssec.android.activity.inhouseuser" >

    <!-- ★ポイント 10★ 独自定義 Signature Permission を利用宣言する -->
    <uses-permission
        android:name="org.jssec.android.activity.inhouseactivity.MY_PERMISSION" />

    <application
        android:allowBackup="false"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name" >

        <activity
            android:name="org.jssec.android.activity.inhouseuser.InhouseUserActivity"
            android:label="@string/app_name"
            android:exported="true" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

InhouseUserActivity.java

```
package org.jssec.android.activity.inhouseuser;

import org.jssec.android.shared.PkgCert;
import org.jssec.android.shared.SigPerm;
import org.jssec.android.shared.Utils;

import android.app.Activity;
import android.content.ActivityNotFoundException;
import android.content.Context;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.widget.Toast;

public class InhouseUserActivity extends Activity {
```

(continues on next page)

(continued from previous page)

```
    // 利用先の Activity 情報
    private static final String TARGET_PACKAGE =  "org.jssec.android.activity.inhouseactivity";
    private static final String TARGET_ACTIVITY = "org.jssec.android.activity.inhouseactivity.
↪InhouseActivity";

    // 自社の Signature Permission
    private static final String MY_PERMISSION = "org.jssec.android.activity.inhouseactivity.MY_PERMISSION
↪";

    // 自社の証明書のハッシュ値
    private static String sMyCertHash = null;
    private static String myCertHash(Context context) {
        if (sMyCertHash == null) {
            if (Utils.isDebuggable(context)) {
                // debug.keystore の "androiddebugkey" の証明書ハッシュ値
                sMyCertHash = "0EFB7236 328348A9 89718BAD DF57F544 D5CCB4AE B9DB34BC 1E29DD26 F77C8255";
            } else {
                // keystore の "my company key" の証明書ハッシュ値
                sMyCertHash = "D397D343 A5CBC10F 4EDDEB7C A10062DE 5690984F 1FB9E88B D7B3A7C2 42E142CA";
            }
        }
        return sMyCertHash;
    }

    private static final int REQUEST_CODE = 1;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }

    public void onUseActivityClick(View view) {

        // ★ポイント 11★ 独自定義 Signature Permission が自社アプリにより定義されていることを確認する
        if (!SigPerm.test(this, MY_PERMISSION, myCertHash(this))) {
            Toast.makeText(this, "独自定義 Signature Permission が自社アプリにより定義されていない。", Toast.
↪LENGTH_LONG).show();
            return;
        }

        // ★ポイント 12★ 利用先アプリの証明書が自社の証明書であることを確認する
        if (!PkgCert.test(this, TARGET_PACKAGE, myCertHash(this))) {
            Toast.makeText(this, "利用先アプリは自社アプリではない。", Toast.LENGTH_LONG).show();
            return;
        }

        try {
            Intent intent = new Intent();

            // ★ポイント 13★ 利用先アプリは自社アプリであるから、センシティブな情報を putExtra() を使う場合に限り
送信してもよい
            intent.putExtra("PARAM", "センシティブな情報");
```

(continues on next page)

(continued from previous page)

```
            // ★ポイント 14★ 明示的 Intent により自社限定 Activity を呼び出す
            intent.setClassName(TARGET_PACKAGE, TARGET_ACTIVITY);
            startActivityForResult(intent, REQUEST_CODE);
        }
        catch (ActivityNotFoundException e) {
            Toast.makeText(this, "利用先 Activity が見つからない。", Toast.LENGTH_LONG).show();
        }

    }

    @Override
    public void onActivityResult(int requestCode, int resultCode, Intent data) {

        super.onActivityResult(requestCode, resultCode, data);
        if (resultCode != RESULT_OK) return;
        switch (requestCode) {
        case REQUEST_CODE:
            String result = data.getStringExtra("RESULT");


            // ★ポイント 15★ 自社アプリからの結果情報であっても、受信 Intent の安全性を確認する
            // サンプルにつき割愛。「3.2 入力データの安全性を確認する」を参照。
            Toast.makeText(this, String.format("結果「%s」を受け取った。", result), Toast.LENGTH_LONG).
↪show();
            break;
        }
    }
}
```

SigPerm.java

```
package org.jssec.android.shared;

import android.content.Context;
import android.content.pm.PackageManager;
import android.content.pm.PackageManager.NameNotFoundException;
import android.content.pm.PermissionInfo;
import android.os.Build;

import static android.content.pm.PackageManager.CERT_INPUT_SHA256;

public class SigPerm {

    public static boolean test(Context ctx, String sigPermName, String correctHash) {
        if (correctHash == null) return false;
        correctHash = correctHash.replaceAll(" ", "");
        try{
            // sigPermName を定義したアプリのパッケージ名を取得する
            PackageManager pm = ctx.getPackageManager();
            PermissionInfo pi = pm.getPermissionInfo(sigPermName, PackageManager.GET_META_DATA);
            String pkgname = pi.packageName;
            // 非 Signature Permission の場合は失敗扱い
            if (pi.protectionLevel != PermissionInfo.PROTECTION_SIGNATURE) return false;
            // pkgname の実際のハッシュ値と正解のハッシュ値を比較する
            if (Build.VERSION.SDK_INT >= 28) {
```

(continues on next page)

(continued from previous page)

```
                // ★ API Level >= 28 では Package Manager の API で直接検証が可能
                return pm.hasSigningCertificate(pkgname, Utils.hex2Bytes(correctHash), CERT_INPUT_
↪SHA256);
            } else {
                // API Level < 28 の場合は PkgCert を利用し、ハッシュ値を取得して比較する
                return correctHash.equals(PkgCert.hash(ctx, pkgname));
            }
        } catch (NameNotFoundException e){
            return false;
        }
    }
}
```

PkgCert.java

```
package org.jssec.android.shared;

import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

import android.content.Context;
import android.content.pm.PackageInfo;
import android.content.pm.PackageManager;
import android.content.pm.PackageManager.NameNotFoundException;
import android.content.pm.Signature;

public class PkgCert {

    public static boolean test(Context ctx, String pkgname, String correctHash) {
        if (correctHash == null) return false;
        correctHash = correctHash.replaceAll(" ", "");
        return correctHash.equals(hash(ctx, pkgname));
    }

    public static String hash(Context ctx, String pkgname) {
        if (pkgname == null) return null;
        try {
            PackageManager pm = ctx.getPackageManager();
            PackageInfo pkginfo = pm.getPackageInfo(pkgname, PackageManager.GET_SIGNATURES);
            if (pkginfo.signatures.length != 1) return null;    // 複数署名は扱わない
            Signature sig = pkginfo.signatures[0];
            byte[] cert = sig.toByteArray();
            byte[] sha256 = computeSha256(cert);
            return byte2hex(sha256);
        } catch (NameNotFoundException e) {
            return null;
        }
    }

    private static byte[] computeSha256(byte[] data) {
        try {
            return MessageDigest.getInstance("SHA-256").digest(data);
        } catch (NoSuchAlgorithmException e) {
            return null;
        }
```

(continues on next page)

(continued from previous page)

```
    }

    private static String byte2hex(byte[] data) {
        if (data == null) return null;
        final StringBuilder hexadecimal = new StringBuilder();
        for (final byte b : data) {
            hexadecimal.append(String.format("%02X", b));
        }
        return hexadecimal.toString();
    }
}
```

★ポイント 16 ★ APK を Export するときに、利用先アプリと同じ開発者鍵で APK を署名する。

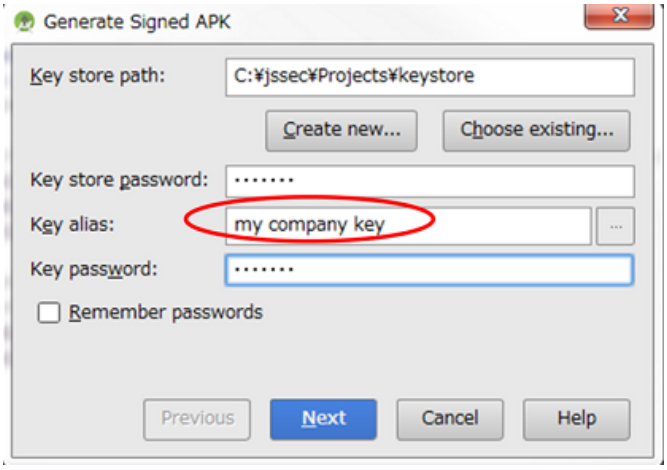

図 4.1.3　利用先アプリと同じ開発者鍵で APK を署名する

4.1.2 ルールブック

Activity を作る際、または Activity に Intent を送信する際には以下のルールを守ること。

1. アプリ内でのみ使用する *Activity* は非公開設定する （必須）
2. *taskAffinity* を指定しない （必須）
3. *launchMode* を指定しない （必須）
4. *Activity* に送信する *Intent* には *FLAG_ACTIVITY_NEW_TASK* を設定しない （必須）
5. 受信 *Intent* の安全性を確認する （必須）
6. 独自定義 *Signature Permission* は、自社アプリが定義したことを確認して利用する （必須）
7. 結果情報を返す場合には、返送先アプリからの結果情報漏洩に注意する （必須）
8. 利用先 *Activity* が固定できる場合は明示的 *Intent* で *Activity* を利用する （必須）
9. 利用先 *Activity* からの戻り *Intent* の安全性を確認する （必須）
10. 他社の特定アプリと連携する場合は利用先 *Activity* を確認する （必須）
11. 資産を二次的に提供する場合には、その資産の従来の保護水準を維持する （必須）

12. センシティブな情報はできる限り送らない （推奨）

4.1.2.1 アプリ内でのみ使用する **Activity** は非公開設定する （必須）

同一アプリ内からのみ利用される Activity は他のアプリから Intent を受け取る必要がない。またこのような Activity では開発者も Activity を攻撃する Intent を想定しないことが多い。このような Activity は明示的に非公開設定し、非公開 Activity とする。

```
AndroidManifest.xml
    <!-- 非公開 Activity -->
    <!-- ★ポイント 3★ exported="false"により、明示的に非公開設定する -->
    <activity
        android:name=".PrivateActivity"
        android:label="@string/app_name"
        android:exported="false" />
```

同一アプリ内からのみ利用される Activity では Intent Filter を設置するような設計はしてはならない。Intent Filter の性質上、同一アプリ内の非公開 Activity を呼び出すつもりでも、Intent Filter 経由で呼び出したときに意図せず他アプリの Activity を呼び出してしまう可能性もある。詳細は、アドバンスト「4.1.3.1. *exported* 設定と *intent-filter* 設定の組み合わせ *(Activity* の場合*)*」を参照すること。

```
AndroidManifest.xml(非推奨)
    <!-- 非公開 Activity -->
    <!-- ★ポイント 3★ exported="false"により、明示的に非公開設定する -->
    <activity
        android:name=".PictureActivity"
        android:label="@string/picture_name"
        android:exported="false" >
        <intent-filter>
            <action android:name=” org.jssec.android.activity.OPEN />
        </intent-filter>
    </activity>
```

4.1.2.2 **taskAffinity** を指定しない （必須）

Android では、Activity はタスクによって管理される。タスクの名前は、ルート Activity の持つアフィニティによって決定される。一方でルート以外の Activity に関しては、所属するタスクがアフィニティだけでは決定されず、Activity の起動モードにも依存する。詳細は「4.1.3.4. ルート *Activity* について」を参照すること。

デフォルト設定では各 Activity はパッケージ名をアフィニティとして持つ。その結果、タスクはアプリごとに割り当てられるので、同一アプリ内の全ての Activity は同一タスクに所属する。タスクの割り当てを変更するには、AndroidManifest.xml への明示的なアフィニティ記述や、Activity に送信する Intent へのフラグ設定をすればよい。ただし、タスクの割り当てを変更した場合は、異なるタスクに属する Activity に送信した Intent を別アプリによって読み出せる可能性がある。

センシティブな情報を含む送信 Intent および受信 Intent の内容を読み取られないようにするには、AndroidManifest.xml 内の application 要素および activity 要素で android:taskAffinity を指定せず、デフォルト (パッケージ名と同一) のままにすべきである。

以下に非公開 Activity の作成側と利用側における AndroidManifest.xml を示す。

```
AndroidManifest.xml
    <!-- ★ポイント 1★ taskAffinity を指定しない -->
    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name" >

        <!-- ★ポイント 1★ taskAffinity を指定しない -->
        <activity
            android:name=".PrivateUserActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

        <!-- 非公開 Activity -->
        <!-- ★ポイント 1★ taskAffinity を指定しない -->
        <activity
            android:name=".PrivateActivity"
            android:label="@string/app_name"
            android:exported="false" />
    </application>
```

タスクとアフィニティの詳細な解説は、「Google Android プログラミング入門」[*2]、あるいは、Google Developers API Guide ”Tasks and Back Stack”[*3] の解説および「4.1.3.3. *Activity* に送信される *Intent* の読み取り *(Android 5.0* より前のバージョンについて*)*」、「4.1.3.4. ルート *Activity* について」を参照すること。

4.1.2.3 launchMode を指定しない （必須）

Activity の起動モードとは、Activity を呼び出す際に、Activity のインスタンスの新規生成や、タスクの新規生成を制御するための設定である。デフォルト設定は”standard” である。”standard” 設定では、Intent を使って Activity を呼び出すときには常に新規インスタンスを生成し、タスクは呼び出し側 Activity が属するタスクに従い、新規にタスクが生成されることはない。タスクが新規に生成されると、呼び出しに使った Intent が別のアプリから読み取り可能になる。そのため、センシティブな情報を Intent に含む場合には、Activity の起動モードには”standard” を用いるべきである。

Activity の起動モードは AndroidManifest.xml 内にて android:launchMode で明示的に設定可能であるが、上記の理由により、各 Activity に対して android:launchMode を指定せず、値をデフォルトのまま”standard” とするべきである。

```
AndroidManifest.xml
        <!-- ★ポイント 2★ Activity には launchMode を指定せず、値をデフォルトのまま” standard” とする -->
        <activity
            android:name=".PrivateUserActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
```

(continues on next page)

[*2] 江川、藤井、麻野、藤田、山田、山岡、佐野、竹端著「Google Android プログラミング入門」(アスキー・メディアワークス、2009 年 7 月)
[*3] https://developer.android.com/guide/components/tasks-and-back-stack.html

(continued from previous page)

```
        <!-- 非公開 Activity -->
        <!-- ★ポイント 2★ Activity には launchMode を指定せず、値をデフォルトのまま”standard”とする -->
        <activity
            android:name=".PrivateActivity"
            android:label="@string/app_name"
            android:exported="false" />
    </application>
```

「4.1.3.3. *Activity* に送信される *Intent* の読み取り *(Android 5.0* より前のバージョンについて*)*」、「4.1.3.4. ルート *Activity* について」を参照すること。

4.1.2.4 Activity に送信する Intent には FLAG_ACTIVITY_NEW_TASK を設定しない （必須）

Activity の起動モードは startActivity() あるいは startActivityForResult() の実行時にも変更することが可能であり、タスクが新規に生成される場合がある。そのため、Activity の起動モードを実行時に変更しないようにする必要がある。

Activity の起動モードを変更するには、setFlags() や addFlags() を用いて Intent にフラグを設定し、その Intent を startActivity() または startActivityForResult() の引数とする。タスクを新規に生成するためのフラグは FLAG_ACTIVITY_NEW_TASK である。FLAG_ACTIVITY_NEW_TASK が設定されると、呼び出された Activity のタスクがバックグラウンドあるいはフォアグラウンド上に存在しない場合に、新規にタスクが生成される。FLAG_ACTIVITY_MULTIPLE_TASK は FLAG_ACTIVITY_NEW_TASK と同時に設定することもできる。この場合には、タスクが必ず新規生成される。どちらの設定もタスクを生成する可能性があるため、センシティブな情報を扱う Intent には設定しないようにすべきである。

Intent の送信例

```
        Intent intent = new Intent();

        // ★ポイント 6★ Activity に送信する Intent には、フラグ FLAG_ACTIVITY_NEW_TASK を設定しない

        intent.setClass(this, PrivateActivity.class);
        intent.putExtra("PARAM", "センシティブな情報");

        startActivityForResult(intent, REQUEST_CODE);
```

なお、Activity に送信する Intent に FLAG_ACTIVITY_EXCLUDE_FROM_RECENTS フラグを明示的に設定することで、タスクが生成されたとしてもその内容が読み取られないようにできると考えるかもしれない。しかしながら、この方法を用いても送信された Intent の内容を読み取ることが可能である。したがって、FLAG_ACTIVITY_NEW_TASK の使用は避けるべきである。

「4.1.3.1. *exported* 設定と *intent-filter* 設定の組み合わせ *(Activity* の場合*)*」および「4.1.3.3. *Activity* に送信される *Intent* の読み取り *(Android 5.0* より前のバージョンについて*)*」、「4.1.3.4. ルート *Activity* について」も参照すること。

4.1.2.5 受信 Intent の安全性を確認する （必須）

Activity のタイプによって若干リスクは異なるが、受信 Intent のデータを処理する際には、まず受信 Intent の安全性を確認しなければならない。

公開 Activity は不特定多数のアプリから Intent を受け取るため、マルウェアの攻撃 Intent を受け取る可能性がある。非公開 Activity は他のアプリから Intent を直接受け取ることはない。しかし同一アプリ内の公開 Activity が他のアプリか

ら受け取った Intent のデータを非公開 Activity に転送することがあるため、受信 Intent を無条件に安全であると考えてはならない。パートナー限定 Activity や自社限定 Activity はその中間のリスクであるため、やはり受信 Intent の安全性を確認する必要がある。

「3.2. 入力データの安全性を確認する」を参照すること。

4.1.2.6 独自定義 Signature Permission は、自社アプリが定義したことを確認して利用する （必須）

自社アプリだけから利用できる自社限定 Activity を作る場合、独自定義 Signature Permission により保護しなければならない。AndroidManifest.xml での Permission 定義、Permission 要求宣言だけでは保護が不十分であるため、「5.2. *Permission* と *Protection Level*」の「5.2.1.2. 独自定義の *Signature Permission* で自社アプリ連携する方法」を参照すること。

4.1.2.7 結果情報を返す場合には、返送先アプリからの結果情報漏洩に注意する （必須）

Activity のタイプによって、setResult() を用いて結果情報を返送する際の返送先アプリの信用度が異なる。公開 Activity が結果情報を返送する場合、結果返送先アプリがマルウェアである可能性があり、結果情報が悪意を持って使われる危険性がある。非公開 Activity や自社限定 Activity の場合は、結果返送先は自社アプリであるため結果情報の扱いをあまり心配する必要はない。パートナー限定 Activity の場合はその中間に位置する。

このように Activity から結果情報を返す場合には、返送先アプリからの結果情報の漏洩に配慮しなければならない。

結果情報を返送する場合の例

```
public void onReturnResultClick(View view) {

    // ★ポイント 6★ パートナーアプリに開示してよい情報に限り返送してよい
    Intent intent = new Intent();
    intent.putExtra("RESULT", "パートナーアプリに開示してよい情報");
    setResult(RESULT_OK, intent);
    finish();
}
```

4.1.2.8 利用先 Activity が固定できる場合は明示的 Intent で Activity を利用する （必須）

暗黙的 Intent により Activity を利用すると、最終的にどの Activity に Intent が送信されるかは Android OS 任せになってしまう。もしマルウェアに Intent が送信されてしまうと情報漏洩が生じる。一方、明示的 Intent により Activity を利用すると、指定した Activity 以外が Intent を受信することはなく比較的安全である。

処理を任せるアプリ（の Activity）をユーザーに選択させるなど、利用先 Activity を実行時に決定したい場合を除けば、利用先 Activity はあらかじめ特定できる。このような Activity を利用する場合には明示的 Intent を利用すべきである。

同一アプリ内の Activity を明示的 Intent で利用する

```
    Intent intent = new Intent(this, PictureActivity.class);
    intent.putExtra("BARCODE", barcode);
    startActivity(intent);
```

他のアプリの公開 Activity を明示的 Intent で利用する

```
        Intent intent = new Intent();
        intent.setClassName(
            "org.jssec.android.activity.publicactivity",
            "org.jssec.android.activity.publicactivity.PublicActivity");
        startActivity(intent);
```

ただし他のアプリの公開 Activity を明示的 Intent で利用した場合も、相手先 Activity を含むアプリがマルウェアである可能性がある。宛先をパッケージ名で限定したとしても、相手先アプリが実は本物アプリと同じパッケージ名を持つ偽物アプリである可能性があるからだ。このようなリスクを排除したい場合は、パートナー限定 Activity や自社限定 Activity の使用を検討する必要がある。

「4.1.3.1. *exported* 設定と *intent-filter* 設定の組み合わせ *(Activity* の場合*)*」も参照すること。

4.1.2.9 利用先 **Activity** からの戻り **Intent** の安全性を確認する （必須）

Activity のタイプによって若干リスクは異なるが、戻り値として受信した Intent のデータを処理する際には、まず受信 Intent の安全性を確認しなければならない。

利用先 Activity が公開 Activity の場合、不特定のアプリから戻り Intent を受け取るため、マルウェアの攻撃 Intent を受け取る可能性がある。利用先 Activity が非公開 Activity の場合、同一アプリ内から戻り Intent を受け取るのでリスクはないように考えがちだが、他のアプリから受け取った Intent のデータを間接的に戻り値として転送することがあるため、受信 Intent を無条件に安全であると考えてはならない。利用先 Activity がパートナー限定 Activity や自社限定 Activity の場合、その中間のリスクであるため、やはり受信 Intent の安全性を確認する必要がある。

「3.2. 入力データの安全性を確認する」を参照すること。

4.1.2.10 他社の特定アプリと連携する場合は利用先 **Activity** を確認する （必須）

他社の特定アプリと連携する場合にはホワイトリストによる確認方法がある。自アプリ内に利用先アプリの証明書ハッシュを予め保持しておく。利用先の証明書ハッシュと保持している証明書ハッシュが一致するかを確認することで、なりすましアプリに Intent を発行することを防ぐことができる。具体的な実装方法についてはサンプルコードセクション「4.1.1.3. パートナー限定 *Activity* を作る・利用する」を参照すること。また、技術的な詳細に関しては「4.1.3.2. 利用元アプリを確認する」を参照すること。

4.1.2.11 資産を二次的に提供する場合には、その資産の従来の保護水準を維持する （必須）

Permission により保護されている情報資産および機能資産を他のアプリに二次的に提供する場合には、提供先アプリに対して同一の Permission を要求するなどして、その保護水準を維持しなければならない。Android の Permission セキュリティモデルでは、保護された資産に対するアプリからの直接アクセスについてのみ権限管理を行う。この仕様上の特性により、アプリに取得された資産がさらに他のアプリに、保護のために必要な Permission を要求することなく提供される可能性がある。このことは Permission を再委譲 (Redelegation) していることと実質的に等価なので、Permission の再委譲問題と呼ばれる。「5.2.3.4. *Permission* の再委譲問題」を参照すること。

4.1.2.12 センシティブな情報はできる限り送らない （推奨）

不特定多数のアプリと連携する場合にはセンシティブな情報を送ってはならない。特定のアプリと連携する場合においても、意図しないアプリに Intent を発行してしまった場合や第三者による Intent の盗聴などで情報が漏洩してしまうリスクがある。「4.1.3.5. *Activity* 利用時のログ出力について」を参照すること。

センシティブな情報を Activity に送付する場合、その情報の漏洩リスクを検討しなければならない。公開 Activity に送付した情報は必ず漏洩すると考えなければならない。またパートナー限定 Activity や自社限定 Activity に送付した情報もそれら Activity の実装に依存して情報漏洩リスクの大小がある。非公開 Activity に送付する情報に至っても、Intent の data に含めた情報は LogCat 経由で漏洩するリスクがある。Intent の extras は LogCat に出力されないので、センシティブな情報は extras で送付するとよい。

センシティブな情報はできるだけ送付しないように工夫すべきである。送付する場合も、利用先 Activity は信頼できる Activity に限定し、Intent の情報が LogCat へ漏洩しないように配慮しなければならない。

また、ルート Activity にはセンシティブな情報を送ってはならない。ルート Activity とは、タスクが生成された時に最初に呼び出された Activity のことである。例えば、ランチャーから起動された Activity は常にルート Activity である。

ルート Activity に関しての詳細は、「4.1.3.3. *Activity* に送信される *Intent* の読み取り *(Android 5.0* より前のバージョンについて*)*」、「4.1.3.4. ルート *Activity* について」も参照すること。

4.1.3 アドバンスト

4.1.3.1 exported 設定と intent-filter 設定の組み合わせ (Activity の場合)

このガイド文書では、Activity の用途から非公開 Activity、公開 Activity、パートナー限定 Activity、自社限定 Activity の 4 タイプの Activity について実装方法を述べている。各タイプに許されている AndroidManifest.xml の exported 属性と intent-filter 要素の組み合わせを次の表にまとめた。作ろうとしている Activity のタイプと exported 属性および intent-filter 要素の対応が正しいことを確認すること。

表 4.1.1　exported 属性と intent-filter 要素の組み合わせ

	exported 属性の値		
	true	**false**	無指定
intent-filter 定義がある	公開	（使用禁止）	（使用禁止）
intent-filter 定義がない	公開、パートナー限定、自社限定	非公開	（使用禁止）

Activity の exported 属性が無指定である場合にその Activity が公開されるか非公開となるかは、intent-filter の定義の有無により決まるが[*4]、本ガイドでは Activity の exported 属性を「無指定」にすることを禁止している。前述のような API のデフォルトの挙動に頼る実装をすることは避けるべきであり、exported 属性のようなセキュリティ上重要な設定を明示的に有効化する手段があるのであればそれを利用すべきであると考えられるためである。

exported 属性の値で「intent-filter 定義がある」&「exported=”false”」を使用禁止にしているのは、Android の振る舞いに抜け穴があり、Intent Filter の性質上、意図せず他アプリの Activity を呼び出してしまう場合が存在するためである。以下の 2 つの図は、その説明のためのものである。図 4.1.4 は、同一アプリ内からしか非公開 Activity(アプリ A）を暗黙的 Intent で呼び出せない正常な動作の例である。Intent-filter(図中 action=”X”) を定義しているのが、アプリ A しかいないので意図通りの動きとなっている。

[*4] intent-filter が定義されていれば公開 Activity、定義されていなければ非公開 Activity となる。https://developer.android.com/guide/topics/manifest/activity-element.html#exported を参照のこと。

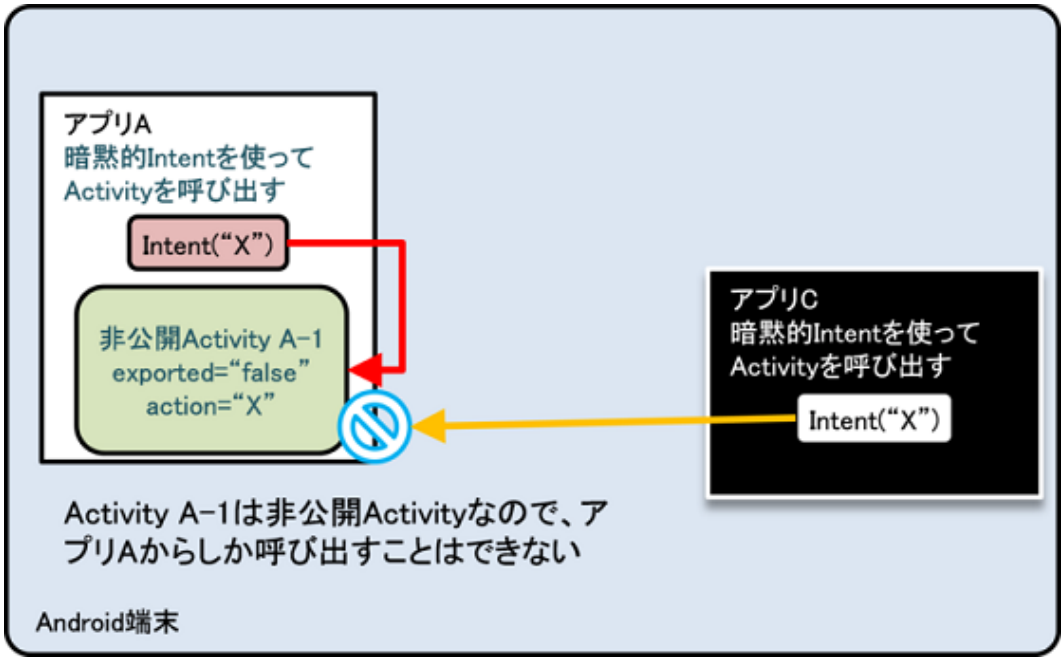

図 4.1.4　正常な動作の例

図 4.1.5 は、アプリ A に加えてアプリ B でも同じ intent-filter(図中 action=”X”) を定義している場合である。図 4.1.5 では、アプリ A が暗黙的 Intent を送信して同一アプリ内の非公開 Activity を呼び出そうとするが、「アプリケーションの選択」ダイアログが表示され、ユーザーの選択によって公開 Activity(B-1) が呼び出されてしまう例を示している。これにより他アプリに対してセンシティブな情報を送信したり、意図せぬ戻り値を受け取る可能性が生じてしまう。

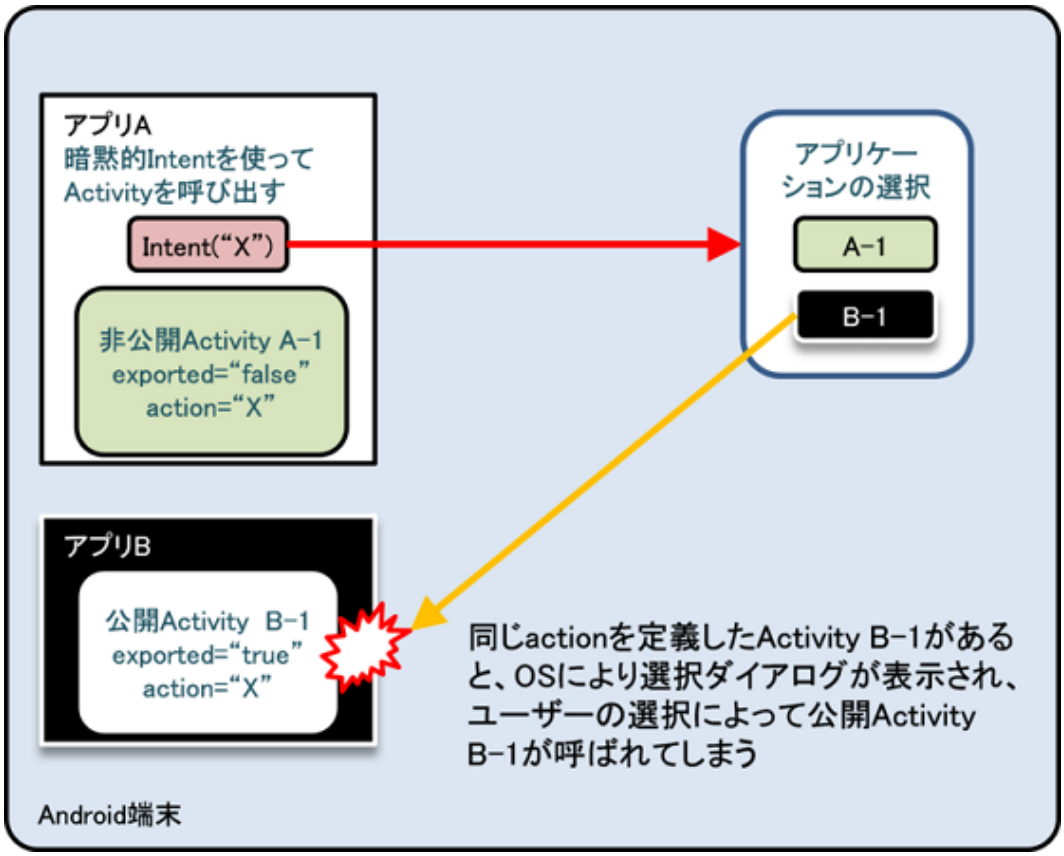

図 4.1.5　意図しない動作の例

このように、Intent Filter を用いた非公開 Activity の暗黙的 Intent 呼び出しは、意図せぬアプリとの情報のやり取りを許

してしまうので行うべきではない。なお、この挙動はアプリ A、アプリ B のインストール順序には依存しないことを確認している。

4.1.3.2 利用元アプリを確認する

ここではパートナー限定 Activity の実装に関する技術情報を解説する。パートナー限定 Activity はホワイトリストに登録された特定のアプリからのアクセスを許可し、それ以外のアプリからはアクセスを拒否する Activity である。自社以外のアプリもアクセス許可対象となるため、Signature Permission による防御手法は利用できない。

基本的な考え方は、パートナー限定 Activity の利用元アプリの身元を確認し、ホワイトリストに登録されたアプリであればサービスを提供する、登録されていないアプリであればサービスを提供しないというものである。利用元アプリの身元確認は、利用元アプリが持つ証明書を取得し、その証明書のハッシュ値をホワイトリストのハッシュ値と比較することで行う。

ここでわざわざ利用元アプリの「証明書」を取得せずとも、利用元アプリの「パッケージ名」との比較で十分ではないか？ と疑問を持たれた方もいるかと思う。しかしパッケージ名は任意に指定できるため他のアプリへの成りすましが簡単である。成りすまし可能なパラメータは身元確認用には使えない。一方、アプリの持つ証明書であれば身元確認に使うことができる。証明書に対応する署名用の開発者鍵は本物のアプリ開発者しか持っていないため、第三者が同じ証明書を持ち、尚且つ署名検証が成功するアプリを作成することはできないからだ。ホワイトリストはアクセスを許可したいアプリの証明書データを丸ごと保持してもよいが、サンプルコードではホワイトリストのデータサイズを小さくするために証明書データの SHA-256 ハッシュ値を保持することにしている。

この方法には次の二つの制約条件がある。

- 利用元アプリにおいて startActivity() ではなく startActivityForResult() を使用しなければならない
- 利用元アプリにおいて Activity 以外から呼び出すことはできない

2 つ目の制約事項はいわば 1 つ目の制約事項の結果として課される制約であるので、厳密には 1 つの同じ制約と言える。この制約は呼び出し元アプリのパッケージ名を取得する Activity.getCallingPackage() の制約により生じている。Activity.getCallingPackage() は startActivityForResult() で呼び出された場合にのみ利用元アプリのパッケージ名を返すが、残念ながら startActivity() で呼び出された場合には null を返す仕様となっている。そのためここで紹介する方法は必ず利用元アプリが、たとえ戻り値が不要であったとしても、startActivityForResult() を使わなければならないという制約がある。さらに startActivityForResult() は Activity クラスでしか使えないため、利用元は Activity に限定されるという制約もある。

PartnerActivity.java

```
package org.jssec.android.activity.partneractivity;

import org.jssec.android.shared.PkgCertWhitelists;
import org.jssec.android.shared.Utils;

import android.app.Activity;
import android.content.Context;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.widget.Toast;

public class PartnerActivity extends Activity {

    // ★ポイント 4★ 利用元アプリの証明書がホワイトリストに登録されていることを確認する
```

(continues on next page)

(continued from previous page)

```
    private static PkgCertWhitelists sWhitelists = null;
    private static void buildWhitelists(Context context) {
        boolean isdebug = Utils.isDebuggable(context);
        sWhitelists = new PkgCertWhitelists();

        // パートナーアプリ org.jssec.android.activity.partneruser の証明書ハッシュ値を登録
        sWhitelists.add("org.jssec.android.activity.partneruser", isdebug ?
                // debug.keystore の"androiddebugkey"の証明書ハッシュ値
                "0EFB7236 328348A9 89718BAD DF57F544 D5CCB4AE B9DB34BC 1E29DD26 F77C8255" :
                // keystore の"partner key"の証明書ハッシュ値
                "1F039BB5 7861C27A 3916C778 8E78CE00 690B3974 3EB8259F E2627B8D 4C0EC35A");

        // 以下同様に他のパートナーアプリを登録...
    }
    private static boolean checkPartner(Context context, String pkgname) {
        if (sWhitelists == null) buildWhitelists(context);
        return sWhitelists.test(context, pkgname);
    }

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        // ★ポイント 4★ 利用元アプリの証明書がホワイトリストに登録されていることを確認する
        if (!checkPartner(this, getCallingActivity().getPackageName())) {
            Toast.makeText(this, "利用元アプリはパートナーアプリではない。", Toast.LENGTH_LONG).show();
            finish();
            return;
        }

        // ★ポイント 5★ パートナーアプリからの Intent であっても、受信 Intent の安全性を確認する
        // サンプルにつき割愛。「3.2 入力データの安全性を確認する」を参照。
        Toast.makeText(this, "パートナーアプリからアクセスあり", Toast.LENGTH_LONG).show();
    }

    public void onReturnResultClick(View view) {

        // ★ポイント 6★ パートナーアプリに開示してよい情報に限り返送してよい
        Intent intent = new Intent();
        intent.putExtra("RESULT", "パートナーアプリに開示してよい情報");
        setResult(RESULT_OK, intent);
        finish();
    }
}
```

PkgCertWhitelists.java

```
package org.jssec.android.shared;

import android.content.pm.PackageManager;
import java.util.HashMap;
import java.util.Map;
import android.content.Context;
import android.os.Build;
```

(continues on next page)

(continued from previous page)

```
import static android.content.pm.PackageManager.CERT_INPUT_SHA256;

public class PkgCertWhitelists {
    private Map<String, String> mWhitelists = new HashMap<String, String>();

    public boolean add(String pkgname, String sha256) {
        if (pkgname == null) return false;
        if (sha256 == null) return false;

        sha256 = sha256.replaceAll(" ", "");
        if (sha256.length() != 64) return false;     // SHA-256 は 32 バイト
        sha256 = sha256.toUpperCase();
        if (sha256.replaceAll("[0-9A-F]+", "").length() != 0) return false; // 0-9A-F 以外の文字がある

        mWhitelists.put(pkgname, sha256);
        return true;
    }

    public boolean test(Context ctx, String pkgname) {
        // pkgname に対応する正解のハッシュ値を取得する
        String correctHash = mWhitelists.get(pkgname);
        android.util.Log.d("Partner", "hash=" + correctHash);
        // pkgname の実際のハッシュ値と正解のハッシュ値を比較する
        if (Build.VERSION.SDK_INT >= 28) {
            // ★ API Level >= 28 では Package Manager の API で直接検証が可能
            PackageManager pm = ctx.getPackageManager();
            return pm.hasSigningCertificate(pkgname, hex2Bytes(correctHash), CERT_INPUT_SHA256);
        } else {
            // API Level < 28 の場合は PkgCert の機能を利用する
            return PkgCert.test(ctx, pkgname, correctHash);
        }
    }

    private byte[] hex2Bytes(String s) {
        int len = s.length();
        byte[] data = new byte[len / 2];
        for (int i = 0; i < len; i += 2) {
            data[i / 2] = (byte) ((Character.digit(s.charAt(i), 16) << 4)
                    + Character.digit(s.charAt(i+1), 16));
        }
        return data;
    }
}
```

PkgCert.java

```
package org.jssec.android.shared;

import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

import android.content.Context;
import android.content.pm.PackageInfo;
import android.content.pm.PackageManager;
```

(continues on next page)

(continued from previous page)

```
import android.content.pm.PackageManager.NameNotFoundException;
import android.content.pm.Signature;

public class PkgCert {

    public static boolean test(Context ctx, String pkgname, String correctHash) {
        if (correctHash == null) return false;
        correctHash = correctHash.replaceAll(" ", "");
        return correctHash.equals(hash(ctx, pkgname));
    }

    public static String hash(Context ctx, String pkgname) {
        if (pkgname == null) return null;
        try {
            PackageManager pm = ctx.getPackageManager();
            PackageInfo pkginfo = pm.getPackageInfo(pkgname, PackageManager.GET_SIGNATURES);
            if (pkginfo.signatures.length != 1) return null;    // 複数署名は扱わない
            Signature sig = pkginfo.signatures[0];
            byte[] cert = sig.toByteArray();
            byte[] sha256 = computeSha256(cert);
            return byte2hex(sha256);
        } catch (NameNotFoundException e) {
            return null;
        }
    }

    private static byte[] computeSha256(byte[] data) {
        try {
            return MessageDigest.getInstance("SHA-256").digest(data);
        } catch (NoSuchAlgorithmException e) {
            return null;
        }
    }

    private static String byte2hex(byte[] data) {
        if (data == null) return null;
        final StringBuilder hexadecimal = new StringBuilder();
        for (final byte b : data) {
            hexadecimal.append(String.format("%02X", b));
        }
        return hexadecimal.toString();
    }
}
```

4.1.3.3 Activity に送信される Intent の読み取り (Android 5.0 より前のバージョンについて)

Android 5.0(API Level 21) 以降では、getRecentTasks() から取得できる情報が、自分自身のタスク情報およびホームアプリのような機密情報ではないものに限定された。しかし、Android 5.0 より前のバージョンでは、自分自身のタスク以外の情報も読み取ることができる。以下に Android 5.0 より前のバージョンで発生する本問題の内容を解説する。

タスクのルート Activity に送信された Intent は、タスク履歴に追加される。ルート Activity とはタスクの起点となる Activity のことである。タスク履歴に追加された Intent は、ActivityManager クラスを使うことでどのアプリからも自由に読み出すことが可能である。

アプリからタスク履歴を参照するためのサンプルコードを以下に示す。タスク履歴を参照するためには、AndroidManifest.xml に GET_TASKS Permission の利用を指定する。

AndroidManifest.xml

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="org.jssec.android.intent.maliciousactivity" >

    <!-- GET_TASKS Permission を指定する -->
    <uses-permission android:name="android.permission.GET_TASKS" />

    <application
        android:allowBackup="false"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name=".MaliciousActivity"
            android:label="@string/title_activity_main"
            android:exported="true" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

MaliciousActivity.java

```
package org.jssec.android.intent.maliciousactivity;

import java.util.List;
import java.util.Set;

import android.app.Activity;
import android.app.ActivityManager;
import android.content.Intent;
import android.os.Bundle;
import android.util.Log;

public class MaliciousActivity extends Activity {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.malicious_activity);

        // ActivityManager を取得する
        ActivityManager activityManager = (ActivityManager) getSystemService(ACTIVITY_SERVICE);
        // タスクの履歴を最新 100 件取得する
        List<ActivityManager.RecentTaskInfo> list = activityManager
                .getRecentTasks(100, ActivityManager.RECENT_WITH_EXCLUDED);
        for (ActivityManager.RecentTaskInfo r : list) {
            // ルート Activity に送信された Intent を取得し、Log に表示する
            Intent intent = r.baseIntent;
```

(continues on next page)

(continued from previous page)

```
            Log.v("baseIntent", intent.toString());
            Log.v("  action:", intent.getAction());
            Log.v("  data:", intent.getDataString());
            if (r.origActivity != null) {
                Log.v("  pkg:", r.origActivity.getPackageName() + r.origActivity.getClassName());
            }
            Bundle extras = intent.getExtras();
            if (extras != null) {
                Set<String> keys = extras.keySet();
                for(String key : keys) {
                    Log.v("  extras:", key + "=" + extras.get(key).toString());
                }
            }
        }
    }
}
```

ActivityManager クラスの getRecentTasks() により、指定した件数のタスク履歴を取得することができる。各タスクの情報は ActivityManager.RecentTaskInfo クラスのインスタンスに格納されるが、そのメンバー変数 baseIntent には、タスクのルート Activity に送信された Intent が格納されている。ルート Activity とはタスクが生成された時に呼び出された Activity であるので、Activity を呼び出す際には、以下の条件をどちらも満たさないように注意しなければならない。

- Activity が呼び出された際に、タスクが新規に生成される
- 呼び出された Activity がバックグラウンドあるいはフォアグラウンド上に既に存在するタスクのルート Activity である

4.1.3.4 ルート Activity について

ルート Activity とはタスクの起点となる Activity のことである。タスクが生成された時に起動された Activity のことである、と言ってもよい。例えば、デフォルト設定の Activity がランチャーから起動された場合、その Activity はルート Activity となる。Android の仕様によると、ルート Activity に送信される Intent の内容は任意のアプリによって読み取られる恐れがある。そこで、ルート Activity へセンシティブな情報を送信しないように対策を講じる必要がある。本ガイドでは、呼び出された Activity がルートとなるのを防ぐために以下の 3 点をルールに掲げた。

- taskAffinity を指定しない
- launchMode を指定しない
- Activity に送信する Intent には FLAG_ACTIVITY_NEW_TASK を設定しない

以下、Activity がルートとなるのはどのような場合かを中心にルート Activity について考察する。呼び出された Activity がルートとなるための条件に関連するのは、以下に挙げる項目である。

- 呼び出される Activity の起動モード
- 呼び出される Activity のタスクとその起動状態

まず、「呼び出される Activity の起動モード」について説明する。Activity の起動モードは、AndroidManifest.xml に android:launchMode を記述することで設定できる。記述しない場合は”standard” とみなされる。また、起動モードは Intent に設定するフラグによっても変更可能である。FLAG_ACTIVITY_NEW_TASK フラグは、Activity を”singleTask” モードで起動させる。

指定可能な起動モードは次のとおりである。特に、ルート Activity との関連に焦点を当てて説明する。

standard

このモードで呼び出された Activity はルートとなることはなく、呼び出し側のタスクに所属する。また、呼び出しの度に Activity のインスタンスが生成される。

singleTop

”standard” と同様であるが、フォアグラウンドタスクの最前面に表示されている Activity を起動する場合には、インスタンスが生成されないという点が異なる。

singleTask

Activity はアフィニティの値に従って所属するタスクが決まる。Activity のアフィニティと一致するタスクがバックグラウンドあるいはフォアグラウンドに存在しない場合には、タスクが Activity のインスタンスとともに新規に生成される。存在する場合にはどちらも生成されない。前者では、起動された Activity のインスタンスはルートとなる。

singleInstance

”singleTask” と同様であるが、次の点で異なる。新規に生成されたタスクには、ルート Activity のみが所属できる点である。したがって、このモードで起動された Activity のインスタンスは常にルートである。ここで注意が必要なのは、呼び出される Activity が持つアフィニティと同じ名前のタスクが既に存在している場合であっても、呼び出される Activity とタスクに含まれる Activity のクラス名が異なる場合である。その場合は、新規にタスクが生成される。

以上より、”singleTask” または”singleInstance” で起動された Activity はルートになる可能性があることが分かる。アプリの安全性を確保するためには、これらのモードで起動しないようにしなければならない。

次に、「呼び出される Activity のタスクとその起動状態」について説明する。たとえ、Activity が”standard” モードで呼び出されたとしても、その Activity が所属するタスクの状態によってルート Activity となる場合がある。

例として、呼び出される Activity のタスクが既にバックグラウンドで起動している場合を考える。問題となるのは、そのタスクの Activity インスタンスが”singleInstance” で起動している場合である。”standard” で呼び出された Activity のアフィニティがタスクと同じだった時に、既存の”singleInstance” の Activity の制限により、新規タスクが生成される。ただし、それぞれの Activity のクラス名が同じ場合は、タスクは生成されず、インスタンスは既存のものが利用される。いずれにしろ、呼び出された Activity はルート Activity になる。

以上のように、ルート Activity が呼び出される条件は実行時の状態に依存するなど複雑である。アプリ開発の際には、”standard” モードで Activity を呼び出すように工夫すべきである。

非公開 Activity に送信される Intent が他アプリから読み取られる例として、非公開 Activity の呼び出し側 Activity を”singleInstance” モードで起動する場合のサンプルコードを以下に示す。このサンプルコードでは、非公開 Activity が”standard” モードで起動されるが、呼び出し側 Activity の”singleInstance” モードの条件により、非公開 Activity は新規タスクのルート Activity となってしまう。この時、非公開 Activity に送信されるセンシティブな情報はタスク履歴に記録されるため、任意のアプリから読み取り可能である。なお、呼び出し側 Activity、非公開 Activity ともに同一のアフィニティを持つ。

AndroidManifest.xml(非推奨)

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="org.jssec.android.activity.singleinstanceactivity" >

    <application
        android:allowBackup="false"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name" >

        <!-- ルート Activity の起動モードを”singleInstance”とする -->
        <!-- アフィニティは設定せず、アプリのパッケージ名とする -->
        <activity
            android:name=".PrivateUserActivity"
            android:label="@string/app_name"
            android:launchMode="singleInstance"
            android:exported="true" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

        <!-- 非公開 Activity -->
        <!-- 起動モードを”standard”とする -->
        <!-- アフィニティは設定せず、アプリのパッケージ名とする -->
        <activity
            android:name=".PrivateActivity"
            android:label="@string/app_name"
            android:exported="false" />
    </application>
</manifest>
```

非公開 Activity は、受信した Intent に対して結果を返すのみである。

PrivateActivity.java

```
package org.jssec.android.activity.singleinstanceactivity;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.widget.Toast;

public class PrivateActivity extends Activity {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.private_activity);

        // 受信 Intent の安全性を確認する
        // サンプルにつき割愛。「3.2 入力データの安全性を確認する」を参照。
        String param = getIntent().getStringExtra("PARAM");
        Toast.makeText(this, String.format("パラメータ「%s」を受け取った。", param), Toast.LENGTH_LONG).
↪show();
```

(continues on next page)

(continued from previous page)

```
    }

    public void onReturnResultClick(View view) {
        Intent intent = new Intent();
        intent.putExtra("RESULT", "センシティブな情報");
        setResult(RESULT_OK, intent);
        finish();
    }
}
```

非公開 Activity の呼び出し側では、Intent にフラグを設定せずに、”standard” モードで非公開 Activity を起動している。

PrivateUserActivity.java

```
package org.jssec.android.activity.singleinstanceactivity;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.widget.Toast;

public class PrivateUserActivity extends Activity {

    private static final int REQUEST_CODE = 1;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.user_activity);
    }

    public void onUseActivityClick(View view) {

        // 非公開 Activity を "standard"モードで起動する
        Intent intent = new Intent();
        intent.setClass(this, PrivateActivity.class);
        intent.putExtra("PARAM", "センシティブな情報");

        startActivityForResult(intent, REQUEST_CODE);
    }

    @Override
    public void onActivityResult(int requestCode, int resultCode, Intent data) {
        super.onActivityResult(requestCode, resultCode, data);

        if (resultCode != RESULT_OK) return;

        switch (requestCode) {
        case REQUEST_CODE:
            String result = data.getStringExtra("RESULT");

            // 受信データの安全性を確認する
            // サンプルにつき割愛。「3.2 入力データの安全性を確認する」を参照。
            Toast.makeText(this, String.format("結果「%s」を受け取った。", result), Toast.LENGTH_LONG).
↪show();
```

(continues on next page)

(continued from previous page)

```
            break;
        }
    }
}
```

4.1.3.5 Activity 利用時のログ出力について

Activity を利用する際に ActivityManager が Intent の内容を LogCat に出力する。以下の内容は LogCat に出力されるため、センシティブな情報が含まれないように注意すべきだ。

- 利用先パッケージ名
- 利用先クラス名
- Intent#setData() で設定した URI

例えば、メール送信する場合、URI にメールアドレスを指定して Intent を発行するとメールアドレスが LogCat に出力されてしまう。Intent#putExtra() で設定した値は LogCat に出力されないため、Extras に設定して送るようにした方が良い。

次のようにメール送信すると LogCat にメールアドレスが表示されてしまう

```
MainActivity.java
    // URI は LogCat に出力される
    Uri uri = Uri.parse("mailto:test@gmail.com");
    Intent intent = new Intent(Intent.ACTION_SENDTO, uri);
    startActivity(intent);
```

次のように Extras を使用すると LogCat にメールアドレスが表示されなくなる

```
MainActivity.java
    // Extra に設定した内容は LogCat に出力されない
    Uri uri = Uri.parse("mailto:");
    Intent intent = new Intent(Intent.ACTION_SENDTO, uri);
    intent.putExtra(Intent.EXTRA_EMAIL, new String[] {"test@gmail.com"});
    startActivity(intent);
```

ただし、ActivityManager#getRecentTasks() によって Intent の Extras を他のアプリから直接読める場合があるので、注意すること。詳しくは「4.1.2.2. *taskAffinity* を指定しない （必須)」、「4.1.2.3. *launchMode* を指定しない （必須)」および「4.1.2.4. *Activity* に送信する *Intent* には *FLAG_ACTIVITY_NEW_TASK* を設定しない （必須)」を参照のこと。

4.1.3.6 PreferenceActivity の Fragment Injection 対策について

PreferenceActivity を継承したクラスが公開 Activity となっている場合、Fragment Injection[*5] と呼ばれる問題が発生する可能性がある。この問題を防ぐためには PreferenceActivity.IsValidFragment() を override し、引数の値を適切にチェックすることで Activity が意図しない Fragment を扱わないようにする必要がある。(入力データの安全性については「3.2. 入力データの安全性を確認する」参照)

以下に、IsValidFragment() を override したサンプルを示す。なお、ソースコードの難読化を行うと、クラス名が変わり、引数の値との比較結果が変わってまう可能性があるので、別途対応が必要になる。

[*5] Fragment Injection の詳細は以下の URL を参照のこと https://securityintelligence.com/new-vulnerability-android-framework-fragment-injection/

override した isValidFragment() メソッドの例

```
protected boolean isValidFragment(String fragmentName) {
    // 難読化時の対応は別途行うこと
    return PreferenceFragmentA.class.getName().equals(fragmentName)
            || PreferenceFragmentB.class.getName().equals(fragmentName)
            || PreferenceFragmentC.class.getName().equals(fragmentName)
            || PreferenceFragmentD.class.getName().equals(fragmentName);
}
```

なお、アプリの targetSdkVersion が 19 以上である場合、PreferenceActivity.isValidFragment() を override しないと、Fragment が挿入された段階（isValidFragment() が呼ばれた段階）でセキュリティ例外が発生しアプリが終了するため、PreferenceActivity.isValidFragment() の override が必須である。

4.1.3.7 Autofill フレームワークについて

Autofill フレームワークは Android 8.0(API Level 26) で追加された仕組みである。この仕組みを利用することで、ユーザー名、パスワード、住所、電話番号、クレジットカード情報等が入力されたときにそれらを保存しておき、再度必要になった時に取り出してアプリに自動入力 (Autofill) する機能を実現することができる。ユーザーの入力負荷を軽減することができる便利な仕組みであるが、あるアプリが扱うパスワードやクレジットカード情報等のセンシティブな情報を他のアプリ（Autofill service）に渡すことを想定しており、取り扱いには十分に気をつける必要がある。

仕組み (概要)

2 つのコンポーネント

以下に、Autofill フレームワークに登場する 2 つのコンポーネント[*6] の概要を示す。

- Autofill の対象となるアプリ（利用アプリ）:
  - View の情報（テキストおよび属性) を Autofill service に渡したり、Autofill service から Autofill に必要な情報を提供されたりする。
  - Activity を持つすべてのアプリが利用アプリとなる（Foreground 時)。
  - 利用アプリが持つすべての View が Autofill の対象になる可能性がある。View 単位で明示的に Autofill の対象外とすることも可能。
  - Autofill 機能の利用を同一パッケージ内の Autofill service に限定することも可能。
- Autofill service を提供するサービス（Autofill service）:
  - アプリから渡された View の情報を保存したり（ユーザーの許可が必要)、View に Autofill するための情報（候補リスト）をアプリに提供したりする。
  - 保存対象の View は Autofill service が決定する（Autofill フレームワークはデフォルトで Activity に含まれるすべての View の情報を Autofill service に渡す)。
  - 3rd Party 製の Autofill service も作成できる。

[*6] 「利用アプリ」と「Autofill service」は、それぞれ同じパッケージ (APK ファイル) であることも、別パッケージであることもあり得る。

- 端末内に複数存在することが可能でユーザーにより「設定」から選択された Service のみ有効になる（「なし」も選択可能）。
- Service が、扱うユーザー情報を保護するためにパスワード入力等によってユーザー認証をするための UI を持つことも可能。

Autofill フレームワークの処理フロー

図 4.1.6 は Autofill 時の Autofill 関連コンポーネント間の処理フローを示している。利用アプリの View のフォーカス移動等を契機に Autoill フレームワークを介して View の情報（主に View の親子関係や個々の属性）が「設定」で選択された Autofill service に渡る。Autofill service は渡された情報を元に Autofill に必要な情報（候補リスト）を DB から取り出し、フレームワークに返信する。フレームワークは候補リストをユーザーに提示し、ユーザーが選択したデータによりアプリで Autofill が行われる。

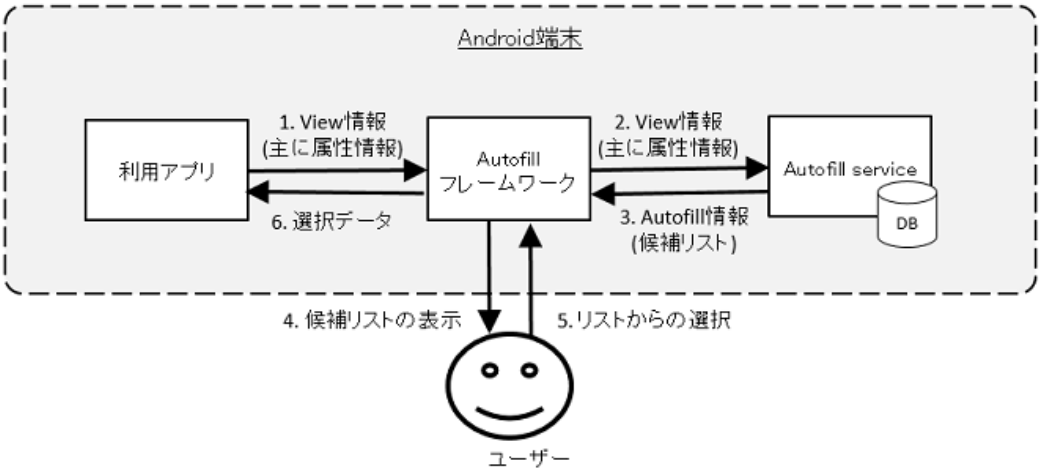

図 4.1.6　Autofill 時のコンポーネント間の処理フロー

一方、図 4.1.7 は Autofill によるユーザーデータ保存時の処理フローを示している。AutofillManager#commit() の呼び出しや Activity の終了を契機に、Autofill した View の値に変更があり、かつ、Autofill フレームワークが表示する保存許可ダイアログに対してユーザーが許可した場合、View の情報（テキスト含む）が Autofill フレームワークを介して「設定」で選択された Autofill service に渡され、Autofill service が View の情報を DB に保存して一連の処理が完了となる。

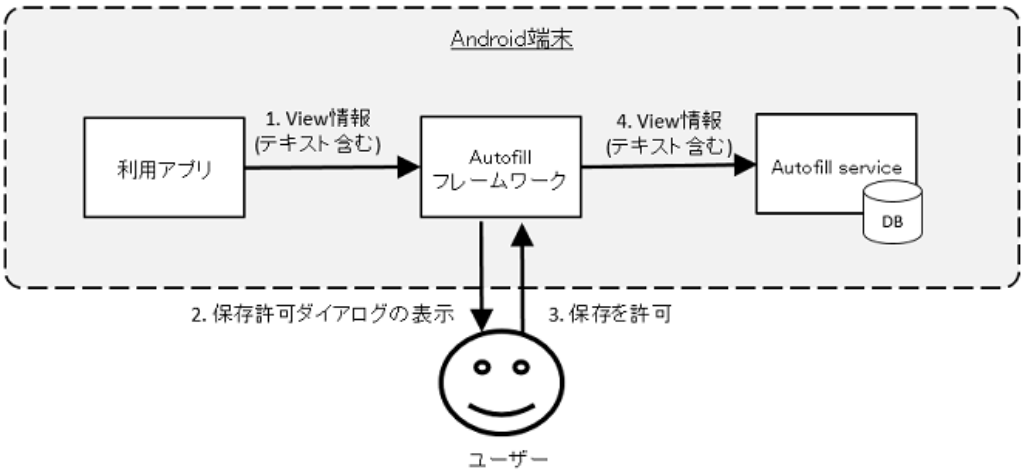

図 4.1.7　ユーザーデータ保存時のコンポーネント間の処理フロー

Autofill 利用アプリにおけるセキュリティ上の懸案

「仕組み（概要）」の項で示した通り Autofill フレームワークにおけるセキュリティモデルでは、ユーザーが「設定」で安全な Autofill service を選択し、保存時にどのデータをどの Autofill service に渡してもよいか適切に判断できることを前提としている。

ところが、ユーザーがうっかり安全でない Autofill service を選択したり、Autofill service に渡すべきでないセンシティブな情報の保存を許可してしまったりする可能性がある。以下に、この場合に起きうる被害について考察する。

保存時、ユーザーが Autofill service を選択し、保存許可ダイアログに対して許可した場合、利用アプリで表示されている Activity に含まれるすべての View の情報が Autofill service に渡る可能性がある。ここで、Autofill service がマルウェアの場合や、Autofill service に View の情報を外部ストレージや安全でないクラウドサービスに保存する等のセキュリティ上の問題があった場合には、利用アプリで扱う情報の漏洩につながってしまうリスクが考えられる。

一方、Autofill 時、ユーザーが Autofill service としてマルウェアを選択してしまっていた場合、マルウェアが送信した値を入力してしまう可能性がある。ここで、アプリやアプリのデータを送信した先のクラウドサービスが入力データの安全性を十分に確認していなかった場合、情報漏洩やアプリ／サービスの停止等につながってしまうリスクが考えられる。

なお、「2 つのコンポーネント」で書いたように、Activity を持つアプリが自動的に Autofill の対象になるため、Activity を持つアプリのすべての開発者は上記のリスクを考慮して設計や実装を行う必要がある。以下に、上記のリスクに対する対策案を示すが、「3.1.3. 資産分類と保護施策」なども参考にして、アプリに必要な対策を検討した上で、適用することをお勧めする。

リスクに対する対策**-1**

前述のように、Autofill フレームワークでは基本的にユーザーの裁量によってセキュリティが担保されている。そのためアプリでできる対策は限られているが、View に対して importantForAutofill 属性で”no” 等を指定して Autofill service に View の情報を渡さないようにする（Autofill の対象外とする）ことで、ユーザーが適切な選択や許可をできなかった場合（マルウェアを Autofill service として利用するように選択する等）でも、上記の懸案を軽減することができる。[*7]

importantForAutofill 属性は、以下のいずれかの方法によって指定することができる。

- レイアウト XML の importantForAutofill 属性を指定する
- View#setImportantForAutofill() を呼び出す

以下に指定可能な値を示す。指定する範囲によって適切な値を使うこと。特に、”no” を指定した場合、指定した View は Autofill の対象外になるが、子供は Autofill の対象になることに注意すること。デフォルト値は、”auto” となっている。

[*7] ユーザーが意図的に Autofill 機能を利用した場合など、本対策でも上記の懸案を回避できないことがある。「リスクに対する対策-2」を実施することでこのような場合にも対応することができる。

表 4.1.2　Autofill の対象になるか

値　　定数名	指定した View	子供の View
”auto”　IMPORTANT_FOR_AUTOFILL_AUTO	自動[*8]	自動[*8]
”no”　IMPORTANT_FOR_AUTOFILL_NO	対象外	対象
”noExcludeDescendants”　IMPORTANT_FOR_AUTOFILL_NO_EXCLUDE_DESCENDANTS	対象外	対象外
”yes”　IMPORTANT_FOR_AUTOFILL_YES	対象	対象
”yesExcludeDescendants”　IMPORTANT_FOR_AUTOFILL_YES_EXCLUDE_DESCENDANTS	対象	対象外

また、AutofillManager#hasEnabledAutofillServices() を利用して、Autofill 機能の利用を同一パッケージ内の Autofill service に限定することも可能である。

以下に、「設定」で同一パッケージ内の Autofill service を利用するように設定されている場合のみ、Activity の全ての View を Autofill の対象にする（実際に Autofill の対象になるかは Autofill service 次第）場合の例を示す。個別の View に対して View#setImportantForAutofill() を呼び出すことも可能である。

DisableForOtherServiceActivity.java

```
package org.jssec.android.autofillframework.autofillapp;

import android.os.Bundle;
import android.support.v7.app.AppCompatActivity;
import android.view.View;
import android.view.autofill.AutofillManager;
import android.widget.EditText;
import android.widget.TextView;

import org.jssec.android.autofillframework.R;

public class DisableForOtherServiceActivity extends AppCompatActivity {
    private boolean mIsAutofillEnabled = false;

    private EditText mUsernameEditText;
    private EditText mPasswordEditText;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.disable_for_other_service_activity);
```

(continues on next page)

[*8] Autofill フレームワークが決定

(continued from previous page)

```
        mUsernameEditText = (EditText)findViewById(R.id.field_username);
        mPasswordEditText = (EditText)findViewById(R.id.field_password);

        findViewById(R.id.button_login).setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                login();
            }
        });

        findViewById(R.id.button_clear).setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                resetFields();
            }
        });
        //この Activity ではフローティングツールバーの対応をしていないため、
        // 「自動入力」の選択で Autofill 機能が利用可能になります。
    }

    @Override
    protected void onStart() {
        super.onStart();
    }

    @Override
    protected void onResume() {
        super.onResume();
        updateAutofillStatus();

        if (!mIsAutofillEnabled) {
            View rootView = this.getWindow().getDecorView();
            //同一パッケージ内の Autofill service を利用しない場合は、全ての View を Autofill の対象外にする
            rootView.setImportantForAutofill(View.IMPORTANT_FOR_AUTOFILL_NO_EXCLUDE_DESCENDANTS);
        } else {
            //同一パッケージ内の Autofill service を利用する場合は、全ての View を Autofill の対象にする
            //特定の View に対して View#setImportantForAutofill() を呼び出すことも可能
            View rootView = this.getWindow().getDecorView();
            rootView.setImportantForAutofill(View.IMPORTANT_FOR_AUTOFILL_AUTO);
        }
    }
    private void login() {
        String username = mUsernameEditText.getText().toString();
        String password = mPasswordEditText.getText().toString();

        //View から取得したデータの安全性を確認する
        if (!Util.validateUsername(username) || !Util.validatePassword(password)) {
            //適切なエラー処理をする
        }

        //サーバーに username, password を送信

        finish();
    }
```

(continues on next page)

(continued from previous page)

```
    private void resetFields() {
        mUsernameEditText.setText("");
        mPasswordEditText.setText("");
    }

    private void updateAutofillStatus() {
        AutofillManager mgr = getSystemService(AutofillManager.class);

        mIsAutofillEnabled = mgr.hasEnabledAutofillServices();

        TextView statusView = (TextView) findViewById(R.id.label_autofill_status);
        String status = "自社の autofill service が--です";
        if (mIsAutofillEnabled) {
            status = "同一パッケージ内の autofill service が有効です";
        } else {
            status = "同一パッケージ内の autofill service が無効です";
        }
        statusView.setText(status);
    }
}
```

リスクに対する対策-2

アプリで「リスクに対する対策-1」を施した場合でも、ユーザーが View の長押しでフローティングツールバーなどを表示させて「自動入力」を選択すると、強制的に Autofill を利用できてしまう。この場合、importantForAutofill 属性で"no"等を指定した View を含む全ての View の情報が Autofill Service に渡ることになる。

「リスクに対する対策-1」に加えて、フローティングツールバーなどのメニューから「自動入力」を削除することで、上記のような場合でも、情報漏えいのリスクを回避することができる。

以下にサンプルコードを示す。

DisableAutofillActivity.java

```
package org.jssec.android.autofillframework.autofillapp;

import android.os.Bundle;
import android.support.v7.app.AppCompatActivity;
import android.view.ActionMode;
import android.view.Menu;
import android.view.MenuItem;
import android.view.SubMenu;
import android.view.View;
import android.widget.EditText;

import org.jssec.android.autofillframework.R;

public class DisableAutofillActivity extends AppCompatActivity {
    private EditText mUsernameEditText;
    private EditText mPasswordEditText;
    private ActionMode.Callback mActionModeCallback;

    @Override
```

(continues on next page)

(continued from previous page)

```
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.disable_autofill_activity);
        mUsernameEditText = (EditText) findViewById(R.id.field_username);
        mPasswordEditText = (EditText) findViewById(R.id.field_password);

        findViewById(R.id.button_login).setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                login();
            }
        });

        findViewById(R.id.button_clear).setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                resetFields();
            }
        });

        mActionModeCallback = new ActionMode.Callback() {
            @Override
            public boolean onCreateActionMode(ActionMode mode, Menu menu) {
                removeAutofillFromMenu(menu);
                return true;
            }

            @Override
            public boolean onPrepareActionMode(ActionMode mode, Menu menu) {
                removeAutofillFromMenu(menu);
                return true;
            }

            @Override
            public boolean onActionItemClicked(ActionMode mode, MenuItem item) {
                return false;
            }

            @Override
            public void onDestroyActionMode(ActionMode mode) {
            }
        };

        //フローティングツールバーから「自動入力」を削除する
        setMenu();
    }

    void setMenu() {
        if (mActionModeCallback == null) {
            return;
        }
         //Activity に含まれる全ての編集可能な TextView についてセットすること
         mUsernameEditText.setCustomInsertionActionModeCallback(mActionModeCallback);
         mPasswordEditText.setCustomInsertionActionModeCallback(mActionModeCallback);
    }
```

(continues on next page)

(continued from previous page)

```
    //Menu の階層を巡って「自動入力」を削除する
    void removeAutofillFromMenu(Menu menu) {
        if (menu.findItem(android.R.id.autofill) != null) {
            menu.removeItem(android.R.id.autofill);
        }

        for (int i=0; i<menu.size(); i++) {
            SubMenu submenu = menu.getItem(i).getSubMenu();
            if (submenu != null) {
                removeAutofillFromMenu(submenu);
            }
        }
    }

    private void login() {
        String username = mUsernameEditText.getText().toString();
        String password = mPasswordEditText.getText().toString();

        //View から取得したデータの安全性を確認する
        if (!Util.validateUsername(username) || Util.validatePassword(password)) {
            //適切なエラー処理をする
        }

        //サーバーに username, password を送信

        finish();
    }

    private void resetFields() {
        mUsernameEditText.setText("");
        mPasswordEditText.setText("");
    }

}
```

リスクに対する対策-3

Android 9.0 (API Level 28) で、現在有効となっている Autofill Service のコンポーネントが何であるかを AutofillManager#getAutofillServiceComponentName() によって知ることができるようになった。これを用いてパッケージ名を取得し、自身のアプリが信頼を置いている Autofill Service かどうかを確認できる。

この場合、先に「4.1.3.2. 利用元アプリを確認する」で述べた通りパッケージ名は成りすましが可能なため、これのみで身元確認をするのは推奨できない。4.1.3.2. に掲載の例と同じく、パッケージ名から Autofill Service の証明書を取得し、それをあらかじめホワイトリストに登録されているものと一致するかどうかによって身元確認を行うべきである。この方法の詳細については 4.1.3.2. に詳しいのでそちらを参照されたい。

以下に、あらかじめホワイトリストに登録されている Autofill Service が有効になっている場合のみ、Activity の全ての View を Autofill の対象にする場合の例を示す。

EnableOnlyWhitelistedServiceActivity.java

```
package org.jssec.android.autofillframework.autofillapp;
```

(continues on next page)

(continued from previous page)

```
import android.content.ComponentName;
import android.content.Context;
import android.os.Bundle;
import android.app.Activity;
import android.view.View;
import android.view.autofill.AutofillManager;
import android.widget.EditText;
import android.widget.TextView;
import android.widget.Toast;

import org.jssec.android.shared.PkgCertWhitelists;
import org.jssec.android.autofillframework.R;

public class EnableOnlyWhitelistedServiceActivity extends Activity {
    private static PkgCertWhitelists sWhitelists = null;
    private static void buildWhitelists(Context context) {
        isdebug = Utils.isDebuggable(context);
        sWhitelists = new PkgCertWhitelists();
        // 信頼する Autofill Service の証明書ハッシュ値を登録
        sWhitelists.add("org.jssec.android.autofillframework.autofillservice", isdebug ?
            // debug.keystore の"androiddebugkey"の証明書ハッシュ値
            "0EFB7236 328348A9 89718BAD DF57F544 D5CCB4AE B9DB34BC 1E29DD26 F77C8255" :
            // keystore の"partner key"の証明書ハッシュ値
            "1F039BB5 7861C27A 3916C778 8E78CE00 690B3974 3EB8259F E2627B8D 4C0EC35A");
        // 同様に他の信頼する Autofill Service を登録する
        //    :
    }
    private static boolean checkService(Context context, String pkgname) {
        if (sWhitelists == null) buildWhitelists(context);
        return sWhitelists.test(context, pkgname);
    }

    private boolean mIsAutofillEnabled = false;

    private EditText mUsernameEditText;
    private EditText mPasswordEditText;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.enable_only_whitelisted_service_activity);

        mUsernameEditText = (EditText)findViewById(R.id.field_username);
        mPasswordEditText = (EditText)findViewById(R.id.field_password);

        findViewById(R.id.button_login).setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                login();
            }
        });
        findViewById(R.id.button_clear).setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
```

(continues on next page)

(continued from previous page)

```
                resetFields();
            }
        });
        //この Activity ではフローティングツールバーの対応をしていないため、
        // 「自動入力」の選択で Autofill 機能が利用可能になる
    }

    @Override
    protected void onStart() {
        super.onStart();
    }

    @Override
    protected void onResume() {
        super.onResume();
        updateAutofillStatus();

        if (!mIsAutofillEnabled) {
            View rootView = this.getWindow().getDecorView();
            // ホワイトリストに登録されている Autofill Service でない場合は、全ての View を Autofill の対象外にする
            rootView.setImportantForAutofill(View.IMPORTANT_FOR_AUTOFILL_NO_EXCLUDE_DESCENDANTS);
        } else {
            // ホワイトリストに登録されている Autofill Service の場合は、全ての View を Autofill の対象にする
            // 特定の View に対して View#setImportantForAutofill() を呼び出すことも可能
            View rootView = this.getWindow().getDecorView();
            rootView.setImportantForAutofill(View.IMPORTANT_FOR_AUTOFILL_AUTO);
        }
    }
    private void login() {
        String username = mUsernameEditText.getText().toString();
        String password = mPasswordEditText.getText().toString();

        //View から取得したデータの安全性を確認する
        if (!Util.validateUsername(username) || !Util.validatePassword(password)) {
            //適切なエラー処理をする
        }

        //サーバーに username, password を送信

        finish();
    }

    private void resetFields() {
        mUsernameEditText.setText("");
        mPasswordEditText.setText("");
    }

    private void updateAutofillStatus() {
        AutofillManager mgr = getSystemService(AutofillManager.class);
        // Android 9.0 (API Level 28) 以降では Autofill Service のコンポーネント情報を取得できるようになった
        ComponentName componentName = mgr.getAutofillServiceComponentName();
        String componentNameString = "None";
        if (componentName == null) {
            mIsAutofillEnabled = false;//「設定」- 「自動入力サービス」が「なし」に設定されている
            Toast.makeText(this, "自動入力サービスなし", Toast.LENGTH_LONG).show();
```

(continues on next page)

(continued from previous page)

```
        } else {
            String autofillServicePackage = componentName.getPackageName();
            // 使おうとしている Autofill Service がホワイトリストに登録されているかを検査する
            if (checkService(this, autofillServicePackage)) {
                mIsAutofillEnabled = true;
                Toast.makeText(this, "信頼している自動入力サービス: " + autofillServicePackage, Toast.
↪LENGTH_LONG).show();
            } else {
                Toast.makeText(this, "信頼できない自動入力サービス: " + autofillServicePackage, Toast.
↪LENGTH_LONG).show();
                mIsAutofillEnabled = false;    //それ以外の場合は Autofill Service を利用しない
            }
            componentNameString = autofillServicePackage + " / " + componentName.getClassName();
        }

        TextView statusView = (TextView) findViewById(R.id.label_autofill_status);
        String status = "current autofill service: \n" + componentNameString;
        statusView.setText(status);
    }

}
```

PkgCertWhitelists.java

```
package org.jssec.android.shared;

import android.content.pm.PackageManager;
import java.util.HashMap;
import java.util.Map;
import android.content.Context;
import android.os.Build;

import static android.content.pm.PackageManager.CERT_INPUT_SHA256;

public class PkgCertWhitelists {
    private Map<String, String> mWhitelists = new HashMap<String, String>();

    public boolean add(String pkgname, String sha256) {
        if (pkgname == null) return false;
        if (sha256 == null) return false;

        sha256 = sha256.replaceAll(" ", "");
        if (sha256.length() != 64) return false;    // SHA-256 は 32 バイト
        sha256 = sha256.toUpperCase();
        if (sha256.replaceAll("[0-9A-F]+", "").length() != 0) return false; // 0-9A-F 以外の文字がある

        mWhitelists.put(pkgname, sha256);
        return true;
    }

    public boolean test(Context ctx, String pkgname) {
        // pkgname に対応する正解のハッシュ値を取得する
        String correctHash = mWhitelists.get(pkgname);
        android.util.Log.d("Partner", "hash=" + correctHash);
        // pkgname の実際のハッシュ値と正解のハッシュ値を比較する
```

(continues on next page)

(continued from previous page)

```
        if (Build.VERSION.SDK_INT >= 28) {
            // ★ API Level >= 28 では Package Manager の API で直接検証が可能
            PackageManager pm = ctx.getPackageManager();
            return pm.hasSigningCertificate(pkgname, hex2Bytes(correctHash), CERT_INPUT_SHA256);
        } else {
            // API Level < 28 の場合は PkgCert の機能を利用する
            return PkgCert.test(ctx, pkgname, correctHash);
        }
    }

    private byte[] hex2Bytes(String s) {
        int len = s.length();
        byte[] data = new byte[len / 2];
        for (int i = 0; i < len; i += 2) {
            data[i / 2] = (byte) ((Character.digit(s.charAt(i), 16) << 4)
                    + Character.digit(s.charAt(i+1), 16));
        }
        return data;
    }
}
```

PkgCert.java

```
package org.jssec.android.shared;

import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

import android.content.Context;
import android.content.pm.PackageInfo;
import android.content.pm.PackageManager;
import android.content.pm.PackageManager.NameNotFoundException;
import android.content.pm.Signature;

public class PkgCert {

    public static boolean test(Context ctx, String pkgname, String correctHash) {
        if (correctHash == null) return false;
        correctHash = correctHash.replaceAll(" ", "");
        return correctHash.equals(hash(ctx, pkgname));
    }

    public static String hash(Context ctx, String pkgname) {
        if (pkgname == null) return null;
        try {
            PackageManager pm = ctx.getPackageManager();
            PackageInfo pkginfo = pm.getPackageInfo(pkgname, PackageManager.GET_SIGNATURES);
            if (pkginfo.signatures.length != 1) return null;    // 複数署名は扱わない
            Signature sig = pkginfo.signatures[0];
            byte[] cert = sig.toByteArray();
            byte[] sha256 = computeSha256(cert);
            return byte2hex(sha256);
        } catch (NameNotFoundException e) {
            return null;
        }
```

(continues on next page)

(continued from previous page)

```
    }

    private static byte[] computeSha256(byte[] data) {
        try {
            return MessageDigest.getInstance("SHA-256").digest(data);
        } catch (NoSuchAlgorithmException e) {
            return null;
        }
    }

    private static String byte2hex(byte[] data) {
        if (data == null) return null;
        final StringBuilder hexadecimal = new StringBuilder();
        for (final byte b : data) {
            hexadecimal.append(String.format("%02X", b));
        }
        return hexadecimal.toString();
    }
}
```

4.2 Broadcast を受信する・送信する

4.2.1 サンプルコード

Broadcast を受信するには Broadcast Receiver を作る必要がある。どのような Broadcast を受信するかによって、Broadcast Receiver が抱えるリスクや適切な防御手段が異なる。次の判定フローによって作成する Broadcast Receiver がどのタイプであるかを判断できる。ちなみに、パートナー限定の連携に必要な Broadcast 送信元アプリのパッケージ名を受信側アプリで確認する手段がないため、パートナー限定 Broadcast Receiver を作る事はできない。

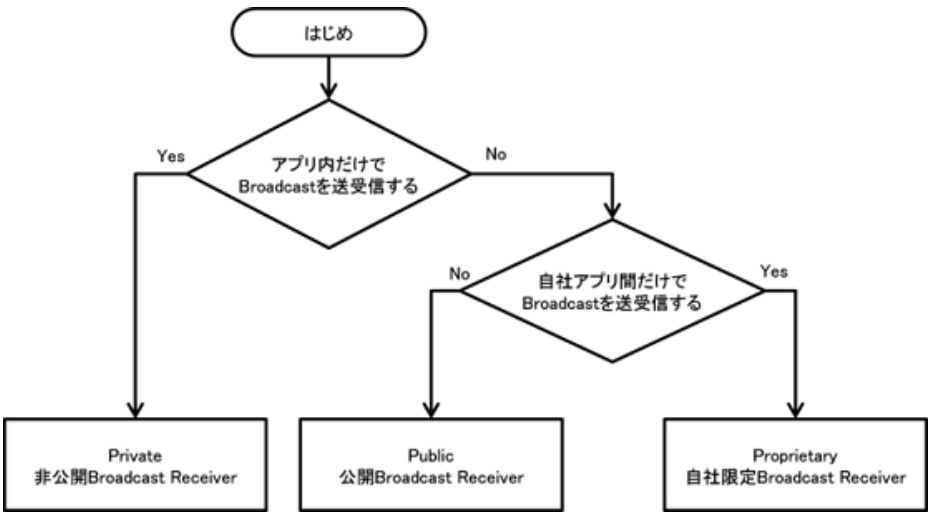

図 4.2.1　Broadcast Receiver のタイプ選択フロー

また Broadcast Receiver にはその定義方法により、静的 Broadcast Receiver と動的 Broadcast Receiver との 2 種類があり、下表のような特徴の違いがある。サンプルコード中では両方の実装方法を紹介している。なお、どのような相手に Broadcast を送信するかによって送信する情報の適切な防御手段が決まるため、送信側の実装についても合わせて説明する。

表 4.2.1: Broadcast Receiver の定義方法と特徴

	定義方法	特徴
静的 Broadcast Receiver	AndroidManifest.xml に<receiver>要素を記述することで定義する	• システムから送信される一部の Broadcast（ACTION_BATTERY_CHANGED など）を受信できない制約がある。 • アプリが初回起動してからアンインストールされるまでの間、Broadcast を受信できる。 • アプリの targetSDKVersion が 26 以上の場合、Android 8.0（API level 26) 以降の端末では、暗黙的 Broadcat Receiver を登録できない[*9]
動的 Broadcast Receiver	プログラム中で registerReceiver() および unregisterReceiver() 動的に Broadcast Receiver を登録／登録解除する	• 静的 Broadcast Receiver では受信できない Broadcast でも受信できる。 • Activity が前面に出ている期間だけ Broadcast を受信したいなど、Broadcast を受信したいなど、Broadcast の受信可能期間をプログラムで制御できる。 • 非公開の Broadcast Receiver を作ることはできない。

4.2.1.1 非公開 Broadcast Receiver - Broadcast を受信する・送信する

非公開 Broadcast Receiver は、同一アプリ内から送信された Broadcast だけを受信できる Broadcast Receiver であり、もっとも安全性の高い Broadcast Receiver である。動的 Broadcast Receiver を非公開で登録することはできないため、非公開 Broadcast Receiver では静的 Broadcast Receiver だけで構成される。

ポイント (Broadcast を受信する) :

1. exported=”false” により、明示的に非公開設定する
2. 同一アプリ内から送信された Broadcast であっても、受信 Intent の安全性を確認する
3. 結果を返す場合、送信元は同一アプリ内であるから、センシティブな情報を返送してよい

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="org.jssec.android.broadcast.privatereceiver" >


    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:allowBackup="false" >
```

(continues on next page)

[*9] システムが送信する暗黙的 Broadcast Intent の中には例外的に利用可能なものも存在する。詳細は以下の URL を参照のこと
https://developer.android.com/guide/components/broadcast-exceptions.html

(continued from previous page)

```
        <!-- 非公開 Broadcast Receiver を定義する -->
        <!-- ★ポイント 1★ exported="false"により、明示的に非公開設定する -->
        <receiver
            android:name=".PrivateReceiver"
            android:exported="false" />

        <activity
            android:name=".PrivateSenderActivity"
            android:label="@string/app_name"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

PrivateReceiver.java

```
package org.jssec.android.broadcast.privatereceiver;

import android.app.Activity;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.widget.Toast;

public class PrivateReceiver extends BroadcastReceiver {

    @Override
    public void onReceive(Context context, Intent intent) {

        // ★ポイント 2★ 同一アプリ内から送信された Broadcast であっても、受信 Intent の安全性を確認する
        // サンプルにつき割愛。「3.2 入力データの安全性を確認する」を参照。
        String param = intent.getStringExtra("PARAM");
        Toast.makeText(context,
                String.format("「%s」を受信した。", param),
                Toast.LENGTH_SHORT).show();

        // ★ポイント 3★ 送信元は同一アプリ内であるから、センシティブな情報を返送してよい
        setResultCode(Activity.RESULT_OK);
        setResultData("センシティブな情報 from Receiver");
        abortBroadcast();
    }
}
```

次に非公開 Broadcast Receiver へ Broadcast 送信するサンプルコードを示す。

ポイント (Broadcast を送信する)：

4. 同一アプリ内 Receiver はクラス指定の明示的 Intent で Broadcast 送信する
5. 送信先は同一アプリ内 Receiver であるため、センシティブな情報を送信してよい
6. 同一アプリ内 Receiver からの結果情報であっても、受信データの安全性を確認する

PrivateSenderActivity.java

```java
package org.jssec.android.broadcast.privatereceiver;

import android.app.Activity;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.widget.TextView;

public class PrivateSenderActivity extends Activity {

    public void onSendNormalClick(View view) {
        // ★ポイント 4★ 同一アプリ内 Receiver はクラス指定の明示的 Intent で Broadcast 送信する
        Intent intent = new Intent(this, PrivateReceiver.class);

        // ★ポイント 5★ 送信先は同一アプリ内 Receiver であるため、センシティブな情報を送信してよい
        intent.putExtra("PARAM", "センシティブな情報 from Sender");
        sendBroadcast(intent);
    }

    public void onSendOrderedClick(View view) {
        // ★ポイント 4★ 同一アプリ内 Receiver はクラス指定の明示的 Intent で Broadcast 送信する
        Intent intent = new Intent(this, PrivateReceiver.class);

        // ★ポイント 5★ 送信先は同一アプリ内 Receiver であるため、センシティブな情報を送信してよい
        intent.putExtra("PARAM", "センシティブな情報 from Sender");
        sendOrderedBroadcast(intent, null, mResultReceiver, null, 0, null, null);
    }

    private BroadcastReceiver mResultReceiver = new BroadcastReceiver() {
        @Override
        public void onReceive(Context context, Intent intent) {

            // ★ポイント 6★ 同一アプリ内 Receiver からの結果情報であっても、受信データの安全性を確認する
            // サンプルにつき割愛。「3.2 入力データの安全性を確認する」を参照。
            String data = getResultData();
            PrivateSenderActivity.this.logLine(
                    String.format("結果「%s」を受信した。", data));
        }
    };

    private TextView mLogView;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        mLogView = (TextView)findViewById(R.id.logview);
    }

    private void logLine(String line) {
        mLogView.append(line);
        mLogView.append("\n");
    }
```

(continues on next page)

(continued from previous page)

```
}
```

4.2.1.2 公開 Broadcast Receiver - Broadcast を受信する・送信する

公開 Broadcast Receiver は、不特定多数のアプリから送信された Broadcast を受信できる Broadcast Receiver である。マルウェアが送信した Broadcast を受信することがあることに注意が必要だ。

ポイント (Broadcast を受信する) :

1. exported=”true” により、明示的に公開設定する
2. 受信 Intent の安全性を確認する
3. 結果を返す場合、センシティブな情報を含めない

公開 Broadcast Receiver のサンプルコードである Public Receiver は、静的 Broadcast Receiver および動的 Broadcast Receiver の両方で利用される。

PublicReceiver.java

```
package org.jssec.android.broadcast.publicreceiver;

import android.app.Activity;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.widget.Toast;

public class PublicReceiver extends BroadcastReceiver {

    private static final String MY_BROADCAST_PUBLIC =
        "org.jssec.android.broadcast.MY_BROADCAST_PUBLIC";

    public boolean isDynamic = false;
    private String getName() {
        return isDynamic ? "公開動的 Broadcast Receiver" : "公開静的 Broadcast Receiver";
    }

    @Override
    public void onReceive(Context context, Intent intent) {

        // ★ポイント 2★ 受信 Intent の安全性を確認する
        // 公開 Broadcast Receiver であるため利用元アプリがマルウェアである可能性がある。
        // サンプルにつき割愛。「3.2 入力データの安全性を確認する」を参照。
        if (MY_BROADCAST_PUBLIC.equals(intent.getAction())) {
            String param = intent.getStringExtra("PARAM");
            Toast.makeText(context,
                    String.format("%s:\n「%s」を受信した。", getName(), param),
                    Toast.LENGTH_SHORT).show();
        }

        // ★ポイント 3★ 結果を返す場合、センシティブな情報を含めない
        // 公開 Broadcast Receiver であるため、
        // Broadcast の送信元アプリがマルウェアである可能性がある。
        // マルウェアに取得されても問題のない情報であれば結果として返してもよい。
```

(continues on next page)

(continued from previous page)

```
        setResultCode(Activity.RESULT_OK);
        setResultData(String.format("センシティブではない情報 from %s", getName()));
        abortBroadcast();
    }
}
```

静的 Broadcast Receiver は AndroidManifest.xml で定義する。表 4.2.1 のように、端末のバージョンによって暗黙的 Broadcst Intent の受信に制限があるので注意すること。

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="org.jssec.android.broadcast.publicreceiver" >


    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:allowBackup="false" >

        <!-- 公開静的 Broadcast Receiver を定義する -->
        <!-- ★ポイント 1★ exported="true"により、明示的に公開設定する -->
        <receiver android:name=".PublicReceiver"
                  android:exported="true" >
            <intent-filter>
                <action android:name="org.jssec.android.broadcast.MY_BROADCAST_PUBLIC" />
            </intent-filter>
        </receiver>

        <service
            android:name=".DynamicReceiverService"
            android:exported="false" />

        <activity
            android:name=".PublicReceiverActivity"
            android:label="@string/app_name"
            android:exported="true" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

動的 Broadcast Receiver はプログラム中で registerReceiver() および unregisterReceiver() を呼び出すことにより登録／登録解除する。ボタン操作により登録／登録解除を行うために PublicReceiverActivity 上にボタンを配置している。動的 Broadcast Receiver インスタンスは PublicReceiverActivity より生存期間が長いため PublicReceiverActivity のメンバー変数として保持することはできない。そのため DynamicReceiverService のメンバー変数として動的 Broadcast Receiver のインスタンスを保持させ、DynamicReceiverService を PublicReceiverActivity から開始／終了することにより動的 Broadcast Receiver を間接的に登録／登録解除している。

DynamicReceiverService.java

```
package org.jssec.android.broadcast.publicreceiver;

import android.app.Service;
import android.content.Intent;
import android.content.IntentFilter;
import android.os.IBinder;
import android.widget.Toast;

public class DynamicReceiverService extends Service {

    private static final String MY_BROADCAST_PUBLIC =
        "org.jssec.android.broadcast.MY_BROADCAST_PUBLIC";

    private PublicReceiver mReceiver;

    @Override
    public IBinder onBind(Intent intent) {
        return null;
    }

    @Override
    public void onCreate() {
        super.onCreate();

        // 公開動的 Broadcast Receiver を登録する
        mReceiver = new PublicReceiver();
        mReceiver.isDynamic = true;
        IntentFilter filter = new IntentFilter();
        filter.addAction(MY_BROADCAST_PUBLIC);
        filter.setPriority(1);  // 静的 Broadcast Receiver より動的 Broadcast Receiver を優先させる
        registerReceiver(mReceiver, filter);
        Toast.makeText(this,
                "動的 Broadcast Receiver を登録した。",
                Toast.LENGTH_SHORT).show();
    }

    @Override
    public void onDestroy() {
        super.onDestroy();

        // 公開動的 Broadcast Receiver を登録解除する
        unregisterReceiver(mReceiver);
        mReceiver = null;
        Toast.makeText(this,
                "動的 Broadcast Receiver を登録解除した。",
                Toast.LENGTH_SHORT).show();
    }
}
```

PublicReceiverActivity.java

```
package org.jssec.android.broadcast.publicreceiver;

import android.app.Activity;
import android.content.Intent;
```

(continues on next page)

(continued from previous page)

```
import android.os.Bundle;
import android.view.View;

public class PublicReceiverActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }

    public void onRegisterReceiverClick(View view) {
        Intent intent = new Intent(this, DynamicReceiverService.class);
        startService(intent);
    }

    public void onUnregisterReceiverClick(View view) {
        Intent intent = new Intent(this, DynamicReceiverService.class);
        stopService(intent);
    }
}
```

次に公開 Broadcast Receiver へ Broadcast 送信するサンプルコードを示す。公開 Broadcast Receiver に Broadcast を送信する場合、送信する Broadcast がマルウェアに受信されることがあることに注意が必要である。

ポイント (Broadcast を送信する) :

4. センシティブな情報を送信してはならない

5. 結果を受け取る場合、結果データの安全性を確認する

```
PublicSenderActivity.java
package org.jssec.android.broadcast.publicsender;

import android.app.Activity;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.widget.TextView;

public class PublicSenderActivity extends Activity {

    private static final String MY_BROADCAST_PUBLIC =
        "org.jssec.android.broadcast.MY_BROADCAST_PUBLIC";

    public void onSendNormalClick(View view) {
        // ★ポイント 4★ センシティブな情報を送信してはならない
        Intent intent = new Intent(MY_BROADCAST_PUBLIC);
        intent.putExtra("PARAM", "センシティブではない情報 from Sender");
        sendBroadcast(intent);
    }

    public void onSendOrderedClick(View view) {
```

(continues on next page)

(continued from previous page)

```
        // ★ポイント 4★ センシティブな情報を送信してはならない
        Intent intent = new Intent(MY_BROADCAST_PUBLIC);
        intent.putExtra("PARAM", "センシティブではない情報 from Sender");
        sendOrderedBroadcast(intent, null, mResultReceiver, null, 0, null, null);
    }

    public void onSendStickyClick(View view) {
        // ★ポイント 4★ センシティブな情報を送信してはならない
        Intent intent = new Intent(MY_BROADCAST_PUBLIC);
        intent.putExtra("PARAM", "センシティブではない情報 from Sender");
        //sendStickyBroadcast メソッドは API Level 21 で deprecated となった
        sendStickyBroadcast(intent);
    }

    public void onSendStickyOrderedClick(View view) {
        // ★ポイント 4★ センシティブな情報を送信してはならない
        Intent intent = new Intent(MY_BROADCAST_PUBLIC);
        intent.putExtra("PARAM", "センシティブではない情報 from Sender");
        //sendStickyBroadcast メソッドは API Level 21 で deprecated となった
        sendStickyOrderedBroadcast(intent, mResultReceiver, null, 0, null, null);
    }

    public void onRemoveStickyClick(View view) {
        Intent intent = new Intent(MY_BROADCAST_PUBLIC);
        //removeStickyBroadcast メソッドは deprecated となっている
        removeStickyBroadcast(intent);
    }

    private BroadcastReceiver mResultReceiver = new BroadcastReceiver() {
        @Override
        public void onReceive(Context context, Intent intent) {

            // ★ポイント 5★ 結果を受け取る場合、結果データの安全性を確認する
            // サンプルにつき割愛。「3.2 入力データの安全性を確認する」を参照。
            String data = getResultData();
            PublicSenderActivity.this.logLine(
                    String.format("結果「%s」を受信した。", data));
        }
    };

    private TextView mLogView;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        mLogView = (TextView)findViewById(R.id.logview);
    }

    private void logLine(String line) {
        mLogView.append(line);
        mLogView.append("\n");
    }
}
```

4.2.1.3 自社限定 **Broadcast Receiver - Broadcast** を受信する・送信する

自社限定 Broadcast Receiver は、自社以外のアプリから送信された Broadcast を一切受信しない Broadcast Receiver である。複数の自社製アプリでシステムを構成し、自社アプリが扱う情報や機能を守るために利用される。

ポイント (Broadcast を受信する) :

1. Broadcast 受信用の独自定義 Signature Permission を定義する
2. 結果受信用の独自定義 Signature Permission を利用宣言する
3. exported=”true” により、明示的に公開設定する
4. 静的 Broadcast Receiver 定義にて、独自定義 Signature Permission を要求宣言する
5. 動的 Broadcast Receiver を登録するとき、独自定義 Signature Permission を要求宣言する
6. 独自定義 Signature Permission が自社アプリにより定義されていることを確認する
7. 自社アプリからの Broadcast であっても、受信 Intent の安全性を確認する
8. Broadcast 送信元は自社アプリであるから、センシティブな情報を返送してよい
9. Broadcast 送信元アプリと同じ開発者鍵で APK を署名する

自社限定 Broadcast Receiver のサンプルコードである ProprietaryReciver は、静的 Broadcast Receiver および動的 Broadcast Receiver の両方で利用される。

InhouseReceiver.java

```
package org.jssec.android.broadcast.inhousereceiver;

import org.jssec.android.shared.SigPerm;
import org.jssec.android.shared.Utils;

import android.app.Activity;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.widget.Toast;

public class InhouseReceiver extends BroadcastReceiver {

    // 自社の Signature Permission
    private static final String MY_PERMISSION = "org.jssec.android.broadcast.inhousereceiver.MY_
↪PERMISSION";

    // 自社の証明書のハッシュ値
    private static String sMyCertHash = null;
    private static String myCertHash(Context context) {
        if (sMyCertHash == null) {
            if (Utils.isDebuggable(context)) {
                // debug.keystore の"androiddebugkey"の証明書ハッシュ値
                sMyCertHash = "0EFB7236 328348A9 89718BAD DF57F544 D5CCB4AE B9DB34BC 1E29DD26 F77C8255";
            } else {
                // keystore の"my company key"の証明書ハッシュ値
                sMyCertHash = "D397D343 A5CBC10F 4EDDEB7C A10062DE 5690984F 1FB9E88B D7B3A7C2 42E142CA";
            }
        }
```

(continues on next page)

(continued from previous page)

```
        return sMyCertHash;
    }

    private static final String MY_BROADCAST_INHOUSE =
        "org.jssec.android.broadcast.MY_BROADCAST_INHOUSE";

    public boolean isDynamic = false;
    private String getName() {
        return isDynamic ? "自社限定動的 Broadcast Receiver" : "自社限定静的 Broadcast Receiver";
    }

    @Override
    public void onReceive(Context context, Intent intent) {

        // ★ポイント 6★ 独自定義 Signature Permission が自社アプリにより定義されていることを確認する
        if (!SigPerm.test(context, MY_PERMISSION, myCertHash(context))) {
            Toast.makeText(context, "独自定義 Signature Permission が自社アプリにより定義されていない。",␣
↪Toast.LENGTH_LONG).show();
            return;
        }

        // ★ポイント 7★ 自社アプリからの Broadcast であっても、受信 Intent の安全性を確認する
        // サンプルにつき割愛。「3.2 入力データの安全性を確認する」を参照。
        if (MY_BROADCAST_INHOUSE.equals(intent.getAction())) {
            String param = intent.getStringExtra("PARAM");
            Toast.makeText(context,
                    String.format("%s:\n「%s」を受信した。", getName(), param),
                    Toast.LENGTH_SHORT).show();
        }

        // ★ポイント 8★ 送信元は自社アプリであるから、センシティブな情報を返送してよい
        setResultCode(Activity.RESULT_OK);
        setResultData(String.format("センシティブな情報 from %s", getName()));
        abortBroadcast();
    }
}
```

静的 Broadcast Receiver は AndroidManifest.xml で定義する。表 4.2.1 のように、端末のバージョンによって暗黙的 Broadcst Intent の受信に制限があるので注意すること。

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="org.jssec.android.broadcast.inhousereceiver" >

    <!-- ★ポイント 1★ Broadcast 受信用の独自定義 Signature Permission を定義する -->
    <permission
        android:name="org.jssec.android.broadcast.inhousereceiver.MY_PERMISSION"
        android:protectionLevel="signature" />

    <!-- ★ポイント 2★ 結果受信用の独自定義 Signature Permission を利用宣言する -->
    <uses-permission
        android:name="org.jssec.android.broadcast.inhousesender.MY_PERMISSION" />
```

(continues on next page)

(continued from previous page)

```
    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:allowBackup="false">

        <!-- ★ポイント 3★ exported="true"により、明示的に公開設定する -->
        <!-- ★ポイント 4★ 静的 Broadcast Receiver 定義にて、独自定義 Signature Permission を要求宣言する -->
        <receiver
            android:name="org.jssec.android.broadcast.inhousereceiver.InhouseReceiver"
            android:exported="true"
            android:permission="org.jssec.android.broadcast.inhousereceiver.MY_PERMISSION">
            <intent-filter>
                <action android:name="org.jssec.android.broadcast.MY_BROADCAST_INHOUSE" />
            </intent-filter>
        </receiver>

        <service
            android:name="org.jssec.android.broadcast.inhousereceiver.DynamicReceiverService"
            android:exported="false" />

        <activity
            android:name="org.jssec.android.broadcast.inhousereceiver.InhouseReceiverActivity"
            android:label="@string/app_name"
            android:exported="true" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

動的 Broadcast Receiver はプログラム中で registerReceiver() および unregisterReceiver() を呼び出すことにより登録／登録解除する。ボタン操作により登録／登録解除を行うために ProprietaryReceiverActivity 上にボタンを配置している。動的 Broadcast Receiver インスタンスは ProprietaryReceiverActivity より生存期間が長いため ProprietaryReceiverActivity のメンバー変数として保持することはできない。そのため DynamicReceiverService のメンバー変数として動的 Broadcast Receiver のインスタンスを保持させ、DynamicReceiverService を ProprietaryReceiverActivity から開始／終了することにより動的 Broadcast Receiver を間接的に登録／登録解除している。

InhouseReceiverActivity.java

```
package org.jssec.android.broadcast.inhousereceiver;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;

public class InhouseReceiverActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
```

(continues on next page)

(continued from previous page)

```
    }

    public void onRegisterReceiverClick(View view) {
        Intent intent = new Intent(this, DynamicReceiverService.class);
        startService(intent);
    }

    public void onUnregisterReceiverClick(View view) {
        Intent intent = new Intent(this, DynamicReceiverService.class);
        stopService(intent);
    }
}
```

DynamicReceiverService.java

```
package org.jssec.android.broadcast.inhousereceiver;

import android.app.Service;
import android.content.Intent;
import android.content.IntentFilter;
import android.os.IBinder;
import android.widget.Toast;

public class DynamicReceiverService extends Service {

    private static final String MY_BROADCAST_INHOUSE =
        "org.jssec.android.broadcast.MY_BROADCAST_INHOUSE";

    private InhouseReceiver mReceiver;

    @Override
    public IBinder onBind(Intent intent) {
        return null;
    }

    @Override
    public void onCreate() {
        super.onCreate();

        mReceiver = new InhouseReceiver();
        mReceiver.isDynamic = true;
        IntentFilter filter = new IntentFilter();
        filter.addAction(MY_BROADCAST_INHOUSE);
        filter.setPriority(1);  // 静的 Broadcast Receiver より動的 Broadcast Receiver を優先させる

        // ★ポイント 5★ 動的 Broadcast Receiver を登録するとき、独自定義 Signature Permission を要求宣言する
        registerReceiver(mReceiver, filter, "org.jssec.android.broadcast.inhousereceiver.MY_PERMISSION",␣
↪null);

        Toast.makeText(this,
                "動的 Broadcast Receiver を登録した。",
                Toast.LENGTH_SHORT).show();
    }

    @Override
```

(continues on next page)

(continued from previous page)

```
    public void onDestroy() {
        super.onDestroy();
        unregisterReceiver(mReceiver);
        mReceiver = null;
        Toast.makeText(this,
                "動的 Broadcast Receiver を登録解除した。",
                Toast.LENGTH_SHORT).show();
    }
}
```

SigPerm.java

```
package org.jssec.android.shared;

import android.content.Context;
import android.content.pm.PackageManager;
import android.content.pm.PackageManager.NameNotFoundException;
import android.content.pm.PermissionInfo;
import android.os.Build;

import static android.content.pm.PackageManager.CERT_INPUT_SHA256;

public class SigPerm {

    public static boolean test(Context ctx, String sigPermName, String correctHash) {
        if (correctHash == null) return false;
        correctHash = correctHash.replaceAll(" ", "");
        try{
            // sigPermName を定義したアプリのパッケージ名を取得する
            PackageManager pm = ctx.getPackageManager();
            PermissionInfo pi = pm.getPermissionInfo(sigPermName, PackageManager.GET_META_DATA);
            String pkgname = pi.packageName;
            // 非 Signature Permission の場合は失敗扱い
            if (pi.protectionLevel != PermissionInfo.PROTECTION_SIGNATURE) return false;
            // pkgname の実際のハッシュ値と正解のハッシュ値を比較する
            if (Build.VERSION.SDK_INT >= 28) {
                // ★ API Level >= 28 では Package Manager の API で直接検証が可能
                return pm.hasSigningCertificate(pkgname, Utils.hex2Bytes(correctHash), CERT_INPUT_
↪SHA256);
            } else {
                // API Level < 28 の場合は PkgCert を利用し、ハッシュ値を取得して比較する
                return correctHash.equals(PkgCert.hash(ctx, pkgname));
            }
        } catch (NameNotFoundException e){
            return false;
        }
    }
}
```

PkgCert.java

```
package org.jssec.android.shared;

import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
```

(continues on next page)

(continued from previous page)

```
import android.content.Context;
import android.content.pm.PackageInfo;
import android.content.pm.PackageManager;
import android.content.pm.PackageManager.NameNotFoundException;
import android.content.pm.Signature;

public class PkgCert {

    public static boolean test(Context ctx, String pkgname, String correctHash) {
        if (correctHash == null) return false;
        correctHash = correctHash.replaceAll(" ", "");
        return correctHash.equals(hash(ctx, pkgname));
    }

    public static String hash(Context ctx, String pkgname) {
        if (pkgname == null) return null;
        try {
            PackageManager pm = ctx.getPackageManager();
            PackageInfo pkginfo = pm.getPackageInfo(pkgname, PackageManager.GET_SIGNATURES);
            if (pkginfo.signatures.length != 1) return null;    // 複数署名は扱わない
            Signature sig = pkginfo.signatures[0];
            byte[] cert = sig.toByteArray();
            byte[] sha256 = computeSha256(cert);
            return byte2hex(sha256);
        } catch (NameNotFoundException e) {
            return null;
        }
    }

    private static byte[] computeSha256(byte[] data) {
        try {
            return MessageDigest.getInstance("SHA-256").digest(data);
        } catch (NoSuchAlgorithmException e) {
            return null;
        }
    }

    private static String byte2hex(byte[] data) {
        if (data == null) return null;
        final StringBuilder hexadecimal = new StringBuilder();
        for (final byte b : data) {
            hexadecimal.append(String.format("%02X", b));
        }
        return hexadecimal.toString();
    }
}
```

★ポイント 9 ★ APK を Export するときに、Broadcast 送信元アプリと同じ開発者鍵で APK を署名する。

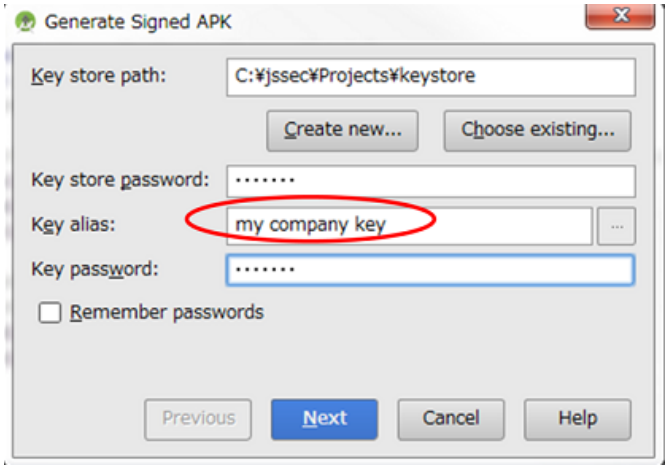

図 4.2.2　Broadcast 送信元アプリと同じ開発者鍵で APK を署名する

次に自社限定 Broadcast Receiver へ Broadcast 送信するサンプルコードを示す。

ポイント (Broadcast を送信する)：

10. 結果受信用の独自定義 Signature Permission を定義する
11. Broadcast 受信用の独自定義 Signature Permission を利用宣言する
12. 独自定義 Signature Permission が自社アプリにより定義されていることを確認する
13. Receiver は自社アプリ限定であるから、センシティブな情報を送信してもよい
14. Receiver に独自定義 Signature Permission を要求する
15. 結果を受け取る場合、結果データの安全性を確認する
16. Receiver 側アプリと同じ開発者鍵で APK を署名する

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="org.jssec.android.broadcast.inhousesender" >

    <uses-permission android:name="android.permission.BROADCAST_STICKY"/>

    <!-- ★ポイント 10★ 結果受信用の独自定義 Signature Permission を定義する -->
    <permission
        android:name="org.jssec.android.broadcast.inhousesender.MY_PERMISSION"
        android:protectionLevel="signature" />

    <!-- ★ポイント 11★ Broadcast 受信用の独自定義 Signature Permission を利用宣言する -->
    <uses-permission
        android:name="org.jssec.android.broadcast.inhousereceiver.MY_PERMISSION" />

    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:allowBackup="false" >
        <activity
            android:name="org.jssec.android.broadcast.inhousesender.InhouseSenderActivity"
```

(continues on next page)

(continued from previous page)

```
            android:label="@string/app_name"
            android:exported="true" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

InhouseSenderActivity.java

```
package org.jssec.android.broadcast.inhousesender;

import org.jssec.android.shared.SigPerm;
import org.jssec.android.shared.Utils;

import android.app.Activity;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.widget.TextView;
import android.widget.Toast;

public class InhouseSenderActivity extends Activity {

    // 自社の Signature Permission
    private static final String MY_PERMISSION = "org.jssec.android.broadcast.inhousesender.MY_PERMISSION
↪";

    // 自社の証明書のハッシュ値
    private static String sMyCertHash = null;
    private static String myCertHash(Context context) {
        if (sMyCertHash == null) {
            if (Utils.isDebuggable(context)) {
                // debug.keystore の"androiddebugkey"の証明書ハッシュ値
                sMyCertHash = "0EFB7236 328348A9 89718BAD DF57F544 D5CCB4AE B9DB34BC 1E29DD26 F77C8255";
            } else {
                // keystore の"my company key"の証明書ハッシュ値
                sMyCertHash = "D397D343 A5CBC10F 4EDDEB7C A10062DE 5690984F 1FB9E88B D7B3A7C2 42E142CA";
            }
        }
        return sMyCertHash;
    }

    private static final String MY_BROADCAST_INHOUSE =
        "org.jssec.android.broadcast.MY_BROADCAST_INHOUSE";

    public void onSendNormalClick(View view) {

        // ★ポイント 12★ 独自定義 Signature Permission が自社アプリにより定義されていることを確認する
        if (!SigPerm.test(this, MY_PERMISSION, myCertHash(this))) {
            Toast.makeText(this, "独自定義 Signature Permission が自社アプリにより定義されていない。", Toast.
↪LENGTH_LONG).show();
```

(continues on next page)

(continued from previous page)

```
            return;
        }

        // ★ポイント 13★ Receiver は自社アプリ限定であるから、センシティブな情報を送信してもよい
        Intent intent = new Intent(MY_BROADCAST_INHOUSE);
        intent.putExtra("PARAM", "センシティブな情報 from Sender");


        // ★ポイント 14★ Receiver に独自定義 Signature Permission を要求する
        sendBroadcast(intent, "org.jssec.android.broadcast.inhousesender.MY_PERMISSION");
    }

    public void onSendOrderedClick(View view) {

        // ★ポイント 12★ 独自定義 Signature Permission が自社アプリにより定義されていることを確認する
        if (!SigPerm.test(this, MY_PERMISSION, myCertHash(this))) {
            Toast.makeText(this, "独自定義 Signature Permission が自社アプリにより定義されていない。", Toast.
↪LENGTH_LONG).show();
            return;
        }

        // ★ポイント 13★ Receiver は自社アプリ限定であるから、センシティブな情報を送信してもよい
        Intent intent = new Intent(MY_BROADCAST_INHOUSE);
        intent.putExtra("PARAM", "センシティブな情報 from Sender");

        // ★ポイント 14★ Receiver に独自定義 Signature Permission を要求する
        sendOrderedBroadcast(intent, "org.jssec.android.broadcast.inhousesender.MY_PERMISSION",
                mResultReceiver, null, 0, null, null);
    }

    private BroadcastReceiver mResultReceiver = new BroadcastReceiver() {
        @Override
        public void onReceive(Context context, Intent intent) {

            // ★ポイント 15★ 結果を受け取る場合、結果データの安全性を確認する
            // サンプルにつき割愛。「3.2 入力データの安全性を確認する」を参照。
            String data = getResultData();
            InhouseSenderActivity.this.logLine(String.format("結果「%s」を受信した。", data));
        }
    };

    private TextView mLogView;
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        mLogView = (TextView)findViewById(R.id.logview);
    }

    private void logLine(String line) {
        mLogView.append(line);
        mLogView.append("\n");
    }
}
```

SigPerm.java

```
package org.jssec.android.shared;

import android.content.Context;
import android.content.pm.PackageManager;
import android.content.pm.PackageManager.NameNotFoundException;
import android.content.pm.PermissionInfo;
import android.os.Build;

import static android.content.pm.PackageManager.CERT_INPUT_SHA256;

public class SigPerm {

    public static boolean test(Context ctx, String sigPermName, String correctHash) {
        if (correctHash == null) return false;
        correctHash = correctHash.replaceAll(" ", "");
        try{
            // sigPermName を定義したアプリのパッケージ名を取得する
            PackageManager pm = ctx.getPackageManager();
            PermissionInfo pi = pm.getPermissionInfo(sigPermName, PackageManager.GET_META_DATA);
            String pkgname = pi.packageName;
            // 非 Signature Permission の場合は失敗扱い
            if (pi.protectionLevel != PermissionInfo.PROTECTION_SIGNATURE) return false;
            // pkgname の実際のハッシュ値と正解のハッシュ値を比較する
            if (Build.VERSION.SDK_INT >= 28) {
                // ★ API Level >= 28 では Package Manager の API で直接検証が可能
                return pm.hasSigningCertificate(pkgname, Utils.hex2Bytes(correctHash), CERT_INPUT_
↪SHA256);
            } else {
                // API Level < 28 の場合は PkgCert を利用し、ハッシュ値を取得して比較する
                return correctHash.equals(PkgCert.hash(ctx, pkgname));
            }
        } catch (NameNotFoundException e){
            return false;
        }
    }
}
```

PkgCert.java

```
package org.jssec.android.shared;

import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

import android.content.Context;
import android.content.pm.PackageInfo;
import android.content.pm.PackageManager;
import android.content.pm.PackageManager.NameNotFoundException;
import android.content.pm.Signature;

public class PkgCert {

    public static boolean test(Context ctx, String pkgname, String correctHash) {
        if (correctHash == null) return false;
        correctHash = correctHash.replaceAll(" ", "");
```

(continues on next page)

(continued from previous page)

```
        return correctHash.equals(hash(ctx, pkgname));
    }

    public static String hash(Context ctx, String pkgname) {
        if (pkgname == null) return null;
        try {
            PackageManager pm = ctx.getPackageManager();
            PackageInfo pkginfo = pm.getPackageInfo(pkgname, PackageManager.GET_SIGNATURES);
            if (pkginfo.signatures.length != 1) return null;    // 複数署名は扱わない
            Signature sig = pkginfo.signatures[0];
            byte[] cert = sig.toByteArray();
            byte[] sha256 = computeSha256(cert);
            return byte2hex(sha256);
        } catch (NameNotFoundException e) {
            return null;
        }
    }

    private static byte[] computeSha256(byte[] data) {
        try {
            return MessageDigest.getInstance("SHA-256").digest(data);
        } catch (NoSuchAlgorithmException e) {
            return null;
        }
    }

    private static String byte2hex(byte[] data) {
        if (data == null) return null;
        final StringBuilder hexadecimal = new StringBuilder();
        for (final byte b : data) {
            hexadecimal.append(String.format("%02X", b));
        }
        return hexadecimal.toString();
    }
}
```

★ポイント 16 ★ APK を Export するときに、Receiver 側アプリと同じ開発者鍵で APK を署名する。

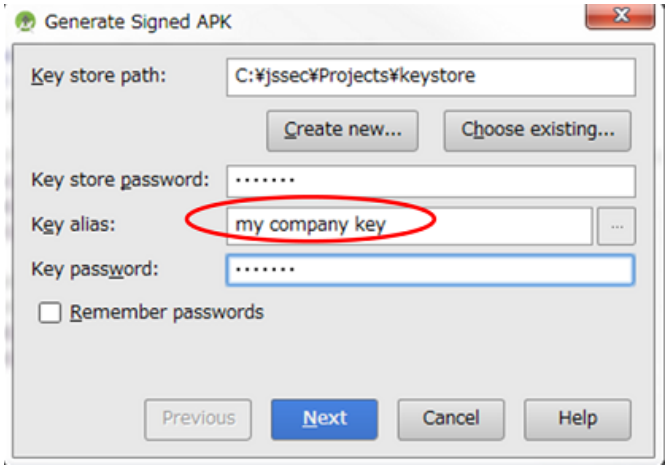

図 4.2.3　Reciver 側アプリと同じ開発者鍵で APK を署名する

4.2.2 ルールブック

Broadcast を送受信する際には以下のルールを守ること。

1. アプリ内でのみ使用する *Broadcast Receiver* は非公開設定する （必須）
2. 受信 *Intent* の安全性を確認する （必須）
3. 独自定義 *Signature Permission* は、自社アプリが定義したことを確認して利用する （必須）
4. 結果情報を返す場合には、返送先アプリからの結果情報漏洩に注意する （必須）
5. センシティブな情報を *Broadcast* 送信する場合は、受信可能な *Receiver* を制限する （必須）
6. *Sticky Broadcast* にはセンシティブな情報を含めない （必須）
7. *receiverPermission* パラメータの指定なし *Ordered Broadcast* は届かないことがあることに注意 （必須）
8. *Broadcast Receiver* からの返信データの安全性を確認する （必須）
9. 資産を二次的に提供する場合には、その資産の従来の保護水準を維持する （必須）

4.2.2.1 アプリ内でのみ使用する **Broadcast Receiver** は非公開設定する （必須）

アプリ内でのみ使用される Broadcast Receiver は非公開設定する。これにより、他のアプリから意図せず Broadcast を受け取ってしまうことがなくなり、アプリの機能を利用されたり、アプリの動作に異常をきたしたりするのを防ぐことができる。

同一アプリ内からのみ利用される Receiver では Intent Filter を設置するような設計はしてはならない。Intent Filter の性質上、同一アプリ内の非公開 Receiver を呼び出すつもりでも、Intent Filter 経由で呼び出したときに意図せず他アプリの公開 Receiver を呼び出してしまう場合が存在するからである。

```
AndroidManifest.xml(非推奨)
    <!-- 外部アプリに非公開とする Broadcast Receiver -->
    <!-- ポイント 1: exported="false" とする -->
    <receiver android:name=".PrivateReceiver"
        android:exported="false" >
        <intent-filter>
            <action android:name="org.jssec.android.broadcast.MY_ACTION" />
        </intent-filter>
    </receiver>
```

「4.2.3.1. 使用してよい *exported* 設定と *intent-filter* 設定の組み合わせ *(Receiver* の場合*)*」も参照すること。

4.2.2.2 受信 **Intent** の安全性を確認する （必須）

Broadcast Receiver のタイプによって若干リスクは異なるが、受信 Intent のデータを処理する際には、まず受信 Intent の安全性を確認しなければならない。

公開 Broadcast Receiver は不特定多数のアプリから Intent を受け取るため、マルウェアの攻撃 Intent を受け取る可能性がある。非公開 Broadcast Receiver は他のアプリから Intent を直接受け取ることはない。しかし同一アプリ内の公開 Component が他のアプリから受け取った Intent のデータを非公開 Broadcast Receiver に転送することがあるため、受信 Intent を無条件に安全であると考えてはならない。自社限定 Broadcast Receiver はその中間のリスクであるため、やはり受信 Intent の安全性を確認する必要がある。

「3.2. 入力データの安全性を確認する」を参照すること。

4.2.2.3 独自定義 **Signature Permission** は、自社アプリが定義したことを確認して利用する （必須）

自社のアプリから送信された Broadcast だけを受信し、それ以外の Broadcast を一切受信しない自社限定 Broadcast Receiver を作る場合、独自定義 Signature Permission により保護しなければならない。AndroidManifest.xml での Permission 定義、Permission 要求宣言だけでは保護が不十分であるため、「5.2. *Permission* と *Protection Level*」の「5.2.1.2. 独自定義の *Signature Permission* で自社アプリ連携する方法」を参照すること。また独自定義 Signature Permission を receiverPermission パラメータに指定して Broadcast 送信する場合も同様に確認する必要がある。

4.2.2.4 結果情報を返す場合には、返送先アプリからの結果情報漏洩に注意する （必須）

Broadcast Receiver のタイプによって setResult() により結果情報を返すアプリの信用度が異なる。公開 Broadcast Receiver の場合は、結果返送先のアプリがマルウェアである可能性もあり、結果情報が悪意を持って使われる危険性がある。非公開 Broadcast Receiver や自社限定 Broadcast Receiver の場合は、結果返送先は自社開発アプリであるため結果情報の扱いをあまり心配する必要はない。

このように Broadcast Receiver から結果情報を返す場合には、返送先アプリからの結果情報の漏洩に配慮しなければならない。

4.2.2.5 センシティブな情報を **Broadcast** 送信する場合は、受信可能な **Receiver** を制限する （必須）

Broadcast という名前が表すように、そもそも Broadcast は不特定多数のアプリに情報を一斉送信したり、タイミングを通知したりすることを意図して作られた仕組みである。そのためセンシティブな情報を Broadcast 送信する場合には、マルウェアに情報を取得されないような注意深い設計が必要となる。

センシティブな情報を Broadcast 送信する場合、信頼できる Broadcast Receiver だけが受信可能であり、他の Broadcast Receiver は受信不可能である必要がある。そのための Broadcast 送信方法には以下のようなものがある。

- 明示的 Intent で Broadcast 送信することで宛先を固定し、意図した信頼できる Broadcast Receiver だけに Broadcast を届ける方法。この方法には次の 2 つのパターンがある。
  - 同一アプリ内 Broadcast Receiver 宛てであれば Intent#setClass(Context, Class) により宛先を限定する。具体的なコードについてはサンプルコードセクション「4.2.1.1. 非公開 *Broadcast Receiver - Broadcast* を受信する・送信する」を参考にすること。
  - 他のアプリの Broadcast Receiver 宛てであれば Intent#setClassName(String, String) により宛先を限定するが、Broadcast 送信に先立ち宛先パッケージの APK 署名の開発者鍵をホワイトリストと照合して許可したアプリであることを確認してから Broadcast を送信する。実際には暗黙的 Intent を利用できる次の方法が実用的である。
- receiverPermission パラメータに独自定義 Signature Permission を指定して Broadcast 送信し、信頼する Broadcast Receiver に当該 Signature Permission を利用宣言してもらう方法。具体的なコードについてはサンプルコードセクション「4.2.1.3. 自社限定 *Broadcast Receiver - Broadcast* を受信する・送信する」を参考にすること。またこの Broadcast 送信方法を実装するにはルール「4.2.2.3. 独自定義 *Signature Permission* は、自社アプリが定義したことを確認して利用する （必須）」も適用しなければならない。

4.2.2.6 **Sticky Broadcast** にはセンシティブな情報を含めない （必須）

通常の Broadcast は、Broadcast 送信時に受信可能状態にある Broadcast Receiver に受信処理されると、その Broadcast は消滅してしまう。一方 Sticky Broadcast（および Sticky Ordered Broadcast、以下同様）は、送信時に受信状態にある Broadcast Receiver に受信処理された後もシステム上に存在しつづけ、その後 registerReceiver() により受信できることが特徴である。不要になった Sticky Broadcast は removeStickyBroadcast() により任意のタイミングで削除できる。

Sticky Broadcast は暗黙的 Intent による使用が前提であり、receiverPermission パラメータを指定した Broadcast 送信はできない。そのため Sticky Broadcast で送信した情報はマルウェアを含む不特定多数のアプリから取得できてしまう。したがってセンシティブな情報を Sticky Broadcast で送信してはならない。なお、Sticky Broadcast の使用は Android 5.0（API Level 21）において非推奨となっている。

4.2.2.7 **receiverPermission** パラメータの指定なし **Ordered Broadcast** は届かないことがあることに注意 （必須）

receiverPermission パラメータを指定せずに送信された Ordered Broadcast は、マルウェアを含む不特定多数のアプリが受信可能である。Ordered Broadcast は Receiver からの返り情報を受け取るため、または複数の Receiver に順次処理をさせるために利用される。優先度の高い Receiver から順次 Broadcast が配送されるため、優先度を高くしたマルウェアが最初に Broadcast を受信し abortBroadcast() すると、後続の Receiver に Broadcast が配信されなくなる。

4.2.2.8 **Broadcast Receiver** からの返信データの安全性を確認する （必須）

結果データを送り返してきた Broadcast Receiver のタイプによって若干リスクは異なるが、基本的には受信した結果データが攻撃データである可能性を考慮して安全に処理しなければならない。

返信元 Broadcast Receiver が公開 Broadcast Receiver の場合、不特定のアプリから戻りデータを受け取るため、マルウェアの攻撃データを受け取る可能性がある。返信元 Broadcast Receiver が非公開 Broadcast Receiver の場合、同一アプリ内からの結果データであるのでリスクはないように考えがちだが、他のアプリから受け取ったデータを間接的に結果データとして転送することがあるため、結果データを無条件に安全であると考えてはならない。返信元 Broadcast Receiver が自社限定 Broadcast Receiver の場合、その中間のリスクであるため、やはり結果データが攻撃データである可能性を考慮して安全に処理しなければならない。

「3.2. 入力データの安全性を確認する」を参照すること。

4.2.2.9 資産を二次的に提供する場合には、その資産の従来の保護水準を維持する （必須）

Permission により保護されている情報資産および機能資産を他のアプリに二次的に提供する場合には、提供先アプリに対して同一の Permission を要求するなどして、その保護水準を維持しなければならない。Android の Permission セキュリティモデルでは、保護された資産に対するアプリからの直接アクセスについてのみ権限管理を行う。この仕様上の特性により、アプリに取得された資産がさらに他のアプリに、保護のために必要な Permission を要求することなく提供される可能性がある。このことは Permission を再委譲していることと実質的に等価なので、Permission の再委譲問題と呼ばれる。「5.2.3.4. *Permission* の再委譲問題」を参照すること。

4.2.3 アドバンスト

4.2.3.1 使用してよい **exported** 設定と **intent-filter** 設定の組み合わせ **(Receiver** の場合**)**

表 4.2.2 は、Receiver を実装するときに使用してもよい exported 属性と intent-filter 要素の組み合わせを示している。「exported=”false” かつ intent-filter 定義あり」の使用を原則禁止としている理由については以下で説明する。

表 4.2.2　exported 属性と intent-filter 要素の組み合わせの使用可否

	exported 属性の値		
	true	**false**	無指定
intent-filter 定義がある	可	原則禁止	禁止
intent-filter 定義がない	可	可	禁止

Receiver の exported 属性が無指定である場合にその Receiver が公開されるか非公開となるかは、intent-filter の定義の有無により決まるが[*10]、本ガイドでは Receiver の exported 属性を「無指定」にすることを禁止している。前述のような API のデフォルトの挙動に頼る実装をすることは避けるべきであり、exported 属性のようなセキュリティ上重要な設定を明示的に有効化する手段があるのであればそれを利用すべきであると考えられるためである。

intent-filter を定義し、かつ、exported=”false” を指定することを原則禁止としているのは、同一アプリ内の非公開 Receiver に向けて Broadcast を送信したつもりでも、意図せず他アプリの公開 Receiver を呼び出してしまう場合が存在するからである。以下の 2 つの図で意図せぬ呼び出しが起こる様子を説明する。

図 4.2.4 は、同一アプリ内からしか非公開 Receiver(アプリ A）を暗黙的 Intent で呼び出せない正常な動作の例である。Intent-filter(図中 action=”X”) を定義しているのが、アプリ A しかいないので意図通りの動きとなっている。

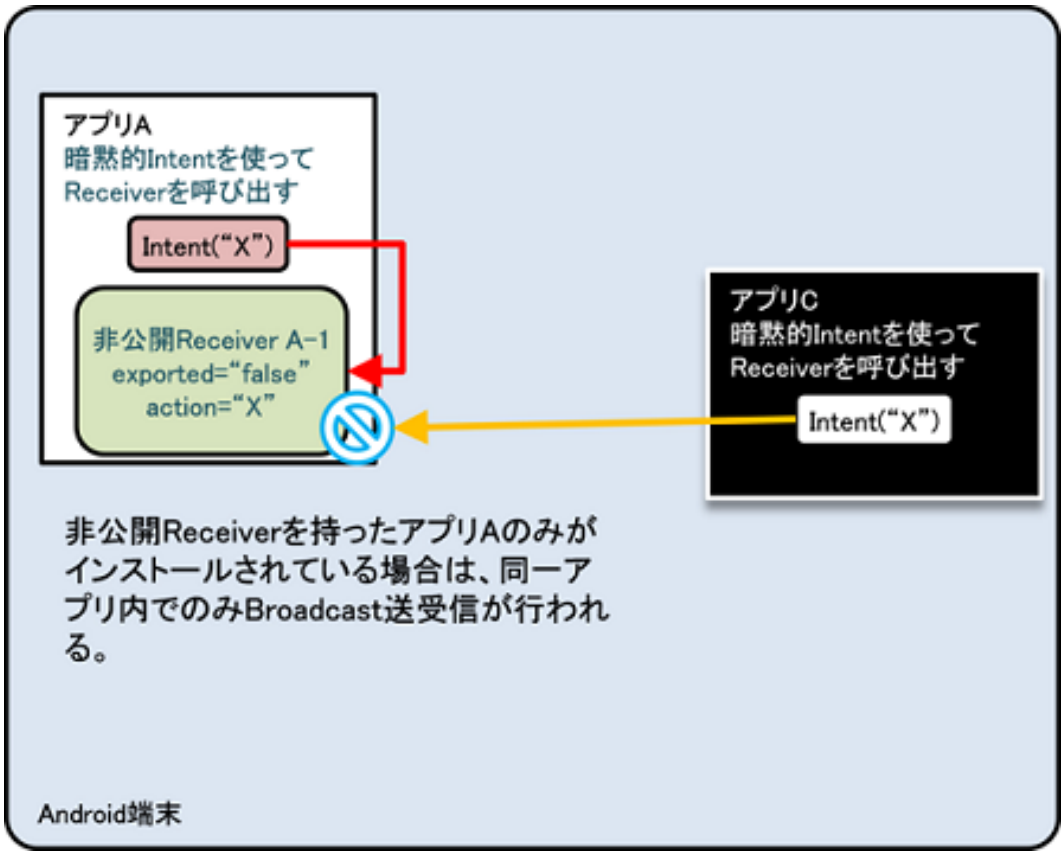

図 4.2.4　正常な動作の例

図 4.2.5 は、アプリ A に加えてアプリ B でも同じ intent-filter(図中 action=”X”) を定義している場合である。まず、他のアプリ (アプリ C) が暗黙的 Intent で Broadcast を送信するのは、非公開 Receiver(A-1) 側は受信をしないので特にセキュリティ的には問題にならない (図の橙色の矢印)。

[*10] intent-filter が定義されていれば公開 Receiver、定義されていなければ非公開 Receiver となる。https://developer.android.com/guide/topics/manifest/receiver-element.html#exported を参照のこと。

セキュリティ面で問題になるのは、アプリ A による同一アプリ内の非公開 Receiver の呼び出しである。アプリ A が暗黙的 Intent を Broadcast すると、同一アプリ内の非公開 Receiver に加えて、同じ Intent-filter を定義した B の持つ公開 Receiver(B-1) もその Intent を受信できてしまうからである (図の赤色の矢印)。アプリ A からアプリ B に対してセンシティブな情報を送信する可能性が生じてしまう。アプリ B がマルウェアであれば、そのままセンシティブな情報の漏洩に繋がる。また、Broadcast が Ordered Broadcast であった場合は、意図しない結果情報を受け取ってしまう可能性もある。

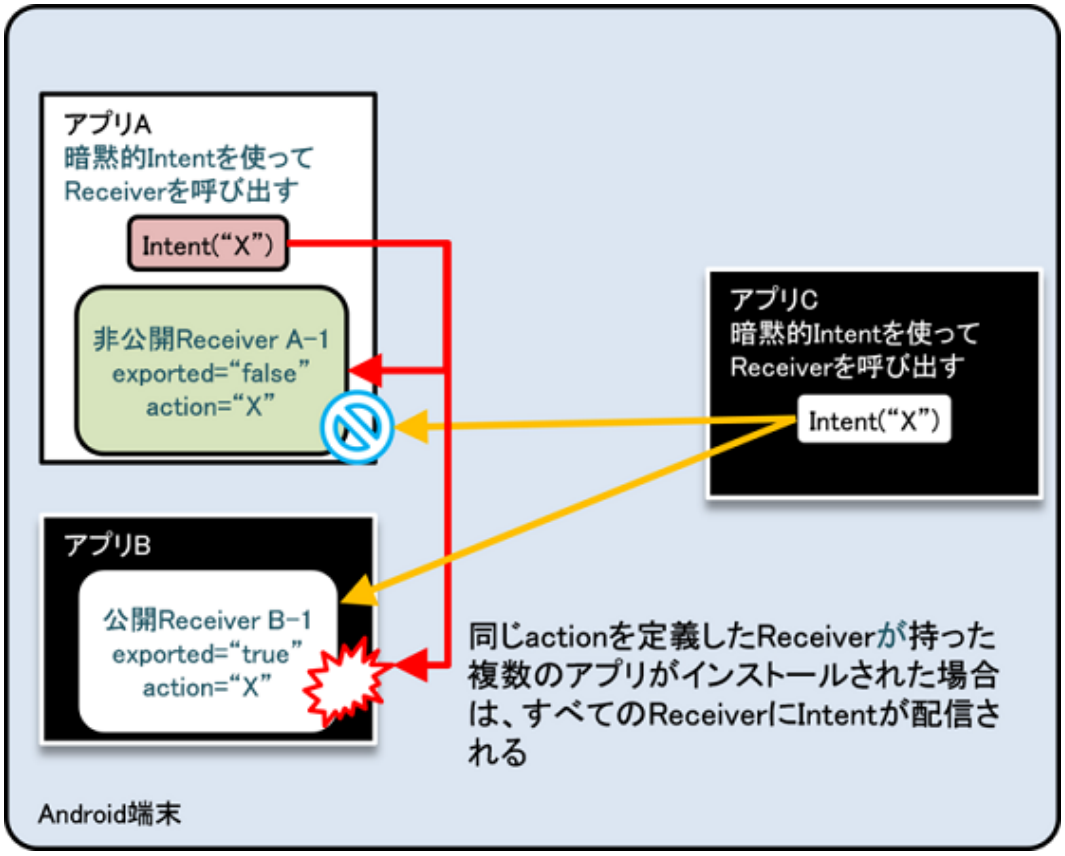

図 4.2.5　意図しない動作の例

ただし、システムの送信する Broadcast Intent のみを受信する BroadcastReceiver を実装する場合には、「exported=”false” かつ intent-filter 定義あり」を使用すること。かつ、これ以外の組み合わせは使っていけない。これは、システムが送信する Broadcast Intent に関しては exported=”false” でも受信可能であるという事実にもとづく。システムが送信する Broadcast Intent と同じ ACTION の Intent を他アプリが送信した場合、それを受信してしまうと意図しない動作を引き起こす可能性があるが、これは exported=”false” を指定することによって防ぐことができる。

4.2.3.2 Receiver はアプリを起動しないと登録されない

AndroidManifest.xml に静的に定義した Broadcast Receiver は、インストールしただけでは有効にならないので注意が必要である[*11]。アプリを 1 回起動することで、それ以降の Broadcast を受信できるようになるため、インストール後に Broadcast 受信をトリガーにして処理を起動させることはできない。ただし Broadcast の送信側で Intent に Intent.FLAG_INCLUDE_STOPPED_PACKAGES を設定して Broadcast 送信した場合は、一度も起動していないアプリであってもこの Broadcast を受信することができる。

[*11] Android 3.0 未満ではアプリのインストールをしただけで Receiver が登録される

4.2.3.3 同じ **UID** を持つアプリから送信された **Broadcast** は、非公開 **Broadcast Receiver** でも受信できる

複数のアプリに同じ UID を持たせることができる。この場合、たとえ非公開 Broadcast Receiver であっても、同じ UID のアプリから送信された Broadcast は受信してしまう。

しかしこれはセキュリティ上問題となることはない。同じ UID を持つアプリは APK を署名する開発者鍵が一致することが保証されており、非公開 Broadcast Receiver が受信するのは自社アプリから送信された Broadcast に限定されるからである。

4.2.3.4 **Broadcast** の種類とその特徴

送信する Broadcast は Ordered かそうでないか、Sticky かそうでないかの組み合わせにより 4 種類の Broadcast が存在する。Broadcast 送信用メソッドに応じて、送信する Broadcast の種類が決まる。なお、Sticky Broadcast の使用は Android 5.0（API Level 21）において非推奨となっている。

表 4.2.3　送信する Broadcast の種類

Broadcast の種類	送信用メソッド	**Ordered?**	**Sticky?**
Normal Broadcast	sendBroadcast()	No	No
Ordered Broadcast	sendOrderedBroadcast()	Yes	No
Sticky Broadcast	sendStickyBroadcast()	No	Yes
Sticky Ordered Broadcast	sendStickyOrderedBroadcast()	Yes	Yes

それぞれの Broadcast の特徴は次のとおりである。

表 4.2.4: 送信する Broadcast の種類ごとの特徴

Broadcast の種類	**Broadcast** の種類ごとの特徴
Normal Broadcast	Normal Broadcast は送信時に受信可能な状態にある Broadcast Receiver に配送されて消滅する。Ordered Broadcast と異なり、複数の Broadcast Receiver が同時に Broadcast を受信するのが特徴である。特定の Permission を持つアプリの BroadcastReceiver だけに Broadcast を受信させることもできる。
Ordered Broadcast	Ordered Broadcast は送信時に受信可能な状態にある BroadcastReceiver が一つずつ順番に Broadcast を受信することが特徴である。より priority 値が大きい Broadcast Receiver が先に受信する。すべての Broadcast Receiver に配送完了するか、途中の Broadcast Receiver が abortBroadcast() を呼び出した場合に、Broadcast は消滅する。特定の Permission を利用宣言したアプリの Broadcast Receiver だけに Broadcast を受信させることもできる。また Ordered Broadcast では送信元が Broadcast Receiver からの結果情報を受け取ることもできる。SMS 受信通知 Broadcast（SMS_RECEIVED）は Ordered Broadcast の代表例である。

次のページに続く

表 4.2.4 – 前のページからの続き

Sticky Broadcast	Sticky Broadcast は送信時に受信可能な状態にある BroadcastReceiver に配送された後に消滅することはなくシステムに残り続け、後に registerReceiver() を呼び出したアプリが Sticky Broadcast を受信することができることが特徴である。Sticky Broadcast は他の Broadcast と異なり自動的に消滅することはないので、Sticky Broadcast が不要になったときに、明示的に removeStickyBroadcast() を呼び出して Sticky Broadcast を消滅させる必要がある。他の Broadcast と異なり、特定の Permission を持つアプリの Broadcast Receiver だけに Broadcast を受信させることはできない。バッテリー状態変更通知 Broadcast （ACTION_BATTERY_CHANGED）は Sticky Broadcast の代表例である。
Sticky Ordered Broadcast	Ordered Broadcast と Sticky Broadcast の両方の特徴を持った Broadcast である。Sticky Broadcast と同様、特定の Permission を持つアプリの Broadcast Receiver だけに Broadcast を受信させることはできない。

Broadcast の特徴的な振る舞いの視点で、上表を逆引き的に再整理すると下表になる。

表 4.2.5: Broadcast の特徴的な振る舞い

Broadcast の特徴的な振る舞い	**Normal Broadcast**	**Ordered Broadcast**	**Sticky Broadcast**	**Sticky Ordered Broadcast**
受信可能な Broadcast Receive を Permission により制限する	o	o	-	-
Broadcast Receiver からの処理結果を取得する	-	o	-	o
順番に Broadcast Receiver に Broadcast を処理させる	-	o	-	o
既に送信されている Broadcast を後から受信する	-	-	o	o

4.2.3.5 Broadcast 送信した情報が LogCat に出力される場合がある

Broadcast の送受信は基本的に LogCat に出力されない。しかし、受信側の Permission 不足によるエラーや、送信側の Permission 不足によるエラーの際に LogCat にエラーログが出力される。エラーログには Broadcast で送信する Intent 情報も含まれるので、エラー発生時には Broadcast 送信する場合は LogCat に表示されることに注意してほしい。

送信側の Permission 不足時のエラー

```
W/ActivityManager(266): Permission Denial: broadcasting Intent {
act=org.jssec.android.broadcastreceiver.creating.action.MY_ACTION }
from org.jssec.android.broadcast.sending (pid=4685, uid=10058) requires
org.jssec.android.permission.MY_PERMISSION due to receiver
org.jssec.android.broadcastreceiver.creating/org.jssec.android.broadcastreceiver.creating.
↪CreatingType3Receiver
```

受信側の Permission 不足時のエラー

```
W/ActivityManager(275): Permission Denial: receiving Intent {
act=org.jssec.android.broadcastreceiver.creating.action.MY_ACTION } to
org.jssec.android.broadcastreceiver.creating requires
org.jssec.android.permission.MY_PERMISSION due to sender
org.jssec.android.broadcast.sending (uid 10158)
```

4.2.3.6 ホーム画面（アプリ）にショートカットを配置する際の注意点

ホーム画面にアプリを起動するためのショートカットボタンや Web ブラウザのブックマークのような URL ショートカットを作成する場合の注意点について説明する。例として、以下のような実装を考えてみる。

ホーム画面（アプリ）にショートカットを配置する

```
        Intent targetIntent = new Intent(this, TargetActivity.class);

       // ショートカット作成依頼のための Intent
        Intent intent = new Intent("com.android.launcher.action.INSTALL_SHORTCUT");

        // ショートカットのタップ時に起動する Intent を指定
        intent.putExtra(Intent.EXTRA_SHORTCUT_INTENT, targetIntent);
        Parcelable icon = Intent.ShortcutIconResource.fromContext(context, iconResource);
        intent.putExtra(Intent.EXTRA_SHORTCUT_ICON_RESOURCE, icon);
        intent.putExtra(Intent.EXTRA_SHORTCUT_NAME, title);
        intent.putExtra("duplicate", false);

        // BroadCast を使って、システムにショートカット作成を依頼する
        context.sendBroadcast(intent);
```

上記で送信している Broadcast は、受け手がホーム画面アプリであり、パッケージ名を特定することが難しいため、暗黙的 Intent による公開 Receiver への送信となっていることに注意が必要である。つまり、上記で送信した Broadcast はマルウェアを含めた任意のアプリが受信することができ、そのため、Intent にセンシティブな情報が含まれていると情報漏洩の被害につながる可能性がある。URL を基にしたショートカットを作成する場合、URL に秘密の情報が含まれていることもあるため、特に注意が必要である。

対策方法としては、「4.2.1.2. 公開 *Broadcast Receiver - Broadcast* を受信する・送信する」に記載されているポイントに従い、送信する Intent にセンシティブな情報が含まないようにすることが必要である。

4.3 Content Provider を作る・利用する

ContentResolver と SQLiteDatabase のインターフェースが似ているため、Content Provider は SQLiteDatabase と密接に関係があると勘違いされることが多い。しかし実際には Content Provider はアプリ間データ共有のインターフェースを規定するだけで、データ保存の形式は一切問わないことに注意が必要だ。作成する Content Provider 内部でデータの保存に SQLiteDatabase を使うこともできるし、XML ファイルなどの別の保存形式を使うこともできる。なお、ここで紹介するサンプルコードにはデータを保存する処理を含まないので、必要に応じて追加すること。

4.3.1 サンプルコード

Content Provider がどのように利用されるかによって、Content Provider が抱えるリスクや適切な防御手段が異なる。次の判定フローによって作成する Content Provider がどのタイプであるかを判断できる。

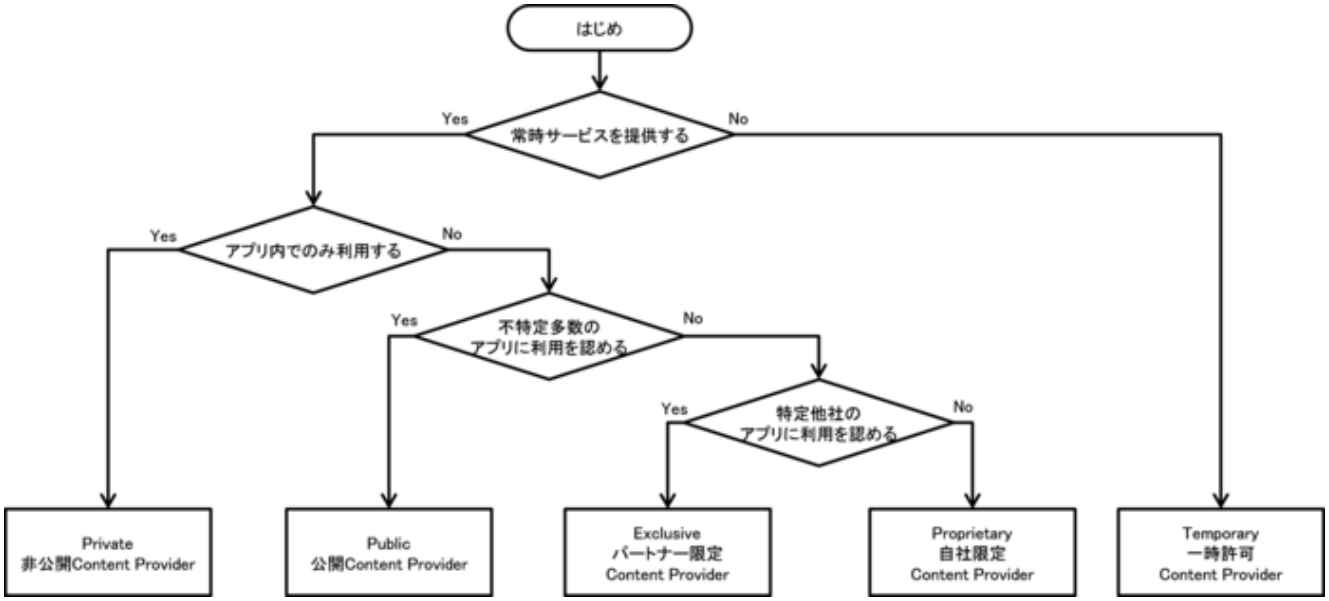

図 4.3.1　Content Provider タイプ判定フロー

4.3.1.1 非公開 Content Provider を作る・利用する

非公開 Content Provider は、同一アプリ内だけで利用される Content Provider であり、もっとも安全性の高い Content Provider である[*12]。

以下、非公開 Content Provider の実装例を示す。

ポイント (Content Provider を作る)：

1. exported=”false” により、明示的に非公開設定する
2. 同一アプリ内からのリクエストであっても、パラメータの安全性を確認する
3. 利用元アプリは同一アプリであるから、センシティブな情報を返送してよい

```
AndroidManifest.xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="org.jssec.android.provider.privateprovider">

    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name" >
        <activity
            android:name=".PrivateUserActivity"
            android:label="@string/app_name"
            android:exported="true" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

        <!-- ★ポイント 1★ exported="false"により、明示的に非公開設定する -->
```

(continues on next page)

[*12] ただし、Content Provider の非公開設定は Android 2.2 (API Level 8) 以前では機能しない。

(continued from previous page)

```
        <provider
            android:name=".PrivateProvider"
            android:authorities="org.jssec.android.provider.privateprovider"
            android:exported="false" />
    </application>
</manifest>
```

PrivateProvider.java

```
package org.jssec.android.provider.privateprovider;

import android.content.ContentProvider;
import android.content.ContentUris;
import android.content.ContentValues;
import android.content.UriMatcher;
import android.database.Cursor;
import android.database.MatrixCursor;
import android.net.Uri;

public class PrivateProvider extends ContentProvider {

    public static final String AUTHORITY = "org.jssec.android.provider.privateprovider";
    public static final String CONTENT_TYPE = "vnd.android.cursor.dir/vnd.org.jssec.contenttype";
    public static final String CONTENT_ITEM_TYPE = "vnd.android.cursor.item/vnd.org.jssec.contenttype";

    // Content Provider が提供するインターフェースを公開
    public interface Download {
        public static final String PATH = "downloads";
        public static final Uri CONTENT_URI = Uri.parse("content://" + AUTHORITY + "/" + PATH);
    }
    public interface Address {
        public static final String PATH = "addresses";
        public static final Uri CONTENT_URI = Uri.parse("content://" + AUTHORITY + "/" + PATH);
    }

    // UriMatcher
    private static final int DOWNLOADS_CODE = 1;
    private static final int DOWNLOADS_ID_CODE = 2;
    private static final int ADDRESSES_CODE = 3;
    private static final int ADDRESSES_ID_CODE = 4;
    private static UriMatcher sUriMatcher;
    static {
        sUriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
        sUriMatcher.addURI(AUTHORITY, Download.PATH, DOWNLOADS_CODE);
        sUriMatcher.addURI(AUTHORITY, Download.PATH + "/#", DOWNLOADS_ID_CODE);
        sUriMatcher.addURI(AUTHORITY, Address.PATH, ADDRESSES_CODE);
        sUriMatcher.addURI(AUTHORITY, Address.PATH + "/#", ADDRESSES_ID_CODE);
    }

    // DB を使用せずに固定値を返す例にしているため、query メソッドで返す Cursor を事前に定義
    private static MatrixCursor sAddressCursor = new MatrixCursor(new String[] { "_id", "pref" });
    static {
        sAddressCursor.addRow(new String[] { "1", "北海道" });
        sAddressCursor.addRow(new String[] { "2", "青森" });
        sAddressCursor.addRow(new String[] { "3", "岩手" });
```

(continues on next page)

(continued from previous page)

```
    }
    private static MatrixCursor sDownloadCursor = new MatrixCursor(new String[] { "_id", "path" });
    static {
        sDownloadCursor.addRow(new String[] { "1", "/sdcard/downloads/sample.jpg" });
        sDownloadCursor.addRow(new String[] { "2", "/sdcard/downloads/sample.txt" });
    }

    @Override
    public boolean onCreate() {

        return true;
    }

    @Override
    public String getType(Uri uri) {

        // ★ポイント 2★ 同一アプリ内からのリクエストであっても、パラメータの安全性を確認する
        // ここでは uri が想定の範囲内であることを、UriMatcher#match() と switch case で確認している。
        // 「3.2 入力データの安全性を確認する」を参照。
        // ★ポイント 3★ 利用元アプリは同一アプリであるから、センシティブな情報を返送してよい
        // ただし getType の結果がセンシティブな意味を持つことはあまりない。
        switch (sUriMatcher.match(uri)) {
        case DOWNLOADS_CODE:
        case ADDRESSES_CODE:
            return CONTENT_TYPE;

        case DOWNLOADS_ID_CODE:
        case ADDRESSES_ID_CODE:
            return CONTENT_ITEM_TYPE;

        default:
            throw new IllegalArgumentException("Invalid URI:" + uri);
        }
    }

    @Override
    public Cursor query(Uri uri, String[] projection, String selection,
            String[] selectionArgs, String sortOrder) {

        // ★ポイント 2★ 同一アプリ内からのリクエストであっても、パラメータの安全性を確認する
        // ここでは uri が想定の範囲内であることを、UriMatcher#match() と switch case で確認している。
        // その他のパラメータの確認はサンプルにつき省略。「3.2 入力データの安全性を確認する」を参照。
        // ★ポイント 3★ 利用元アプリは同一アプリであるから、センシティブな情報を返送してよい
        // query の結果がセンシティブな意味を持つかどうかはアプリ次第。
        switch (sUriMatcher.match(uri)) {
        case DOWNLOADS_CODE:
        case DOWNLOADS_ID_CODE:
            return sDownloadCursor;

        case ADDRESSES_CODE:
        case ADDRESSES_ID_CODE:
            return sAddressCursor;

        default:
            throw new IllegalArgumentException("Invalid URI:" + uri);
```

(continues on next page)

(continued from previous page)

```
        }
    }

    @Override
    public Uri insert(Uri uri, ContentValues values) {

        // ★ポイント 2★ 同一アプリ内からのリクエストであっても、パラメータの安全性を確認する
        // ここでは uri が想定の範囲内であることを、UriMatcher#match() と switch case で確認している。
        // その他のパラメータの確認はサンプルにつき省略。「3.2 入力データの安全性を確認する」を参照。
        // ★ポイント 3★ 利用元アプリは同一アプリであるから、センシティブな情報を返送してよい
        // Insert 結果、発番される ID がセンシティブな意味を持つかどうかはアプリ次第。
        switch (sUriMatcher.match(uri)) {
        case DOWNLOADS_CODE:
            return ContentUris.withAppendedId(Download.CONTENT_URI, 3);

        case ADDRESSES_CODE:
            return ContentUris.withAppendedId(Address.CONTENT_URI, 4);

        default:
            throw new IllegalArgumentException("Invalid URI : " + uri);
        }
    }

    @Override
    public int update(Uri uri, ContentValues values, String selection,
            String[] selectionArgs) {

        // ★ポイント 2★ 同一アプリ内からのリクエストであっても、パラメータの安全性を確認する
        // ここでは uri が想定の範囲内であることを、UriMatcher#match() と switch case で確認している。
        // その他のパラメータの確認はサンプルにつき省略。「3.2 入力データの安全性を確認する」を参照。
        // ★ポイント 3★ 利用元アプリは同一アプリであるから、センシティブな情報を返送してよい
        // Update されたレコード数がセンシティブな意味を持つかどうかはアプリ次第。
        switch (sUriMatcher.match(uri)) {
        case DOWNLOADS_CODE:
            return 5;   // update されたレコード数を返す

        case DOWNLOADS_ID_CODE:
            return 1;

        case ADDRESSES_CODE:
            return 15;

        case ADDRESSES_ID_CODE:
            return 1;

        default:
            throw new IllegalArgumentException("Invalid URI : " + uri);
        }
    }

    @Override
    public int delete(Uri uri, String selection, String[] selectionArgs) {

        // ★ポイント 2★ 同一アプリ内からのリクエストであっても、パラメータの安全性を確認する
        // ここでは uri が想定の範囲内であることを、UriMatcher#match() と switch case で確認している。
```

(continues on next page)

(continued from previous page)

```
        // その他のパラメータの確認はサンプルにつき省略。「3.2 入力データの安全性を確認する」を参照。
        // ★ポイント 3★ 利用元アプリは同一アプリであるから、センシティブな情報を返送してよい
        // Delete されたレコード数がセンシティブな意味を持つかどうかはアプリ次第。
        switch (sUriMatcher.match(uri)) {
        case DOWNLOADS_CODE:
            return 10;  // delete されたレコード数を返す

        case DOWNLOADS_ID_CODE:
            return 1;

        case ADDRESSES_CODE:
            return 20;

        case ADDRESSES_ID_CODE:
            return 1;

        default:
            throw new IllegalArgumentException("Invalid URI : " + uri);
        }
    }
}
```

次に、非公開 Content Provider を利用する Activity の例を示す。

ポイント (Content Provider を利用する) :

4. 同一アプリ内へのリクエストであるから、センシティブな情報をリクエストに含めてよい
5. 同一アプリ内からの結果情報であっても、受信データの安全性を確認する

```
PrivateUserActivity.java
package org.jssec.android.provider.privateprovider;

import android.app.Activity;
import android.database.Cursor;
import android.net.Uri;
import android.os.Bundle;
import android.view.View;
import android.widget.TextView;

public class PrivateUserActivity extends Activity {

    public void onQueryClick(View view) {
        logLine("[Query]");

        // ★ポイント 4★ 同一アプリ内へのリクエストであるから、センシティブな情報をリクエストに含めてよい
        Cursor cursor = null;
        try {
            cursor = getContentResolver().query(
                    PrivateProvider.Download.CONTENT_URI, null, null, null, null);

            // ★ポイント 5★ 同一アプリ内からの結果情報であっても、受信データの安全性を確認する
            // サンプルにつき割愛。「3.2 入力データの安全性を確認する」を参照。
            if (cursor == null) {
                logLine("  null cursor");
```

(continues on next page)

(continued from previous page)

```
            } else {
                boolean moved = cursor.moveToFirst();
                while (moved) {
                    logLine(String.format("  %d, %s", cursor.getInt(0), cursor.getString(1)));
                    moved = cursor.moveToNext();
                }
            }
        }
        finally {
            if (cursor != null) cursor.close();
        }

    }

    public void onInsertClick(View view) {

        logLine("[Insert]");

        // ★ポイント 4★ 同一アプリ内へのリクエストであるから、センシティブな情報をリクエストに含めてよい
        Uri uri = getContentResolver().insert(PrivateProvider.Download.CONTENT_URI, null);

        // ★ポイント 5★ 同一アプリ内からの結果情報であっても、受信データの安全性を確認する
        // サンプルにつき割愛。「3.2 入力データの安全性を確認する」を参照。
        logLine("  uri:" + uri);
    }

    public void onUpdateClick(View view) {

        logLine("[Update]");

        // ★ポイント 4★ 同一アプリ内へのリクエストであるから、センシティブな情報をリクエストに含めてよい
        int count = getContentResolver().update(PrivateProvider.Download.CONTENT_URI, null, null, null);

        // ★ポイント 5★ 同一アプリ内からの結果情報であっても、受信データの安全性を確認する
        // サンプルにつき割愛。「3.2 入力データの安全性を確認する」を参照。
        logLine(String.format("  %s records updated", count));
    }

    public void onDeleteClick(View view) {

        logLine("[Delete]");

        // ★ポイント 4★ 同一アプリ内へのリクエストであるから、センシティブな情報をリクエストに含めてよい
        int count = getContentResolver().delete(
                PrivateProvider.Download.CONTENT_URI, null, null);

        // ★ポイント 5★ 同一アプリ内からの結果情報であっても、受信データの安全性を確認する
        // サンプルにつき割愛。「3.2 入力データの安全性を確認する」を参照。
        logLine(String.format("  %s records deleted", count));
    }

    private TextView mLogView;

    @Override
    public void onCreate(Bundle savedInstanceState) {
```

(continues on next page)

(continued from previous page)

```
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        mLogView = (TextView)findViewById(R.id.logview);
    }

    private void logLine(String line) {
        mLogView.append(line);
        mLogView.append("\n");
    }
}
```

4.3.1.2 公開 Content Provider を作る・利用する

公開 Content Provider は、不特定多数のアプリに利用されることを想定した Content Provider である。クライアントを限定しないことにより、マルウェアから select() によって保持しているデータを抜き取られたり、update() によってデータを書き換えられたり、insert()/delete() によって偽のデータの挿入やデータの削除といった攻撃を受けたりして改ざんされ得ることに注意が必要だ。

また、Android OS 既定ではない独自作成の公開 Content Provider を利用する場合、その公開 Content Provider に成り済ましたマルウェアにリクエストパラメータを受信されることがあること、および、攻撃結果データを受け取ることがあることに注意が必要である。Android OS 既定の Contacts や MediaStore 等も公開 Content Provider であるが、マルウェアはそれら Content Provider に成り済ましできない。

以下、公開 Content Provider の実装例を示す。

ポイント (Content Provider を作る) :

1. exported=”true” により、明示的に公開設定する
2. リクエストパラメータの安全性を確認する
3. センシティブな情報を返送してはならない

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="org.jssec.android.provider.publicprovider">

    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name" >

        <!-- ★ポイント 1★ exported="true"により、明示的に公開設定する -->
        <provider
            android:name=".PublicProvider"
            android:authorities="org.jssec.android.provider.publicprovider"
            android:exported="true"/>
    </application>
</manifest>
```

PublicProvider.java

```
package org.jssec.android.provider.publicprovider;
```

(continues on next page)

(continued from previous page)

```
import android.content.ContentProvider;
import android.content.ContentUris;
import android.content.ContentValues;
import android.content.UriMatcher;
import android.database.Cursor;
import android.database.MatrixCursor;
import android.net.Uri;

public class PublicProvider extends ContentProvider {

    public static final String AUTHORITY = "org.jssec.android.provider.publicprovider";
    public static final String CONTENT_TYPE = "vnd.android.cursor.dir/vnd.org.jssec.contenttype";
    public static final String CONTENT_ITEM_TYPE = "vnd.android.cursor.item/vnd.org.jssec.contenttype";

    // Content Provider が提供するインターフェースを公開
    public interface Download {
        public static final String PATH = "downloads";
        public static final Uri CONTENT_URI = Uri.parse("content://" + AUTHORITY + "/" + PATH);
    }
    public interface Address {
        public static final String PATH = "addresses";
        public static final Uri CONTENT_URI = Uri.parse("content://" + AUTHORITY + "/" + PATH);
    }

    // UriMatcher
    private static final int DOWNLOADS_CODE = 1;
    private static final int DOWNLOADS_ID_CODE = 2;
    private static final int ADDRESSES_CODE = 3;
    private static final int ADDRESSES_ID_CODE = 4;
    private static UriMatcher sUriMatcher;
    static {
        sUriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
        sUriMatcher.addURI(AUTHORITY, Download.PATH, DOWNLOADS_CODE);
        sUriMatcher.addURI(AUTHORITY, Download.PATH + "/#", DOWNLOADS_ID_CODE);
        sUriMatcher.addURI(AUTHORITY, Address.PATH, ADDRESSES_CODE);
        sUriMatcher.addURI(AUTHORITY, Address.PATH + "/#", ADDRESSES_ID_CODE);
    }

    // DB を使用せずに固定値を返す例にしているため、query メソッドで返す Cursor を事前に定義
    private static MatrixCursor sAddressCursor = new MatrixCursor(new String[] { "_id", "pref" });
    static {
        sAddressCursor.addRow(new String[] { "1", "北海道" });
        sAddressCursor.addRow(new String[] { "2", "青森" });
        sAddressCursor.addRow(new String[] { "3", "岩手" });
    }
    private static MatrixCursor sDownloadCursor = new MatrixCursor(new String[] { "_id", "path" });
    static {
        sDownloadCursor.addRow(new String[] { "1", "/sdcard/downloads/sample.jpg" });
        sDownloadCursor.addRow(new String[] { "2", "/sdcard/downloads/sample.txt" });
    }

    @Override
    public boolean onCreate() {
        return true;
    }
```

(continues on next page)

(continued from previous page)

```
    @Override
    public String getType(Uri uri) {
        switch (sUriMatcher.match(uri)) {
        case DOWNLOADS_CODE:
        case ADDRESSES_CODE:
            return CONTENT_TYPE;

        case DOWNLOADS_ID_CODE:
        case ADDRESSES_ID_CODE:
            return CONTENT_ITEM_TYPE;

        default:
            throw new IllegalArgumentException("Invalid URI : " + uri);
        }
    }

    @Override
    public Cursor query(Uri uri, String[] projection, String selection,
            String[] selectionArgs, String sortOrder) {

        // ★ポイント 2★ リクエストパラメータの安全性を確認する
        // ここでは uri が想定の範囲内であることを、UriMatcher#match() と switch case で確認している。
        // その他のパラメータの確認はサンプルにつき省略。「3.2 入力データの安全性を確認する」を参照。
        // ★ポイント 3★ センシティブな情報を返送してはならない
        // query の結果がセンシティブな意味を持つかどうかはアプリ次第。
        // リクエスト元のアプリがマルウェアである可能性がある。
        // マルウェアに取得されても問題のない情報であれば結果として返してもよい。
        switch (sUriMatcher.match(uri)) {
        case DOWNLOADS_CODE:
        case DOWNLOADS_ID_CODE:
            return sDownloadCursor;

        case ADDRESSES_CODE:
        case ADDRESSES_ID_CODE:
            return sAddressCursor;

        default:
            throw new IllegalArgumentException("Invalid URI : " + uri);
        }
    }

    @Override
    public Uri insert(Uri uri, ContentValues values) {

        // ★ポイント 2★ リクエストパラメータの安全性を確認する
        // ここでは uri が想定の範囲内であることを、UriMatcher#match() と switch case で確認している。
        // その他のパラメータの確認はサンプルにつき省略。「3.2 入力データの安全性を確認する」を参照。
        // ★ポイント 3★ センシティブな情報を返送してはならない
        // Insert 結果、発番される ID がセンシティブな意味を持つかどうかはアプリ次第。
        // リクエスト元のアプリがマルウェアである可能性がある。
        // マルウェアに取得されても問題のない情報であれば結果として返してもよい。
        switch (sUriMatcher.match(uri)) {
        case DOWNLOADS_CODE:
            return ContentUris.withAppendedId(Download.CONTENT_URI, 3);
```

(continues on next page)

(continued from previous page)

```
        case ADDRESSES_CODE:
            return ContentUris.withAppendedId(Address.CONTENT_URI, 4);

        default:
            throw new IllegalArgumentException("Invalid URI : " + uri);
        }
    }

    @Override
    public int update(Uri uri, ContentValues values, String selection,
            String[] selectionArgs) {

        // ★ポイント 2★ リクエストパラメータの安全性を確認する
        // ここでは uri が想定の範囲内であることを、UriMatcher#match() と switch case で確認している。
        // その他のパラメータの確認はサンプルにつき省略。「3.2 入力データの安全性を確認する」を参照。
        // ★ポイント 3★ センシティブな情報を返送してはならない
        // Update されたレコード数がセンシティブな意味を持つかどうかはアプリ次第。
        // リクエスト元のアプリがマルウェアである可能性がある。
        // マルウェアに取得されても問題のない情報であれば結果として返してもよい。
        switch (sUriMatcher.match(uri)) {
        case DOWNLOADS_CODE:
            return 5;   // update されたレコード数を返す

        case DOWNLOADS_ID_CODE:
            return 1;

        case ADDRESSES_CODE:
            return 15;

        case ADDRESSES_ID_CODE:
            return 1;

        default:
            throw new IllegalArgumentException("Invalid URI : " + uri);
        }
    }


    @Override
    public int delete(Uri uri, String selection, String[] selectionArgs) {

        // ★ポイント 2★ リクエストパラメータの安全性を確認する
        // ここでは uri が想定の範囲内であることを、UriMatcher#match() と switch case で確認している。
        // その他のパラメータの確認はサンプルにつき省略。「3.2 入力データの安全性を確認する」を参照。
        // ★ポイント 3★ センシティブな情報を返送してはならない
        // Delete されたレコード数がセンシティブな意味を持つかどうかはアプリ次第。
        // リクエスト元のアプリがマルウェアである可能性がある。
        // マルウェアに取得されても問題のない情報であれば結果として返してもよい。
        switch (sUriMatcher.match(uri)) {
        case DOWNLOADS_CODE:
            return 10;  // delete されたレコード数を返す

        case DOWNLOADS_ID_CODE:
            return 1;
```

(continues on next page)

(continued from previous page)

```
        case ADDRESSES_CODE:
            return 20;

        case ADDRESSES_ID_CODE:
            return 1;

        default:
            throw new IllegalArgumentException("Invalid URI:" + uri);
        }
    }
}
```

次に、公開 Content Provider を利用する Activity の例を示す。

ポイント (Content Provider を利用する):

4. センシティブな情報をリクエストに含めてはならない

5. 結果データの安全性を確認する

PublicUserActivity.java

```
package org.jssec.android.provider.publicuser;

import android.app.Activity;
import android.content.ContentValues;
import android.content.pm.ProviderInfo;
import android.database.Cursor;
import android.net.Uri;
import android.os.Bundle;
import android.view.View;
import android.widget.TextView;

public class PublicUserActivity extends Activity {

    // 利用先の Content Provider 情報
    private static final String AUTHORITY = "org.jssec.android.provider.publicprovider";
    private interface Address {
        public static final String PATH = "addresses";
        public static final Uri CONTENT_URI = Uri.parse("content://" + AUTHORITY + "/" + PATH);
    }

    public void onQueryClick(View view) {

        logLine("[Query]");

        if (!providerExists(Address.CONTENT_URI)) {
            logLine("  Content Provider が不在");
            return;
        }

        // ★ポイント 4★ センシティブな情報をリクエストに含めてはならない
        // リクエスト先のアプリがマルウェアである可能性がある。
        // マルウェアに取得されても問題のない情報であればリクエストに含めてもよい。
        Cursor cursor = null;
```

(continues on next page)

(continued from previous page)

```
        try {
            cursor = getContentResolver().query(Address.CONTENT_URI, null, null, null, null);

            // ★ポイント 5★ 結果データの安全性を確認する
            // サンプルにつき割愛。「3.2 入力データの安全性を確認する」を参照。
            if (cursor == null) {
                logLine("  null cursor");
            } else {
                boolean moved = cursor.moveToFirst();

                while (moved) {
                    logLine(String.format("  %d, %s", cursor.getInt(0), cursor.getString(1)));
                    moved = cursor.moveToNext();
                }
            }
        }
        finally {
            if (cursor != null) cursor.close();
        }
    }

    public void onInsertClick(View view) {

        logLine("[Insert]");

        if (!providerExists(Address.CONTENT_URI)) {
            logLine("  Content Provider が不在");
            return;
        }

        // ★ポイント 4★ センシティブな情報をリクエストに含めてはならない
        // リクエスト先のアプリがマルウェアである可能性がある。
        // マルウェアに取得されても問題のない情報であればリクエストに含めてもよい。
        ContentValues values = new ContentValues();
        values.put("pref", "東京都");
        Uri uri = getContentResolver().insert(Address.CONTENT_URI, values);

        // ★ポイント 5★ 結果データの安全性を確認する
        // サンプルにつき割愛。「3.2 入力データの安全性を確認する」を参照。
        logLine("  uri:" + uri);
    }

    public void onUpdateClick(View view) {

        logLine("[Update]");

        if (!providerExists(Address.CONTENT_URI)) {
            logLine("  Content Provider が不在");
            return;
        }

        // ★ポイント 4★ センシティブな情報をリクエストに含めてはならない
        // リクエスト先のアプリがマルウェアである可能性がある。
        // マルウェアに取得されても問題のない情報であればリクエストに含めてもよい。
        ContentValues values = new ContentValues();
```

(continues on next page)

(continued from previous page)

```
        values.put("pref", "東京都");
        String where = "_id = ?";
        String[] args = { "4" };
        int count = getContentResolver().update(Address.CONTENT_URI, values, where, args);

        // ★ポイント 5★ 結果データの安全性を確認する
        // サンプルにつき割愛。「3.2 入力データの安全性を確認する」を参照。
        logLine(String.format("  %s records updated", count));
    }

    public void onDeleteClick(View view) {

        logLine("[Delete]");

        if (!providerExists(Address.CONTENT_URI)) {
            logLine("  Content Provider が不在");
            return;
        }

        // ★ポイント 4★ センシティブな情報をリクエストに含めてはならない
        // リクエスト先のアプリがマルウェアである可能性がある。
        // マルウェアに取得されても問題のない情報であればリクエストに含めてもよい。
        int count = getContentResolver().delete(Address.CONTENT_URI, null, null);

        // ★ポイント 5★ 結果データの安全性を確認する
        // サンプルにつき割愛。「3.2 入力データの安全性を確認する」を参照。
        logLine(String.format("  %s records deleted", count));
    }

    private boolean providerExists(Uri uri) {
        ProviderInfo pi = getPackageManager().resolveContentProvider(uri.getAuthority(), 0);
        return (pi != null);
    }

    private TextView mLogView;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        mLogView = (TextView)findViewById(R.id.logview);
    }

    private void logLine(String line) {
        mLogView.append(line);
        mLogView.append("\n");
    }
}
```

4.3.1.3 パートナー限定 Content Provider を作る・利用する

パートナー限定 Content Provider は、特定のアプリだけから利用できる Content Provider である。パートナー企業のアプリと自社アプリが連携してシステムを構成し、パートナーアプリとの間で扱う情報や機能を守るために利用される。

以下、パートナー限定 Content Provider の実装例を示す。

ポイント (Content Provider を作る)：

1. exported=”true” により、明示的に公開設定する
2. 利用元アプリの証明書がホワイトリストに登録されていることを確認する
3. パートナーアプリからのリクエストであっても、パラメータの安全性を確認する
4. パートナーアプリに開示してよい情報に限り返送してよい

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="org.jssec.android.provider.partnerprovider">

    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name" >

        <!-- ★ポイント 1★ exported="true"により、明示的に公開設定する -->
        <provider
            android:name="org.jssec.android.provider.partnerprovider.PartnerProvider"
            android:authorities="org.jssec.android.provider.partnerprovider"
            android:exported="true"/>
    </application>
</manifest>
```

PartnerProvider.java

```
package org.jssec.android.provider.partnerprovider;

import java.util.List;

import org.jssec.android.shared.PkgCertWhitelists;
import org.jssec.android.shared.Utils;

import android.app.ActivityManager;
import android.app.ActivityManager.RunningAppProcessInfo;
import android.content.ContentProvider;
import android.content.ContentUris;
import android.content.ContentValues;
import android.content.Context;
import android.content.UriMatcher;
import android.database.Cursor;
import android.database.MatrixCursor;
import android.net.Uri;
import android.os.Binder;
import android.os.Build;

public class PartnerProvider extends ContentProvider {

    public static final String AUTHORITY = "org.jssec.android.provider.partnerprovider";
    public static final String CONTENT_TYPE = "vnd.android.cursor.dir/vnd.org.jssec.contenttype";
    public static final String CONTENT_ITEM_TYPE = "vnd.android.cursor.item/vnd.org.jssec.contenttype";

```

(continues on next page)

(continued from previous page)

```
    // Content Provider が提供するインターフェースを公開
    public interface Download {
        public static final String PATH = "downloads";
        public static final Uri CONTENT_URI = Uri.parse("content://" + AUTHORITY + "/" + PATH);
    }

    public interface Address {
        public static final String PATH = "addresses";
        public static final Uri CONTENT_URI = Uri.parse("content://" + AUTHORITY + "/" + PATH);
    }

    // UriMatcher
    private static final int DOWNLOADS_CODE = 1;
    private static final int DOWNLOADS_ID_CODE = 2;
    private static final int ADDRESSES_CODE = 3;
    private static final int ADDRESSES_ID_CODE = 4;
    private static UriMatcher sUriMatcher;
    static {
        sUriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
        sUriMatcher.addURI(AUTHORITY, Download.PATH, DOWNLOADS_CODE);
        sUriMatcher.addURI(AUTHORITY, Download.PATH + "/#", DOWNLOADS_ID_CODE);
        sUriMatcher.addURI(AUTHORITY, Address.PATH, ADDRESSES_CODE);
        sUriMatcher.addURI(AUTHORITY, Address.PATH + "/#", ADDRESSES_ID_CODE);
    }

    // DB を使用せずに固定値を返す例にしているため、query メソッドで返す Cursor を事前に定義
    private static MatrixCursor sAddressCursor = new MatrixCursor(new String[] { "_id", "pref" });
    static {
        sAddressCursor.addRow(new String[] { "1", "北海道" });
        sAddressCursor.addRow(new String[] { "2", "青森" });
        sAddressCursor.addRow(new String[] { "3", "岩手" });
    }
    private static MatrixCursor sDownloadCursor = new MatrixCursor(new String[] { "_id", "path" });
    static {
        sDownloadCursor.addRow(new String[] { "1", "/sdcard/downloads/sample.jpg" });
        sDownloadCursor.addRow(new String[] { "2", "/sdcard/downloads/sample.txt" });
    }

    // ★ポイント 2★ 利用元アプリの証明書がホワイトリストに登録されていることを確認する
    private static PkgCertWhitelists sWhitelists = null;
    private static void buildWhitelists(Context context) {
        boolean isdebug = Utils.isDebuggable(context);
        sWhitelists = new PkgCertWhitelists();

        // パートナーアプリ org.jssec.android.provider.partneruser の証明書ハッシュ値を登録
        sWhitelists.add("org.jssec.android.provider.partneruser", isdebug ?
                // debug.keystore の"androiddebugkey"の証明書ハッシュ値
                "0EFB7236 328348A9 89718BAD DF57F544 D5CCB4AE B9DB34BC 1E29DD26 F77C8255" :
                // keystore の"partner key"の証明書ハッシュ値
                "1F039BB5 7861C27A 3916C778 8E78CE00 690B3974 3EB8259F E2627B8D 4C0EC35A");

        // 以下同様に他のパートナーアプリを登録...
    }
    private static boolean checkPartner(Context context, String pkgname) {
        if (sWhitelists == null) buildWhitelists(context);
```

(continues on next page)

(continued from previous page)

```
        return sWhitelists.test(context, pkgname);
    }
    // 利用元アプリのパッケージ名を取得
    private String getCallingPackage(Context context) {
        String pkgname;
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.KITKAT) {
            pkgname = super.getCallingPackage();
        } else {
            pkgname = null;
            ActivityManager am = (ActivityManager) context.getSystemService(Context.ACTIVITY_SERVICE);
            List<RunningAppProcessInfo> procList = am.getRunningAppProcesses();
            int callingPid = Binder.getCallingPid();
            if (procList != null) {
                for (RunningAppProcessInfo proc : procList) {
                    if (proc.pid == callingPid) {
                        pkgname = proc.pkgList[proc.pkgList.length - 1];
                        break;
                    }
                }
            }
        }

        return pkgname;
    }

    @Override
    public boolean onCreate() {
        return true;
    }

    @Override
    public String getType(Uri uri) {
        switch (sUriMatcher.match(uri)) {
        case DOWNLOADS_CODE:
        case ADDRESSES_CODE:
            return CONTENT_TYPE;

        case DOWNLOADS_ID_CODE:
        case ADDRESSES_ID_CODE:
            return CONTENT_ITEM_TYPE;

        default:
            throw new IllegalArgumentException("Invalid URI : " + uri);
        }
    }

    @Override
    public Cursor query(Uri uri, String[] projection, String selection,
            String[] selectionArgs, String sortOrder) {

        // ★ポイント２★ 利用元アプリの証明書がホワイトリストに登録されていることを確認する
        if (!checkPartner(getContext(), getCallingPackage(getContext()))) {
            throw new SecurityException("利用元アプリはパートナーアプリではない。");
        }
```

(continues on next page)

(continued from previous page)

```
        // ★ポイント 3★ パートナーアプリからのリクエストであっても、パラメータの安全性を確認する
        // ここでは uri が想定の範囲内であることを、UriMatcher#match() と switch case で確認している。
        // 「3.2 入力データの安全性を確認する」を参照。
        // ★ポイント 4★ パートナーアプリに開示してよい情報に限り返送してよい
        // query の結果がパートナーアプリに開示してよい情報かどうかはアプリ次第。
        switch (sUriMatcher.match(uri)) {
        case DOWNLOADS_CODE:
        case DOWNLOADS_ID_CODE:
            return sDownloadCursor;

        case ADDRESSES_CODE:
        case ADDRESSES_ID_CODE:
            return sAddressCursor;

        default:
            throw new IllegalArgumentException("Invalid URI:" + uri);
        }
    }

    @Override
    public Uri insert(Uri uri, ContentValues values) {

        // ★ポイント 2★ 利用元アプリの証明書がホワイトリストに登録されていることを確認する
        if (!checkPartner(getContext(), getCallingPackage(getContext()))) {
            throw new SecurityException("利用元アプリはパートナーアプリではない。");
        }

        // ★ポイント 3★ パートナーアプリからのリクエストであっても、パラメータの安全性を確認する
        // ここでは uri が想定の範囲内であることを、UriMatcher#match() と switch case で確認している。
        // 「3.2 入力データの安全性を確認する」を参照。
        // ★ポイント 4★ パートナーアプリに開示してよい情報に限り返送してよい
        // Insert 結果、発番される ID がパートナーアプリに開示してよい情報かどうかはアプリ次第。
        switch (sUriMatcher.match(uri)) {
        case DOWNLOADS_CODE:
            return ContentUris.withAppendedId(Download.CONTENT_URI, 3);

        case ADDRESSES_CODE:
            return ContentUris.withAppendedId(Address.CONTENT_URI, 4);

        default:
            throw new IllegalArgumentException("Invalid URI:" + uri);
        }
    }

    @Override
    public int update(Uri uri, ContentValues values, String selection,
            String[] selectionArgs) {

        // ★ポイント 2★ 利用元アプリの証明書がホワイトリストに登録されていることを確認する
        if (!checkPartner(getContext(), getCallingPackage(getContext()))) {
            throw new SecurityException("利用元アプリはパートナーアプリではない。");
        }

        // ★ポイント 3★ パートナーアプリからのリクエストであっても、パラメータの安全性を確認する
        // ここでは uri が想定の範囲内であることを、UriMatcher#match() と switch case で確認している。
```

(continues on next page)

(continued from previous page)

```
        // 「3.2 入力データの安全性を確認する」を参照。
        // ★ポイント 4★ パートナーアプリに開示してよい情報に限り返送してよい
        // Update されたレコード数がセンシティブな意味を持つかどうかはアプリ次第。
        switch (sUriMatcher.match(uri)) {
        case DOWNLOADS_CODE:
            return 5;   // update されたレコード数を返す

        case DOWNLOADS_ID_CODE:
            return 1;

        case ADDRESSES_CODE:
            return 15;

        case ADDRESSES_ID_CODE:
            return 1;

        default:
            throw new IllegalArgumentException("Invalid URI : " + uri);
        }
    }

    @Override
    public int delete(Uri uri, String selection, String[] selectionArgs) {

        // ★ポイント 2★ 利用元アプリの証明書がホワイトリストに登録されていることを確認する
        if (!checkPartner(getContext(), getCallingPackage(getContext()))) {
            throw new SecurityException("利用元アプリはパートナーアプリではない。");
        }

        // ★ポイント 3★ パートナーアプリからのリクエストであっても、パラメータの安全性を確認する
        // ここでは uri が想定の範囲内であることを、UriMatcher#match() と switch case で確認している。
        // 「3.2 入力データの安全性を確認する」を参照。
        // ★ポイント 4★ パートナーアプリに開示してよい情報に限り返送してよい
        // Delete されたレコード数がセンシティブな意味を持つかどうかはアプリ次第。
        switch (sUriMatcher.match(uri)) {
        case DOWNLOADS_CODE:
            return 10;  // delete されたレコード数を返す

        case DOWNLOADS_ID_CODE:
            return 1;

        case ADDRESSES_CODE:
            return 20;

        case ADDRESSES_ID_CODE:
            return 1;

        default:
            throw new IllegalArgumentException("Invalid URI : " + uri);
        }
    }
}
```

次に、パートナー限定 Content Provider を利用する Activity の例を示す。

ポイント (Content Provider を利用する)：

5. 利用先パートナー限定 Content Provider アプリの証明書がホワイトリストに登録されていることを確認する
6. パートナー限定 Content Provider アプリに開示してよい情報に限りリクエストに含めてよい
7. パートナー限定 Content Provider アプリからの結果であっても、結果データの安全性を確認する

PartnerUserActivity.java

```
package org.jssec.android.provider.partneruser;

import org.jssec.android.shared.PkgCertWhitelists;
import org.jssec.android.shared.Utils;

import android.app.Activity;
import android.content.ContentValues;
import android.content.Context;
import android.content.pm.ProviderInfo;
import android.database.Cursor;
import android.net.Uri;
import android.os.Bundle;
import android.view.View;
import android.widget.TextView;

public class PartnerUserActivity extends Activity {

    // 利用先の Content Provider 情報
    private static final String AUTHORITY = "org.jssec.android.provider.partnerprovider";
    private interface Address {
        public static final String PATH = "addresses";
        public static final Uri CONTENT_URI = Uri.parse("content://" + AUTHORITY + "/" + PATH);
    }

    // ★ポイント 5★ 利用先パートナー限定 Content Provider アプリの証明書がホワイトリストに登録されていることを確認
する
    private static PkgCertWhitelists sWhitelists = null;
    private static void buildWhitelists(Context context) {
        boolean isdebug = Utils.isDebuggable(context);
        sWhitelists = new PkgCertWhitelists();

        // パートナー限定 Content Provider アプリ org.jssec.android.provider.partnerprovider の証明書ハッシュ
値を登録
        sWhitelists.add("org.jssec.android.provider.partnerprovider", isdebug ?
                // debug.keystore の"androiddebugkey"の証明書ハッシュ値
                "0EFB7236 328348A9 89718BAD DF57F544 D5CCB4AE B9DB34BC 1E29DD26 F77C8255" :
                // keystore の"my company key"の証明書ハッシュ値
                "D397D343 A5CBC10F 4EDDEB7C A10062DE 5690984F 1FB9E88B D7B3A7C2 42E142CA");

        // 以下同様に他のパートナー限定 Content Provider アプリを登録...
    }
    private static boolean checkPartner(Context context, String pkgname) {
        if (sWhitelists == null) buildWhitelists(context);
        return sWhitelists.test(context, pkgname);
    }
    // uri を AUTHORITY とする Content Provider のパッケージ名を取得
    private String providerPkgname(Uri uri) {
        String pkgname = null;
```

(continues on next page)

(continued from previous page)

```
        ProviderInfo pi = getPackageManager().resolveContentProvider(uri.getAuthority(), 0);
        if (pi != null) pkgname = pi.packageName;
        return pkgname;
    }

    public void onQueryClick(View view) {

        logLine("[Query]");

        // ★ポイント 5★ 利用先パートナー限定 Content Provider アプリの証明書がホワイトリストに登録されていることを
確認する
        if (!checkPartner(this, providerPkgname(Address.CONTENT_URI))) {
            logLine("  利用先 Content Provider アプリはホワイトリストに登録されていない。");
            return;
        }

        // ★ポイント 6★ パートナー限定 Content Provider アプリに開示してよい情報に限りリクエストに含めてよい
        Cursor cursor = null;
        try {
            cursor = getContentResolver().query(Address.CONTENT_URI, null, null, null, null);

            // ★ポイント 7★ パートナー限定 Content Provider アプリからの結果であっても、結果データの安全性を確認
する
            // サンプルにつき割愛。「3.2 入力データの安全性を確認する」を参照。
            if (cursor == null) {
                logLine("  null cursor");
            } else {
                boolean moved = cursor.moveToFirst();
                while (moved) {
                    logLine(String.format("  %d, %s", cursor.getInt(0), cursor.getString(1)));
                    moved = cursor.moveToNext();
                }
            }
        }
        finally {
            if (cursor != null) cursor.close();
        }
    }

    public void onInsertClick(View view) {

        logLine("[Insert]");

        // ★ポイント 5★ 利用先パートナー限定 Content Provider アプリの証明書がホワイトリストに登録されていることを
確認する
        if (!checkPartner(this, providerPkgname(Address.CONTENT_URI))) {
            logLine("  利用先 Content Provider アプリはホワイトリストに登録されていない。");
            return;
        }

        // ★ポイント 6★ パートナー限定 Content Provider アプリに開示してよい情報に限りリクエストに含めてよい
        ContentValues values = new ContentValues();
        values.put("pref", "東京都");
        Uri uri = getContentResolver().insert(Address.CONTENT_URI, values);
```

(continues on next page)

(continued from previous page)

```
        // ★ポイント 7★ パートナー限定 Content Provider アプリからの結果であっても、結果データの安全性を確認する
        // サンプルにつき割愛。「3.2 入力データの安全性を確認する」を参照。
        logLine("  uri:" + uri);
    }

    public void onUpdateClick(View view) {

        logLine("[Update]");

        // ★ポイント 5★ 利用先パートナー限定 Content Provider アプリの証明書がホワイトリストに登録されていることを
確認する
        if (!checkPartner(this, providerPkgname(Address.CONTENT_URI))) {
            logLine("  利用先 Content Provider アプリはホワイトリストに登録されていない。");
            return;
        }

        // ★ポイント 6★ パートナー限定 Content Provider アプリに開示してよい情報に限りリクエストに含めてよい
        ContentValues values = new ContentValues();

        values.put("pref", "東京都");
        String where = "_id = ?";
        String[] args = { "4" };
        int count = getContentResolver().update(Address.CONTENT_URI, values, where, args);

        // ★ポイント 7★ パートナー限定 Content Provider アプリからの結果であっても、結果データの安全性を確認する
        // サンプルにつき割愛。「3.2 入力データの安全性を確認する」を参照。
        logLine(String.format("  %s records updated", count));
    }

    public void onDeleteClick(View view) {

        logLine("[Delete]");

        // ★ポイント 5★ 利用先パートナー限定 Content Provider アプリの証明書がホワイトリストに登録されていることを
確認する
        if (!checkPartner(this, providerPkgname(Address.CONTENT_URI))) {
            logLine("  利用先 Content Provider アプリはホワイトリストに登録されていない。");
            return;
        }

        // ★ポイント 6★ パートナー限定 Content Provider アプリに開示してよい情報に限りリクエストに含めてよい
        int count = getContentResolver().delete(Address.CONTENT_URI, null, null);

        // ★ポイント 7★ パートナー限定 Content Provider アプリからの結果であっても、結果データの安全性を確認する
        // サンプルにつき割愛。「3.2 入力データの安全性を確認する」を参照。
        logLine(String.format("  %s records deleted", count));
    }

    private TextView mLogView;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        mLogView = (TextView)findViewById(R.id.logview);
```

(continues on next page)

(continued from previous page)

```
    }

    private void logLine(String line) {
        mLogView.append(line);
        mLogView.append("\n");
    }
}
```

PkgCertWhitelists.java

```
package org.jssec.android.shared;

import android.content.pm.PackageManager;
import java.util.HashMap;
import java.util.Map;
import android.content.Context;
import android.os.Build;

import static android.content.pm.PackageManager.CERT_INPUT_SHA256;

public class PkgCertWhitelists {
    private Map<String, String> mWhitelists = new HashMap<String, String>();

    public boolean add(String pkgname, String sha256) {
        if (pkgname == null) return false;
        if (sha256 == null) return false;

        sha256 = sha256.replaceAll(" ", "");
        if (sha256.length() != 64) return false;    // SHA-256 は 32 バイト
        sha256 = sha256.toUpperCase();
        if (sha256.replaceAll("[0-9A-F]+", "").length() != 0) return false; // 0-9A-F 以外の文字がある

        mWhitelists.put(pkgname, sha256);
        return true;
    }

    public boolean test(Context ctx, String pkgname) {
        // pkgname に対応する正解のハッシュ値を取得する
        String correctHash = mWhitelists.get(pkgname);
        android.util.Log.d("Partner", "hash=" + correctHash);
        // pkgname の実際のハッシュ値と正解のハッシュ値を比較する
        if (Build.VERSION.SDK_INT >= 28) {
            // ★ API Level >= 28 では Package Manager の API で直接検証が可能
            PackageManager pm = ctx.getPackageManager();
            return pm.hasSigningCertificate(pkgname, hex2Bytes(correctHash), CERT_INPUT_SHA256);
        } else {
            // API Level < 28 の場合は PkgCert の機能を利用する
            return PkgCert.test(ctx, pkgname, correctHash);
        }
    }

    private byte[] hex2Bytes(String s) {
        int len = s.length();
        byte[] data = new byte[len / 2];
        for (int i = 0; i < len; i += 2) {
```

(continues on next page)

(continued from previous page)

```
            data[i / 2] = (byte) ((Character.digit(s.charAt(i), 16) << 4)
                    + Character.digit(s.charAt(i+1), 16));
        }
        return data;
    }
}
```

PkgCert.java

```
package org.jssec.android.shared;

import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

import android.content.Context;
import android.content.pm.PackageInfo;
import android.content.pm.PackageManager;
import android.content.pm.PackageManager.NameNotFoundException;
import android.content.pm.Signature;

public class PkgCert {

    public static boolean test(Context ctx, String pkgname, String correctHash) {
        if (correctHash == null) return false;
        correctHash = correctHash.replaceAll(" ", "");
        return correctHash.equals(hash(ctx, pkgname));
    }

    public static String hash(Context ctx, String pkgname) {
        if (pkgname == null) return null;
        try {
            PackageManager pm = ctx.getPackageManager();
            PackageInfo pkginfo = pm.getPackageInfo(pkgname, PackageManager.GET_SIGNATURES);
            if (pkginfo.signatures.length != 1) return null;    // 複数署名は扱わない
            Signature sig = pkginfo.signatures[0];
            byte[] cert = sig.toByteArray();
            byte[] sha256 = computeSha256(cert);
            return byte2hex(sha256);
        } catch (NameNotFoundException e) {
            return null;
        }
    }

    private static byte[] computeSha256(byte[] data) {
        try {
            return MessageDigest.getInstance("SHA-256").digest(data);
        } catch (NoSuchAlgorithmException e) {
            return null;
        }
    }

    private static String byte2hex(byte[] data) {
        if (data == null) return null;
        final StringBuilder hexadecimal = new StringBuilder();
        for (final byte b : data) {
```

(continues on next page)

(continued from previous page)

```
            hexadecimal.append(String.format("%02X", b));
        }
        return hexadecimal.toString();
    }
}
```

4.3.1.4 自社限定 Content Provider を作る・利用する

自社限定 Content Provider は、自社以外のアプリから利用されることを禁止する Content Provider である。複数の自社製アプリでシステムを構成し、自社アプリが扱う情報や機能を守るために利用される。

以下、自社限定 Content Provider の実装例を示す。

ポイント (Content Provider を作る) :

1. 独自定義 Signature Permission を定義する
2. 独自定義 Signature Permission を要求宣言する
3. exported=”true” により、明示的に公開設定する
4. 独自定義 Signature Permission が自社アプリにより定義されていることを確認する
5. 自社アプリからのリクエストであっても、パラメータの安全性を確認する
6. 利用元アプリは自社アプリであるから、センシティブな情報を返送してよい
7. 利用元アプリと同じ開発者鍵で APK を署名する

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="org.jssec.android.provider.inhouseprovider">

    <!-- ★ポイント 1★ 独自定義 Signature Permission を定義する -->
    <permission
        android:name="org.jssec.android.provider.inhouseprovider.MY_PERMISSION"
        android:protectionLevel="signature" />

    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name" >

        <!-- ★ポイント 2★ 独自定義 Signature Permission を要求宣言する -->
        <!-- ★ポイント 3★ exported="true"により、明示的に公開設定する -->
        <provider
            android:name="org.jssec.android.provider.inhouseprovider.InhouseProvider"
            android:authorities="org.jssec.android.provider.inhouseprovider"
            android:permission="org.jssec.android.provider.inhouseprovider.MY_PERMISSION"
            android:exported="true"/>
    </application>
</manifest>
```

InhouseProvider.java

```
package org.jssec.android.provider.inhouseprovider;

import org.jssec.android.shared.SigPerm;
import org.jssec.android.shared.Utils;

import android.content.ContentProvider;
import android.content.ContentUris;
import android.content.ContentValues;
import android.content.Context;
import android.content.UriMatcher;
import android.database.Cursor;
import android.database.MatrixCursor;
import android.net.Uri;

public class InhouseProvider extends ContentProvider {

    public static final String AUTHORITY = "org.jssec.android.provider.inhouseprovider";
    public static final String CONTENT_TYPE = "vnd.android.cursor.dir/vnd.org.jssec.contenttype";
    public static final String CONTENT_ITEM_TYPE = "vnd.android.cursor.item/vnd.org.jssec.contenttype";

    // Content Provider が提供するインターフェースを公開
    public interface Download {
        public static final String PATH = "downloads";
        public static final Uri CONTENT_URI = Uri.parse("content://" + AUTHORITY + "/" + PATH);
    }
    public interface Address {
        public static final String PATH = "addresses";
        public static final Uri CONTENT_URI = Uri.parse("content://" + AUTHORITY + "/" + PATH);
    }

    // UriMatcher
    private static final int DOWNLOADS_CODE = 1;
    private static final int DOWNLOADS_ID_CODE = 2;
    private static final int ADDRESSES_CODE = 3;
    private static final int ADDRESSES_ID_CODE = 4;
    private static UriMatcher sUriMatcher;
    static {
        sUriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
        sUriMatcher.addURI(AUTHORITY, Download.PATH, DOWNLOADS_CODE);
        sUriMatcher.addURI(AUTHORITY, Download.PATH + "/#", DOWNLOADS_ID_CODE);
        sUriMatcher.addURI(AUTHORITY, Address.PATH, ADDRESSES_CODE);
        sUriMatcher.addURI(AUTHORITY, Address.PATH + "/#", ADDRESSES_ID_CODE);
    }

    // DB を使用せずに固定値を返す例にしているため、query メソッドで返す Cursor を事前に定義
    private static MatrixCursor sAddressCursor = new MatrixCursor(new String[] { "_id", "pref" });
    static {
        sAddressCursor.addRow(new String[] { "1", "北海道" });
        sAddressCursor.addRow(new String[] { "2", "青森" });
        sAddressCursor.addRow(new String[] { "3", "岩手" });
    }
    private static MatrixCursor sDownloadCursor = new MatrixCursor(new String[] { "_id", "path" });
    static {
        sDownloadCursor.addRow(new String[] { "1", "/sdcard/downloads/sample.jpg" });
        sDownloadCursor.addRow(new String[] { "2", "/sdcard/downloads/sample.txt" });
```

(continues on next page)

(continued from previous page)

```
    }


    // 自社の Signature Permission
    private static final String MY_PERMISSION = "org.jssec.android.provider.inhouseprovider.MY_PERMISSION
↪";

    // 自社の証明書のハッシュ値
    private static String sMyCertHash = null;
    private static String myCertHash(Context context) {
        if (sMyCertHash == null) {
            if (Utils.isDebuggable(context)) {
                // debug.keystore の"androiddebugkey"の証明書ハッシュ値
                sMyCertHash = "0EFB7236 328348A9 89718BAD DF57F544 D5CCB4AE B9DB34BC 1E29DD26 F77C8255";
            } else {
                // keystore の"my company key"の証明書ハッシュ値
                sMyCertHash = "D397D343 A5CBC10F 4EDDEB7C A10062DE 5690984F 1FB9E88B D7B3A7C2 42E142CA";
            }
        }
        return sMyCertHash;
    }

    @Override
    public boolean onCreate() {
        return true;
    }

    @Override
    public String getType(Uri uri) {

        switch (sUriMatcher.match(uri)) {
        case DOWNLOADS_CODE:
        case ADDRESSES_CODE:
            return CONTENT_TYPE;

        case DOWNLOADS_ID_CODE:
        case ADDRESSES_ID_CODE:
            return CONTENT_ITEM_TYPE;

        default:
            throw new IllegalArgumentException("Invalid URI:" + uri);
        }
    }

    @Override
    public Cursor query(Uri uri, String[] projection, String selection,
            String[] selectionArgs, String sortOrder) {

        // ★ポイント 4★ 独自定義 Signature Permission が自社アプリにより定義されていることを確認する
        if (!SigPerm.test(getContext(), MY_PERMISSION, myCertHash(getContext()))) {
            throw new SecurityException("独自定義 Signature Permission が自社アプリにより定義されていない。");
        }

        // ★ポイント 5★ 自社アプリからのリクエストであっても、パラメータの安全性を確認する
        // ここでは uri が想定の範囲内であることを、UriMatcher#match() と switch case で確認している。
```

(continues on next page)

(continued from previous page)

```
        // 「3.2 入力データの安全性を確認する」を参照。
        // ★ポイント 6★ 利用元アプリは自社アプリであるから、センシティブな情報を返送してよい
        // query の結果が自社アプリに開示してよい情報かどうかはアプリ次第。
        switch (sUriMatcher.match(uri)) {
        case DOWNLOADS_CODE:
        case DOWNLOADS_ID_CODE:
            return sDownloadCursor;

        case ADDRESSES_CODE:
        case ADDRESSES_ID_CODE:
            return sAddressCursor;

        default:
            throw new IllegalArgumentException("Invalid URI:" + uri);
        }
    }

    @Override
    public Uri insert(Uri uri, ContentValues values) {

        // ★ポイント 4★ 独自定義 Signature Permission が自社アプリにより定義されていることを確認する
        if (!SigPerm.test(getContext(), MY_PERMISSION, myCertHash(getContext()))) {
            throw new SecurityException("独自定義 Signature Permission が自社アプリにより定義されていない。");
        }

        // ★ポイント 5★ 自社アプリからのリクエストであっても、パラメータの安全性を確認する
        // ここでは uri が想定の範囲内であることを、UriMatcher#match() と switch case で確認している。
        // 「3.2 入力データの安全性を確認する」を参照。
        // ★ポイント 6★ 利用元アプリは自社アプリであるから、センシティブな情報を返送してよい
        // Insert 結果、発番される ID が自社アプリに開示してよい情報かどうかはアプリ次第。
        switch (sUriMatcher.match(uri)) {
        case DOWNLOADS_CODE:
            return ContentUris.withAppendedId(Download.CONTENT_URI, 3);


        case ADDRESSES_CODE:
            return ContentUris.withAppendedId(Address.CONTENT_URI, 4);

        default:
            throw new IllegalArgumentException("Invalid URI:" + uri);
        }
    }

    @Override
    public int update(Uri uri, ContentValues values, String selection,
            String[] selectionArgs) {

        // ★ポイント 4★ 独自定義 Signature Permission が自社アプリにより定義されていることを確認する
        if (!SigPerm.test(getContext(), MY_PERMISSION, myCertHash(getContext()))) {
            throw new SecurityException("独自定義 Signature Permission が自社アプリにより定義されていない。");
        }

        // ★ポイント 5★ 自社アプリからのリクエストであっても、パラメータの安全性を確認する
        // ここでは uri が想定の範囲内であることを、UriMatcher#match() と switch case で確認している。
        // 「3.2 入力データの安全性を確認する」を参照。
```

(continues on next page)

(continued from previous page)

```
        // ★ポイント 6★ 利用元アプリは自社アプリであるから、センシティブな情報を返送してよい
        // Update されたレコード数がセンシティブな意味を持つかどうかはアプリ次第。
        switch (sUriMatcher.match(uri)) {
        case DOWNLOADS_CODE:
            return 5;   // update されたレコード数を返す

        case DOWNLOADS_ID_CODE:
            return 1;

        case ADDRESSES_CODE:
            return 15;

        case ADDRESSES_ID_CODE:
            return 1;

        default:
            throw new IllegalArgumentException("Invalid URI : " + uri);
        }
    }

    @Override
    public int delete(Uri uri, String selection, String[] selectionArgs) {

        // ★ポイント 4★ 独自定義 Signature Permission が自社アプリにより定義されていることを確認する
        if (!SigPerm.test(getContext(), MY_PERMISSION, myCertHash(getContext()))) {
            throw new SecurityException("独自定義 Signature Permission が自社アプリにより定義されていない。");
        }

        // ★ポイント 5★ 自社アプリからのリクエストであっても、パラメータの安全性を確認する
        // ここでは uri が想定の範囲内であることを、UriMatcher#match() と switch case で確認している。
        // 「3.2 入力データの安全性を確認する」を参照。
        // ★ポイント 6★ 利用元アプリは自社アプリであるから、センシティブな情報を返送してよい
        // Delete されたレコード数がセンシティブな意味を持つかどうかはアプリ次第。
        switch (sUriMatcher.match(uri)) {
        case DOWNLOADS_CODE:
            return 10;  // delete されたレコード数を返す

        case DOWNLOADS_ID_CODE:
            return 1;

        case ADDRESSES_CODE:
            return 20;

        case ADDRESSES_ID_CODE:
            return 1;

        default:
            throw new IllegalArgumentException("Invalid URI : " + uri);
        }
    }
}
```

SigPerm.java

```
package org.jssec.android.shared;
```

(continues on next page)

(continued from previous page)

```
import android.content.Context;
import android.content.pm.PackageManager;
import android.content.pm.PackageManager.NameNotFoundException;
import android.content.pm.PermissionInfo;
import android.os.Build;

import static android.content.pm.PackageManager.CERT_INPUT_SHA256;

public class SigPerm {

    public static boolean test(Context ctx, String sigPermName, String correctHash) {
        if (correctHash == null) return false;
        correctHash = correctHash.replaceAll(" ", "");
        try{
            // sigPermName を定義したアプリのパッケージ名を取得する
            PackageManager pm = ctx.getPackageManager();
            PermissionInfo pi = pm.getPermissionInfo(sigPermName, PackageManager.GET_META_DATA);
            String pkgname = pi.packageName;
            // 非 Signature Permission の場合は失敗扱い
            if (pi.protectionLevel != PermissionInfo.PROTECTION_SIGNATURE) return false;
            // pkgname の実際のハッシュ値と正解のハッシュ値を比較する
            if (Build.VERSION.SDK_INT >= 28) {
                // ★ API Level >= 28 では Package Manager の API で直接検証が可能
                return pm.hasSigningCertificate(pkgname, Utils.hex2Bytes(correctHash), CERT_INPUT_
↪SHA256);
            } else {
                // API Level < 28 の場合は PkgCert を利用し、ハッシュ値を取得して比較する
                return correctHash.equals(PkgCert.hash(ctx, pkgname));
            }
        } catch (NameNotFoundException e){
            return false;
        }
    }
}
```

PkgCert.java

```
package org.jssec.android.shared;

import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

import android.content.Context;
import android.content.pm.PackageInfo;
import android.content.pm.PackageManager;
import android.content.pm.PackageManager.NameNotFoundException;
import android.content.pm.Signature;

public class PkgCert {

    public static boolean test(Context ctx, String pkgname, String correctHash) {
        if (correctHash == null) return false;
        correctHash = correctHash.replaceAll(" ", "");
        return correctHash.equals(hash(ctx, pkgname));
```

(continues on next page)

(continued from previous page)

```
    }

    public static String hash(Context ctx, String pkgname) {
        if (pkgname == null) return null;
        try {
            PackageManager pm = ctx.getPackageManager();
            PackageInfo pkginfo = pm.getPackageInfo(pkgname, PackageManager.GET_SIGNATURES);
            if (pkginfo.signatures.length != 1) return null;    // 複数署名は扱わない
            Signature sig = pkginfo.signatures[0];
            byte[] cert = sig.toByteArray();
            byte[] sha256 = computeSha256(cert);
            return byte2hex(sha256);
        } catch (NameNotFoundException e) {
            return null;
        }
    }

    private static byte[] computeSha256(byte[] data) {
        try {
            return MessageDigest.getInstance("SHA-256").digest(data);
        } catch (NoSuchAlgorithmException e) {
            return null;
        }
    }

    private static String byte2hex(byte[] data) {
        if (data == null) return null;
        final StringBuilder hexadecimal = new StringBuilder();
        for (final byte b : data) {
            hexadecimal.append(String.format("%02X", b));
        }
        return hexadecimal.toString();
    }
}
```

★ポイント 7 ★　APK を Export するときに、利用元アプリと同じ開発者鍵で APK を署名する。

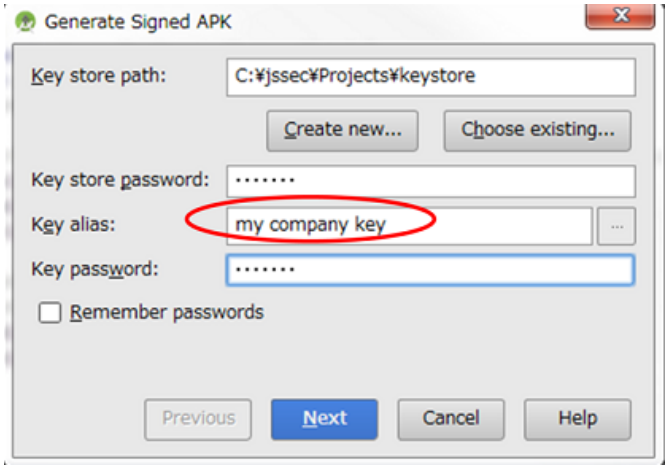

図 4.3.2　利用元アプリと同じ開発者鍵で APK を署名する

次に、自社限定 Content Provider を利用する Activity の例を示す。

ポイント (Content Provider を利用する)：

8. 独自定義 Signature Permission を利用宣言する
9. 独自定義 Signature Permission が自社アプリにより定義されていることを確認する
10. 利用先 Content Provider アプリの証明書が自社の証明書であることを確認する
11. 自社限定 Content Provider アプリに開示してよい情報に限りリクエストに含めてよい
12. 自社限定 Content Provider アプリからの結果であっても、結果データの安全性を確認する
13. 利用先アプリと同じ開発者鍵で APK を署名する

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="org.jssec.android.provider.inhouseuser">

    <!-- ★ポイント 8★ 独自定義 Signature Permission を利用宣言する -->
    <uses-permission
        android:name="org.jssec.android.provider.inhouseprovider.MY_PERMISSION" />

    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name" >
        <activity
            android:name="org.jssec.android.provider.inhouseuser.InhouseUserActivity"
            android:label="@string/app_name"
            android:exported="true" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

InhouseUserActivity.java

```
package org.jssec.android.provider.inhouseuser;

import org.jssec.android.shared.PkgCert;
import org.jssec.android.shared.SigPerm;

import org.jssec.android.shared.Utils;

import android.app.Activity;
import android.content.ContentValues;
import android.content.Context;
import android.content.pm.PackageManager;
import android.content.pm.ProviderInfo;
import android.database.Cursor;
import android.net.Uri;
import android.os.Bundle;
import android.view.View;
import android.widget.TextView;
```

(continues on next page)

(continued from previous page)

```
public class InhouseUserActivity extends Activity {

    // 利用先の Content Provider 情報
    private static final String AUTHORITY = "org.jssec.android.provider.inhouseprovider";
    private interface Address {
        public static final String PATH = "addresses";
        public static final Uri CONTENT_URI = Uri.parse("content://" + AUTHORITY + "/" + PATH);
    }

    // 自社の Signature Permission
    private static final String MY_PERMISSION = "org.jssec.android.provider.inhouseprovider.MY_PERMISSION
↪";

    // 自社の証明書のハッシュ値
    private static String sMyCertHash = null;
    private static String myCertHash(Context context) {
        if (sMyCertHash == null) {
            if (Utils.isDebuggable(context)) {
                // debug.keystore の"androiddebugkey"の証明書ハッシュ値
                sMyCertHash = "0EFB7236 328348A9 89718BAD DF57F544 D5CCB4AE B9DB34BC 1E29DD26 F77C8255";
            } else {
                // keystore の"my company key"の証明書ハッシュ値
                sMyCertHash = "D397D343 A5CBC10F 4EDDEB7C A10062DE 5690984F 1FB9E88B D7B3A7C2 42E142CA";
            }
        }
        return sMyCertHash;
    }

    // 利用先 Content Provider のパッケージ名を取得
    private static String providerPkgname(Context context, Uri uri) {
        String pkgname = null;
        PackageManager pm = context.getPackageManager();
        ProviderInfo pi = pm.resolveContentProvider(uri.getAuthority(), 0);
        if (pi != null) pkgname = pi.packageName;
        return pkgname;
    }

    public void onQueryClick(View view) {

        logLine("[Query]");

        // ★ポイント 9★ 独自定義 Signature Permission が自社アプリにより定義されていることを確認する
        if (!SigPerm.test(this, MY_PERMISSION, myCertHash(this))) {
            logLine("  独自定義 Signature Permission が自社アプリにより定義されていない。");
            return;
        }

        // ★ポイント 10★ 利用先 Content Provider アプリの証明書が自社の証明書であることを確認する
        String pkgname = providerPkgname(this, Address.CONTENT_URI);
        if (!PkgCert.test(this, pkgname, myCertHash(this))) {
            logLine("  利用先 Content Provider は自社アプリではない。");
            return;
        }

        // ★ポイント 11★ 自社限定 Content Provider アプリに開示してよい情報に限りリクエストに含めてよい
```

(continues on next page)

(continued from previous page)

```
        Cursor cursor = null;
        try {
            cursor = getContentResolver().query(Address.CONTENT_URI, null, null, null, null);

            // ★ポイント 12★ 自社限定 Content Provider アプリからの結果であっても、結果データの安全性を確認する
            // サンプルにつき割愛。「3.2 入力データの安全性を確認する」を参照。
            if (cursor == null) {
                logLine("  null cursor");
            } else {
                boolean moved = cursor.moveToFirst();
                while (moved) {
                    logLine(String.format("  %d, %s", cursor.getInt(0), cursor.getString(1)));
                    moved = cursor.moveToNext();
                }
            }
        }
        finally {
            if (cursor != null) cursor.close();

        }
    }

    public void onInsertClick(View view) {

        logLine("[Insert]");

        // ★ポイント 9★ 独自定義 Signature Permission が自社アプリにより定義されていることを確認する
        String correctHash = myCertHash(this);
        if (!SigPerm.test(this, MY_PERMISSION, correctHash)) {
            logLine("  独自定義 Signature Permission が自社アプリにより定義されていない。");
            return;
        }

        // ★ポイント 10★ 利用先 Content Provider アプリの証明書が自社の証明書であることを確認する
        String pkgname = providerPkgname(this, Address.CONTENT_URI);
        if (!PkgCert.test(this, pkgname, correctHash)) {
            logLine("  利用先 Content Provider は自社アプリではない。");
            return;
        }

        // ★ポイント 11★ 自社限定 Content Provider アプリに開示してよい情報に限りリクエストに含めてよい
        ContentValues values = new ContentValues();
        values.put("pref", "東京都");
        Uri uri = getContentResolver().insert(Address.CONTENT_URI, values);

        // ★ポイント 12★ 自社限定 Content Provider アプリからの結果であっても、結果データの安全性を確認する
        // サンプルにつき割愛。「3.2 入力データの安全性を確認する」を参照。
        logLine("  uri:" + uri);
    }

    public void onUpdateClick(View view) {

        logLine("[Update]");

        // ★ポイント 9★ 独自定義 Signature Permission が自社アプリにより定義されていることを確認する
```

(continues on next page)

(continued from previous page)

```
        String correctHash = myCertHash(this);
        if (!SigPerm.test(this, MY_PERMISSION, correctHash)) {
            logLine("  独自定義 Signature Permission が自社アプリにより定義されていない。");
            return;
        }

        // ★ポイント 10★ 利用先 Content Provider アプリの証明書が自社の証明書であることを確認する
        String pkgname = providerPkgname(this, Address.CONTENT_URI);
        if (!PkgCert.test(this, pkgname, correctHash)) {
            logLine("  利用先 Content Provider は自社アプリではない。");
            return;
        }

        // ★ポイント 11★ 自社限定 Content Provider アプリに開示してよい情報に限りリクエストに含めてよい
        ContentValues values = new ContentValues();
        values.put("pref", "東京都");
        String where = "_id = ?";
        String[] args = { "4" };
        int count = getContentResolver().update(Address.CONTENT_URI, values, where, args);

        // ★ポイント 12★ 自社限定 Content Provider アプリからの結果であっても、結果データの安全性を確認する
        // サンプルにつき割愛。「3.2 入力データの安全性を確認する」を参照。
        logLine(String.format("  %s records updated", count));
    }

    public void onDeleteClick(View view) {

        logLine("[Delete]");

        // ★ポイント 9★ 独自定義 Signature Permission が自社アプリにより定義されていることを確認する
        String correctHash = myCertHash(this);
        if (!SigPerm.test(this, MY_PERMISSION, correctHash)) {
            logLine("  独自定義 Signature Permission が自社アプリにより定義されていない。");
            return;
        }

        // ★ポイント 10★ 利用先 Content Provider アプリの証明書が自社の証明書であることを確認する
        String pkgname = providerPkgname(this, Address.CONTENT_URI);
        if (!PkgCert.test(this, pkgname, correctHash)) {
            logLine("  利用先 Content Provider は自社アプリではない。");
            return;
        }

        // ★ポイント 11★ 自社限定 Content Provider アプリに開示してよい情報に限りリクエストに含めてよい
        int count = getContentResolver().delete(Address.CONTENT_URI, null, null);

        // ★ポイント 12★ 自社限定 Content Provider アプリからの結果であっても、結果データの安全性を確認する
        // サンプルにつき割愛。「3.2 入力データの安全性を確認する」を参照。
        logLine(String.format("  %s records deleted", count));

    }

    private TextView mLogView;

    @Override
```

(continues on next page)

(continued from previous page)

```
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        mLogView = (TextView)findViewById(R.id.logview);
    }

    private void logLine(String line) {
        mLogView.append(line);
        mLogView.append("\n");
    }
}
```

SigPerm.java

```
package org.jssec.android.shared;

import android.content.Context;
import android.content.pm.PackageManager;
import android.content.pm.PackageManager.NameNotFoundException;
import android.content.pm.PermissionInfo;
import android.os.Build;

import static android.content.pm.PackageManager.CERT_INPUT_SHA256;

public class SigPerm {

    public static boolean test(Context ctx, String sigPermName, String correctHash) {
        if (correctHash == null) return false;
        correctHash = correctHash.replaceAll(" ", "");
        try{
            // sigPermName を定義したアプリのパッケージ名を取得する
            PackageManager pm = ctx.getPackageManager();
            PermissionInfo pi = pm.getPermissionInfo(sigPermName, PackageManager.GET_META_DATA);
            String pkgname = pi.packageName;
            // 非 Signature Permission の場合は失敗扱い
            if (pi.protectionLevel != PermissionInfo.PROTECTION_SIGNATURE) return false;
            // pkgname の実際のハッシュ値と正解のハッシュ値を比較する
            if (Build.VERSION.SDK_INT >= 28) {
                // ★ API Level >= 28 では Package Manager の API で直接検証が可能
                return pm.hasSigningCertificate(pkgname, Utils.hex2Bytes(correctHash), CERT_INPUT_
↪SHA256);
            } else {
                // API Level < 28 の場合は PkgCert を利用し、ハッシュ値を取得して比較する
                return correctHash.equals(PkgCert.hash(ctx, pkgname));
            }
        } catch (NameNotFoundException e){
            return false;
        }
    }
}
```

PkgCert.java

```
package org.jssec.android.shared;

import java.security.MessageDigest;
```

(continues on next page)

(continued from previous page)

```
import java.security.NoSuchAlgorithmException;

import android.content.Context;
import android.content.pm.PackageInfo;
import android.content.pm.PackageManager;
import android.content.pm.PackageManager.NameNotFoundException;
import android.content.pm.Signature;

public class PkgCert {

    public static boolean test(Context ctx, String pkgname, String correctHash) {
        if (correctHash == null) return false;
        correctHash = correctHash.replaceAll(" ", "");
        return correctHash.equals(hash(ctx, pkgname));
    }

    public static String hash(Context ctx, String pkgname) {
        if (pkgname == null) return null;
        try {
            PackageManager pm = ctx.getPackageManager();
            PackageInfo pkginfo = pm.getPackageInfo(pkgname, PackageManager.GET_SIGNATURES);
            if (pkginfo.signatures.length != 1) return null;    // 複数署名は扱わない
            Signature sig = pkginfo.signatures[0];
            byte[] cert = sig.toByteArray();
            byte[] sha256 = computeSha256(cert);
            return byte2hex(sha256);
        } catch (NameNotFoundException e) {
            return null;
        }
    }

    private static byte[] computeSha256(byte[] data) {
        try {
            return MessageDigest.getInstance("SHA-256").digest(data);
        } catch (NoSuchAlgorithmException e) {
            return null;
        }
    }

    private static String byte2hex(byte[] data) {
        if (data == null) return null;
        final StringBuilder hexadecimal = new StringBuilder();
        for (final byte b : data) {
            hexadecimal.append(String.format("%02X", b));
        }
        return hexadecimal.toString();
    }
}
```

★ポイント 13 ★ APK を Export するときに、利用先アプリと同じ開発者鍵で APK を署名する。

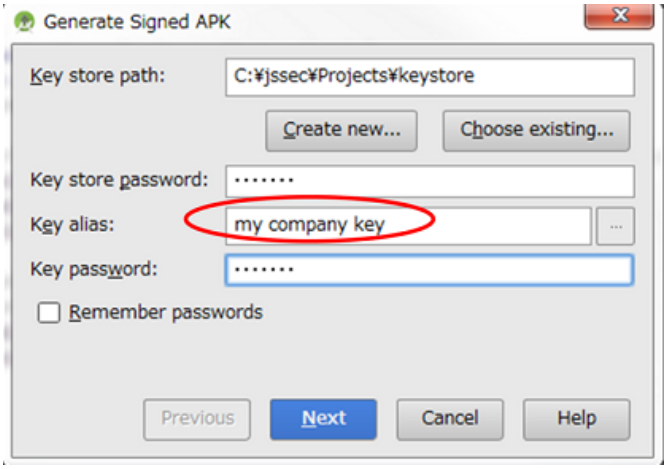

図 4.3.3　利用先アプリと同じ開発者鍵で APK を署名する

4.3.1.5 一時許可 Content Provider を作る・利用する

一時許可 Content Provider は、基本的には非公開の Content Provider であるが、特定のアプリに対して一時的に特定 URI へのアクセスを許可する Content Provider である。特殊なフラグを指定した Intent を対象アプリに送付することにより、そのアプリに一時的なアクセス権限が付されるようになっている。Content Provider 側アプリが能動的に他のアプリにアクセス許可を与えることもできるし、一時的なアクセス許可を求めてきたアプリに Content Provider 側アプリが受動的にアクセス許可を与えることもできる。

以下、一時許可 Content Provider の実装例を示す。

ポイント (Content Provider を作る) :

1. exported=”false” により、一時許可する Path 以外を非公開設定する
2. grant-uri-permission により、一時許可する Path を指定する
3. 一時的に許可したアプリからのリクエストであっても、パラメータの安全性を確認する
4. 一時的に許可したアプリに開示してよい情報に限り返送してよい
5. 一時的にアクセスを許可する URI を Intent に指定する
6. 一時的に許可するアクセス権限を Intent に指定する
7. 一時的にアクセスを許可するアプリに明示的 Intent を送信する
8. 一時許可の要求元アプリに Intent を返信する

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="org.jssec.android.provider.temporaryprovider">

    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name" >

        <activity
            android:name=".TemporaryActiveGrantActivity"
            android:label="@string/app_name"
```

(continues on next page)

(continued from previous page)

```
            android:exported="true" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

        <!-- 一時許可 Content Provider -->
        <!-- ★ポイント 1★ exported="false"により、一時許可する Path 以外を非公開設定する -->
        <provider
            android:name=".TemporaryProvider"
            android:authorities="org.jssec.android.provider.temporaryprovider"
            android:exported="false" >

            <!-- ★ポイント 2★ grant-uri-permission により、一時許可する Path を指定する -->
            <grant-uri-permission android:path="/addresses" />

        </provider>

        <activity
            android:name=".TemporaryPassiveGrantActivity"
            android:label="@string/app_name"
            android:exported="true" />
    </application>
</manifest>
```

TemporaryProvider.java

```
package org.jssec.android.provider.temporaryprovider;

import android.content.ContentProvider;
import android.content.ContentUris;
import android.content.ContentValues;
import android.content.UriMatcher;
import android.database.Cursor;
import android.database.MatrixCursor;
import android.net.Uri;

public class TemporaryProvider extends ContentProvider {

    public static final String AUTHORITIY = "org.jssec.android.provider.temporaryprovider";
    public static final String CONTENT_TYPE = "vnd.android.cursor.dir/vnd.org.jssec.contenttype";
    public static final String CONTENT_ITEM_TYPE = "vnd.android.cursor.item/vnd.org.jssec.contenttype";

    // Content Provider が提供するインターフェースを公開
    public interface Download {
        public static final String PATH = "downloads";
        public static final Uri CONTENT_URI = Uri.parse("content://" + AUTHORITIY + "/" + PATH);
    }
    public interface Address {
        public static final String PATH = "addresses";
        public static final Uri CONTENT_URI = Uri.parse("content://" + AUTHORITIY + "/" + PATH);
    }

    // UriMatcher
```

(continues on next page)

(continued from previous page)

```
    private static final int DOWNLOADS_CODE = 1;
    private static final int DOWNLOADS_ID_CODE = 2;
    private static final int ADDRESSES_CODE = 3;
    private static final int ADDRESSES_ID_CODE = 4;
    private static UriMatcher sUriMatcher;

    static {
        sUriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
        sUriMatcher.addURI(AUTHORITIY, Download.PATH, DOWNLOADS_CODE);
        sUriMatcher.addURI(AUTHORITIY, Download.PATH + "/#", DOWNLOADS_ID_CODE);
        sUriMatcher.addURI(AUTHORITIY, Address.PATH, ADDRESSES_CODE);
        sUriMatcher.addURI(AUTHORITIY, Address.PATH + "/#", ADDRESSES_ID_CODE);
    }

    // DB を使用せずに固定値を返す例にしているため、query メソッドで返す Cursor を事前に定義
    private static MatrixCursor sAddressCursor = new MatrixCursor(new String[] { "_id", "pref" });

    static {
        sAddressCursor.addRow(new String[] { "1", "北海道" });
        sAddressCursor.addRow(new String[] { "2", "青森" });
        sAddressCursor.addRow(new String[] { "3", "岩手" });
    }

    private static MatrixCursor sDownloadCursor = new MatrixCursor(new String[] { "_id", "path" });

    static {
        sDownloadCursor.addRow(new String[] { "1", "/sdcard/downloads/sample.jpg" });
        sDownloadCursor.addRow(new String[] { "2", "/sdcard/downloads/sample.txt" });
    }

    @Override
    public boolean onCreate() {
        return true;
    }

    @Override
    public String getType(Uri uri) {

        switch (sUriMatcher.match(uri)) {
        case DOWNLOADS_CODE:
        case ADDRESSES_CODE:
            return CONTENT_TYPE;

        case DOWNLOADS_ID_CODE:
        case ADDRESSES_ID_CODE:
            return CONTENT_ITEM_TYPE;

        default:
            throw new IllegalArgumentException("Invalid URI:" + uri);
        }
    }

    @Override
    public Cursor query(Uri uri, String[] projection, String selection,
            String[] selectionArgs, String sortOrder) {
```

(continues on next page)

(continued from previous page)

```
    // ★ポイント 3★ 一時的に許可したアプリからのリクエストであっても、パラメータの安全性を確認する
    // ここでは uri が想定の範囲内であることを、UriMatcher#match() と switch case で確認している。
    // その他のパラメータの確認はサンプルにつき省略。「3.2 入力データの安全性を確認する」を参照。
    // ★ポイント 4★ 一時的に許可したアプリに開示してよい情報に限り返送してよい
    // query の結果がセンシティブな意味を持つかどうかはアプリ次第。
    switch (sUriMatcher.match(uri)) {
    case DOWNLOADS_CODE:
    case DOWNLOADS_ID_CODE:
        return sDownloadCursor;

    case ADDRESSES_CODE:
    case ADDRESSES_ID_CODE:
        return sAddressCursor;

    default:
        throw new IllegalArgumentException("Invalid URI : " + uri);
    }
}

@Override
public Uri insert(Uri uri, ContentValues values) {

    // ★ポイント 3★ 一時的に許可したアプリからのリクエストであっても、パラメータの安全性を確認する
    // ここでは uri が想定の範囲内であることを、UriMatcher#match() と switch case で確認している。
    // その他のパラメータの確認はサンプルにつき省略。「3.2 入力データの安全性を確認する」を参照。
    // ★ポイント 4★ 一時的に許可したアプリに開示してよい情報に限り返送してよい
    // Insert 結果、発番される ID がセンシティブな意味を持つかどうかはアプリ次第。
    switch (sUriMatcher.match(uri)) {
    case DOWNLOADS_CODE:
        return ContentUris.withAppendedId(Download.CONTENT_URI, 3);

    case ADDRESSES_CODE:
        return ContentUris.withAppendedId(Address.CONTENT_URI, 4);

    default:
        throw new IllegalArgumentException("Invalid URI : " + uri);
    }
}

@Override
public int update(Uri uri, ContentValues values, String selection,
        String[] selectionArgs) {

    // ★ポイント 3★ 一時的に許可したアプリからのリクエストであっても、パラメータの安全性を確認する
    // ここでは uri が想定の範囲内であることを、UriMatcher#match() と switch case で確認している。
    // その他のパラメータの確認はサンプルにつき省略。「3.2 入力データの安全性を確認する」を参照。
    // ★ポイント 4★ 一時的に許可したアプリに開示してよい情報に限り返送してよい
    // Update されたレコード数がセンシティブな意味を持つかどうかはアプリ次第。
    switch (sUriMatcher.match(uri)) {
    case DOWNLOADS_CODE:
        return 5;   // update されたレコード数を返す

    case DOWNLOADS_ID_CODE:
        return 1;
```

(continues on next page)

(continued from previous page)

```
        case ADDRESSES_CODE:
            return 15;

        case ADDRESSES_ID_CODE:
            return 1;

        default:
            throw new IllegalArgumentException("Invalid URI:" + uri);
        }
    }

    @Override
    public int delete(Uri uri, String selection, String[] selectionArgs) {

        // ★ポイント3★ 一時的に許可したアプリからのリクエストであっても、パラメータの安全性を確認する
        // ここではuriが想定の範囲内であることを、UriMatcher#match()とswitch caseで確認している。
        // その他のパラメータの確認はサンプルにつき省略。「3.2 入力データの安全性を確認する」を参照。
        // ★ポイント4★ 一時的に許可したアプリに開示してよい情報に限り返送してよい
        // Deleteされたレコード数がセンシティブな意味を持つかどうかはアプリ次第。
        switch (sUriMatcher.match(uri)) {
        case DOWNLOADS_CODE:
            return 10;  // deleteされたレコード数を返す

        case DOWNLOADS_ID_CODE:
            return 1;

        case ADDRESSES_CODE:
            return 20;

        case ADDRESSES_ID_CODE:
            return 1;

        default:
            throw new IllegalArgumentException("Invalid URI:" + uri);
        }
    }
}
```

TemporaryActiveGrantActivity.java

```
package org.jssec.android.provider.temporaryprovider;

import android.app.Activity;
import android.content.ActivityNotFoundException;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.widget.Toast;

public class TemporaryActiveGrantActivity extends Activity {

    // User Activityに関する情報
    private static final String TARGET_PACKAGE =  "org.jssec.android.provider.temporaryuser";
    private static final String TARGET_ACTIVITY = "org.jssec.android.provider.temporaryuser.
↪TemporaryUserActivity";
```

(continues on next page)

(continued from previous page)

```
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.active_grant);
    }

    // Content Provider 側アプリが能動的に他のアプリにアクセス許可を与えるケース
    public void onSendClick(View view) {
        try {
            Intent intent = new Intent();

            // ★ポイント 5★ 一時的にアクセスを許可する URI を Intent に指定する
            intent.setData(TemporaryProvider.Address.CONTENT_URI);

            // ★ポイント 6★ 一時的に許可するアクセス権限を Intent に指定する
            intent.setFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION);

            // ★ポイント 7★ 一時的にアクセスを許可するアプリに明示的 Intent を送信する
            intent.setClassName(TARGET_PACKAGE, TARGET_ACTIVITY);
            startActivity(intent);

        } catch (ActivityNotFoundException e) {
            Toast.makeText(this, "User Activity が見つからない。", Toast.LENGTH_LONG).show();
        }
    }
}
```

TemporaryPassiveGrantActivity.java

```
package org.jssec.android.provider.temporaryprovider;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;

public class TemporaryPassiveGrantActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.passive_grant);
    }

    // 一時的なアクセス許可を求めてきたアプリに Content Provider 側アプリが受動的にアクセス許可を与えるケース
    public void onGrantClick(View view) {
        Intent intent = new Intent();

        // ★ポイント 5★ 一時的にアクセスを許可する URI を Intent に指定する
        intent.setData(TemporaryProvider.Address.CONTENT_URI);

        // ★ポイント 6★ 一時的に許可するアクセス権限を Intent に指定する
        intent.setFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION);

        // ★ポイント 8★ 一時許可の要求元アプリに Intent を返信する
```

(continues on next page)

(continued from previous page)

```
        setResult(Activity.RESULT_OK, intent);
        finish();
    }

    public void onCloseClick(View view) {
        finish();
    }
}
```

次に、一時許可 Content Provider を利用する Activity の例を示す。

ポイント (Content Provider を利用する)：

9. センシティブな情報をリクエストに含めてはならない

10. 結果データの安全性を確認する

```
TemporaryUserActivity.java
package org.jssec.android.provider.temporaryuser;

import android.app.Activity;
import android.content.ActivityNotFoundException;
import android.content.Intent;
import android.content.pm.ProviderInfo;
import android.database.Cursor;
import android.net.Uri;
import android.os.Bundle;
import android.view.View;
import android.widget.TextView;
import android.widget.Toast;

public class TemporaryUserActivity extends Activity {

    // Provider Activity に関する情報
    private static final String TARGET_PACKAGE =  "org.jssec.android.provider.temporaryprovider";
    private static final String TARGET_ACTIVITY = "org.jssec.android.provider.temporaryprovider.
↪TemporaryPassiveGrantActivity";

    // 利用先の Content Provider 情報
    private static final String AUTHORITY = "org.jssec.android.provider.temporaryprovider";
    private interface Address {
        public static final String PATH = "addresses";
        public static final Uri CONTENT_URI = Uri.parse("content://" + AUTHORITY + "/" + PATH);
    }

    private static final int REQUEST_CODE = 1;

    public void onQueryClick(View view) {

        logLine("[Query]");

        Cursor cursor = null;
        try {
            if (!providerExists(Address.CONTENT_URI)) {
                logLine("  Content Provider が不在");
```

(continues on next page)

(continued from previous page)

```
                return;
            }

            // ★ポイント 9★ センシティブな情報をリクエストに含めてはならない
            // リクエスト先のアプリがマルウェアである可能性がある。
            // マルウェアに取得されても問題のない情報であればリクエストに含めてもよい。
            cursor = getContentResolver().query(Address.CONTENT_URI, null, null, null, null);

            // ★ポイント 10★ 結果データの安全性を確認する
            // サンプルにつき割愛。「3.2 入力データの安全性を確認する」を参照。
            if (cursor == null) {
                logLine("  null cursor");
            } else {
                boolean moved = cursor.moveToFirst();
                while (moved) {
                    logLine(String.format("  %d, %s", cursor.getInt(0), cursor.getString(1)));
                    moved = cursor.moveToNext();
                }
            }
        } catch (SecurityException ex) {
            logLine("  例外:" + ex.getMessage());
        }
        finally {
            if (cursor != null) cursor.close();
        }
    }

    // このアプリが一時的なアクセス許可を要求し、Content Provider 側アプリが受動的にアクセス許可を与えるケース
    public void onGrantRequestClick(View view) {
        Intent intent = new Intent();
        intent.setClassName(TARGET_PACKAGE, TARGET_ACTIVITY);
        try {
            startActivityForResult(intent, REQUEST_CODE);
        } catch (ActivityNotFoundException e) {
            logLine("Grant の要求に失敗しました。\nTemporaryProvider がインストールされているか確認してください。
↪");
        }
    }

    private boolean providerExists(Uri uri) {
        ProviderInfo pi = getPackageManager().resolveContentProvider(uri.getAuthority(), 0);

        return (pi != null);
    }

    private TextView mLogView;

    // Content Provider 側アプリが能動的にこのアプリにアクセス許可を与えるケース
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        mLogView = (TextView)findViewById(R.id.logview);
    }
```

(continues on next page)

(continued from previous page)

```
    private void logLine(String line) {
        mLogView.append(line);
        mLogView.append("\n");
    }
}
```

4.3.2 ルールブック

Content Provider の実装時には以下のルールを守ること。

1. アプリ内でのみ使用する *Content Provider* は非公開設定する （必須）
2. リクエストパラメータの安全性を確認する （必須）
3. 独自定義 *Signature Permission* は、自社アプリが定義したことを確認して利用する （必須）
4. 結果情報を返す場合には、返送先アプリからの結果情報漏洩に注意する （必須）
5. 資産を二次的に提供する場合には、その資産の従来の保護水準を維持する （必須）

また、利用側は、以下のルールも守ること。

6. *Content Provider* の結果データの安全性を確認する （必須）

4.3.2.1 アプリ内でのみ使用する Content Provider は非公開設定する （必須）

同一アプリ内からのみ利用される Content Provider は他のアプリからアクセスできる必要がないだけでなく、開発者も Content Provider を攻撃するアクセスを考慮しないことが多い。Content Provider はデータ共有するための仕組みであるため、デフォルトでは公開扱いになってしまう。同一アプリ内からのみ利用される Content Provider は明示的に非公開設定し、非公開 Content Provider とすべきである。

```
AndroidManifest.xml
    <!-- ★ポイント 1★ exported="false"により、明示的に非公開設定する -->
    <provider
        android:name=".PrivateProvider"
        android:authorities="org.jssec.android.provider.privateprovider"
        android:exported="false" />
```

4.3.2.2 リクエストパラメータの安全性を確認する （必須）

Content Provider のタイプによって若干リスクは異なるが、リクエストパラメータを処理する際には、まずその安全性を確認しなければならない。

Content Provider の各メソッドは SQL 文の構成要素パラメータを受け取ることを想定したインターフェースになってはいるものの、仕組みの上では単に任意の文字列を受け渡すだけのものであり、Content Provider 側では想定外のパラメータが与えられるケースを想定しなければならないことに注意が必要だ。

公開 Content Provider は不特定多数のアプリからリクエストを受け取るため、マルウェアの攻撃リクエストを受け取る可能性がある。非公開 Content Provider は他のアプリからリクエストを直接受け取ることはない。しかし同一アプリ内の公開 Activity が他のアプリから受け取った Intent のデータを非公開 Content Provider に転送するといったケースも考えら

れるため、リクエストを無条件に安全であると考えてはならない。その他の Content Provider についても、やはりリクエストの安全性を確認する必要がある。

「3.2. 入力データの安全性を確認する」を参照すること。

4.3.2.3 独自定義 Signature Permission は、自社アプリが定義したことを確認して利用する （必須）

自社アプリだけから利用できる自社限定 Content Provider を作る場合、独自定義 Signature Permission により保護しなければならない。AndroidManifest.xml での Permission 定義、Permission 要求宣言だけでは保護が不十分であるため、「5.2. *Permission と Protection Level*」の「5.2.1.2. 独自定義の *Signature Permission* で自社アプリ連携する方法」を参照すること。

4.3.2.4 結果情報を返す場合には、返送先アプリからの結果情報漏洩に注意する （必須）

query() や insert() ではリクエスト要求元アプリに結果情報として Cursor や Uri が返送される。結果情報にセンシティブな情報が含まれる場合、返送先アプリから情報漏洩する可能性がある。また update() や delete() では更新または削除されたレコード数がリクエスト要求元アプリに結果情報として返送される。まれにアプリ仕様によっては更新または削除されたレコード数がセンシティブな意味を持つ場合があるので注意すべきだ。

4.3.2.5 資産を二次的に提供する場合には、その資産の従来の保護水準を維持する （必須）

Permission により保護されている情報資産および機能資産を他のアプリに二次的に提供する場合には、提供先アプリに対して同一の Permission を要求するなどして、その保護水準を維持しなければならない。Android の Permission セキュリティモデルでは、保護された資産に対するアプリからの直接アクセスについてのみ権限管理を行う。この仕様上の特性により、アプリに取得された資産がさらに他のアプリに、保護のために必要な Permission を要求することなく提供される可能性がある。このことは Permission を再委譲していることと実質的に等価なので、Permission の再委譲問題と呼ばれる。「5.2.3.4. *Permission* の再委譲問題」を参照すること。

4.3.2.6 Content Provider の結果データの安全性を確認する （必須）

Content Provider のタイプによって若干リスクは異なるが、結果データを処理する際には、まず結果データの安全性を確認しなければならない。

利用先 Content Provider が公開 Content Provider の場合、公開 Content Provider に成り済ましたマルウェアが攻撃結果データを返送してくる可能性がある。利用先 Content Provider が非公開 Content Provider の場合、同一アプリ内から結果データを受け取るのでリスクは少ないが、結果データを無条件に安全であると考えてはならない。その他の Content Provider についても、やはり結果データの安全性を確認する必要がある。

「3.2. 入力データの安全性を確認する」を参照すること。

4.4 Service を作る・利用する

4.4.1 サンプルコード

Service がどのように利用されるかによって、Service が抱えるリスクや適切な防御手段が異なる。次の判定フローによって作成する Service がどのタイプであるかを判断できる。なお、作成する Service のタイプによって Service を利用する側の実装も決まるので、利用側の実装についても合わせて説明する。

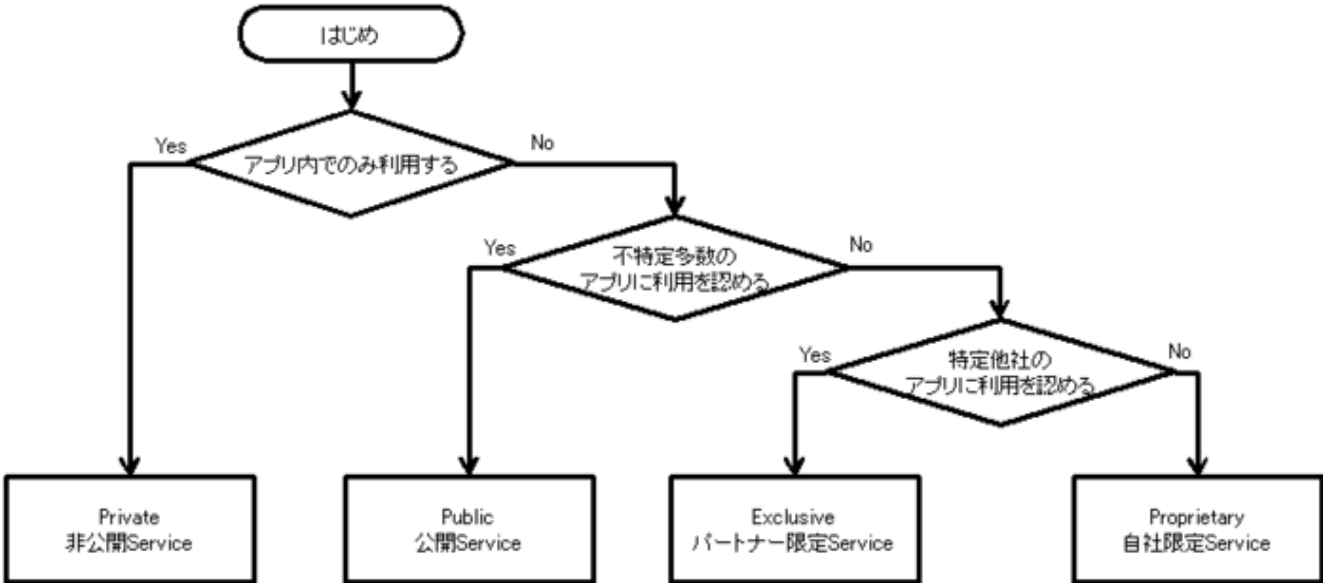

図 4.4.1　Service タイプの選択フロー

Service には複数の実装方法があり、その中から作成する Service のタイプに合った方法を選択することになる。下表の縦の項目が本文書で扱う実装方法であり、5 種類に分類した。表中の o 印は実現可能な組み合わせを示し、その他は実現不可能もしくは困難なものを示す。

なお、Service の実装方法の詳細については、「4.4.3.2. *Service* の実装方法について」および各 Service タイプのサンプルコード（表中で*印の付いたもの）を参照すること。

表 4.4.1　Service の実装方法

分類	非公開 **Service**	公開 **Service**	パートナー限定 **Service**	自社限定 **Service**
startService 型	**o***	o	-	o
IntentService 型	o	**o***	-	o
Messenger bind 型	o	o	-	**o***
AIDL bind 型	o	o	**o***	o

以下では 表 4.4.1 中の*印の組み合わせを使って各セキュリティタイプの Service のサンプルコードを示す。

4.4.1.1 非公開 Service を作る・利用する

非公開 Service は、同一アプリ内でのみ利用される Service であり、もっとも安全性の高い Service である。

また、非公開 Service を利用するには、クラスを指定する明示的 Intent を使えば誤って外部アプリに Intent を送信してしまうことがない。

以下、startService 型の Service を使用した例を示す。

ポイント (Service を作る）:

1. exported=”false” により、明示的に非公開設定する
2. 同一アプリからの Intent であっても、受信 Intent の安全性を確認する
3. 結果を返す場合、利用元アプリは同一アプリであるから、センシティブな情報を返送してよい

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="org.jssec.android.service.privateservice" >

    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:allowBackup="false">
        <activity
            android:name=".PrivateUserActivity"
            android:label="@string/app_name"
            android:exported="true" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

        <!-- 非公開 Service -->
        <!-- ★ポイント 1★ exported="false"により、明示的に非公開設定する -->
        <service android:name=".PrivateStartService" android:exported="false"/>

        <!-- IntentService を継承した Service -->
        <!-- 非公開 Service -->
        <!-- ★ポイント 1★ exported="false"により、明示的に非公開設定する -->
        <service android:name=".PrivateIntentService" android:exported="false"/>

        </application>

</manifest>
```

PrivateStartService.java

```
package org.jssec.android.service.privateservice;

import android.app.Service;
import android.content.Intent;
import android.os.IBinder;
import android.widget.Toast;

public class PrivateStartService extends Service{
    // Service が起動するときに 1 回だけ呼び出される
    @Override
    public void onCreate() {
        Toast.makeText(this, this.getClass().getSimpleName() + " - onCreate()", Toast.LENGTH_SHORT).
↪show();
    }

    // startService() が呼ばれた回数だけ呼び出される
    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        // ★ポイント 2★ 同一アプリからの Intent であっても、受信 Intent の安全性を確認する
        // サンプルにつき割愛。「3.2 入力データの安全性を確認する」を参照。
        String param = intent.getStringExtra("PARAM");
        Toast.makeText(this, String.format("パラメータ「%s」を受け取った。", param), Toast.LENGTH_LONG).
↪show();
```

(continues on next page)

(continued from previous page)

```
        // サービスは明示的に終了させる
        // stopSelf や stopService を実行したときにサービスを終了する
        // START_NOT_STICKY は、メモリが少ない等で kill された場合に自動的には復帰しない
        return Service.START_NOT_STICKY;
    }

    // Service が終了するときに 1 回だけ呼び出される
    @Override
    public void onDestroy() {
        Toast.makeText(this, this.getClass().getSimpleName() + " - onDestroy()", Toast.LENGTH_SHORT).
↪show();
    }

    @Override
    public IBinder onBind(Intent intent) {
        // このサービスにはバインドしない
        return null;
    }
}
```

次に非公開 Service を利用する Activity のサンプルコードを示す。

ポイント (Service を利用する)：

4. 同一アプリ内 Service はクラス指定の明示的 Intent で呼び出す
5. 利用先アプリは同一アプリであるから、センシティブな情報を送信してもよい
6. 結果を受け取る場合、同一アプリ内 Service からの結果情報であっても、受信データの安全性を確認する

```
PrivateUserActivity.java
package org.jssec.android.service.privateservice;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;

public class PrivateUserActivity extends Activity {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.privateservice_activity);
    }

    // サービス開始

    public void onStartServiceClick(View v) {
          // ★ポイント 4★ 同一アプリ内 Service はクラス指定の明示的 Intent で呼び出す
        Intent intent = new Intent(this, PrivateStartService.class);

          // ★ポイント 5★ 利用先アプリは同一アプリであるから、センシティブな情報を送信してもよい
        intent.putExtra("PARAM", "センシティブな情報");
```

(continues on next page)

(continued from previous page)

```
        startService(intent);
    }

    // サービス停止ボタン
    public void onStopServiceClick(View v) {
        doStopService();
    }

    @Override
    public void onStop(){
        super.onStop();
        // サービスが終了していない場合は終了する
        doStopService();
    }
    // サービスを停止する
    private void doStopService() {
        // ★ポイント 4★ 同一アプリ内 Service はクラス指定の明示的 Intent で呼び出す
        Intent intent = new Intent(this, PrivateStartService.class);
        stopService(intent);
    }

    // IntentService 開始ボタン

    public void onIntentServiceClick(View v) {
          // ★ポイント 4★ 同一アプリ内 Service はクラス指定の明示的 Intent で呼び出す
        Intent intent = new Intent(this, PrivateIntentService.class);

          // ★ポイント 5★ 利用先アプリは同一アプリであるから、センシティブな情報を送信してもよい
        intent.putExtra("PARAM", "センシティブな情報");

        startService(intent);
    }

}
```

4.4.1.2 公開 Service を作る・利用する

公開 Service は、不特定多数のアプリに利用されることを想定した Service である。マルウェアが送信した情報（Intent など）を受信することがあることに注意が必要である。また、公開 Service を利用するには、送信する情報（Intent など）がマルウェアに受信されることがあることに注意が必要である。

以下、IntentService 型の Service を使用した例を示す。

ポイント（Service を作る）：

1. exported=”true” により、明示的に公開設定する
2. 受信 Intent の安全性を確認する
3. 結果を返す場合、センシティブな情報を含めない

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
```

(continues on next page)

(continued from previous page)

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="org.jssec.android.service.publicservice" >

    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:allowBackup="false" >

        <!-- 最も標準的な Service -->
        <!-- ★ポイント 1★ exported="true"により、明示的に公開設定する -->
        <service android:name=".PublicStartService" android:exported="true">
            <intent-filter>
                <action android:name="org.jssec.android.service.publicservice.action.startservice" />
            </intent-filter>
        </service>

        <!-- IntentService を継承した Service -->
        <!-- ★ポイント 1★ exported="true"により、明示的に公開設定する -->
        <service android:name=".PublicIntentService" android:exported="true">
            <intent-filter>
                <action android:name="org.jssec.android.service.publicservice.action.intentservice" />
            </intent-filter>
        </service>

        </application>

</manifest>
```

PublicIntentService.java

```
package org.jssec.android.service.publicservice;

import android.app.IntentService;
import android.content.Intent;
import android.widget.Toast;

public class PublicIntentService extends IntentService{

    /**
     * IntentService を継承した場合、引数無しのコンストラクタを必ず用意する。
     * これが無い場合、エラーになる。
     */
    public PublicIntentService() {
        super("CreatingTypeBService");
    }

    // Service が起動するときに 1 回だけ呼び出される
    @Override
    public void onCreate() {
        super.onCreate();

        Toast.makeText(this, this.getClass().getSimpleName() + " - onCreate()", Toast.LENGTH_SHORT).
↪show();
    }
```

(continues on next page)

(continued from previous page)

```
    // Service で行いたい処理をこのメソッドに記述する
    @Override
    protected void onHandleIntent(Intent intent) {
        // ★ポイント 2★ 受信 Intent の安全性を確認する
        // 公開 Activity であるため利用元アプリがマルウェアである可能性がある。
        // サンプルにつき割愛。「3.2 入力データの安全性を確認する」を参照。
        String param = intent.getStringExtra("PARAM");
        Toast.makeText(this, String.format("パラメータ「%s」を受け取った。", param), Toast.LENGTH_LONG).
↪show();
    }


    // Service が終了するときに 1 回だけ呼び出される
    @Override
    public void onDestroy() {
        Toast.makeText(this, this.getClass().getSimpleName() + " - onDestroy()", Toast.LENGTH_SHORT).
↪show();
    }

}
```

次に公開 Service を利用する Activity のサンプルコードを示す。

ポイント（Service を利用する）：

4. センシティブな情報を送信してはならない
5. 結果を受け取る場合、結果データの安全性を確認する

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="org.jssec.android.service.publicserviceuser" >

    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:allowBackup="false" >
        <activity
            android:name=".PublicUserActivity"
            android:label="@string/app_name"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

    </application>

</manifest>
```

PublicUserActivity.java

```
package org.jssec.android.service.publicserviceuser;
```

(continues on next page)

(continued from previous page)

```
import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;

public class PublicUserActivity extends Activity {

    // 利用先 Service 情報
    private static final String TARGET_PACKAGE = "org.jssec.android.service.publicservice";
    private static final String TARGET_START_CLASS = "org.jssec.android.service.publicservice.
↪PublicStartService";
    private static final String TARGET_INTENT_CLASS = "org.jssec.android.service.publicservice.
↪PublicIntentService";

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        setContentView(R.layout.publicservice_activity);
    }

    // サービス開始
    public void onStartServiceClick(View v) {

        Intent intent = new Intent("org.jssec.android.service.publicservice.action.startservice");

        // 利用先 Service が固定できる場合は明示的 Intent で Service を利用する
        intent.setClassName(TARGET_PACKAGE, TARGET_START_CLASS);

        // ★ポイント 4★ センシティブな情報を送信してはならない
        intent.putExtra("PARAM", "センシティブではない情報");

        startService(intent);

     // ★ポイント 5★ 結果を受け取る場合、結果データの安全性を確認する
     // 本サンプルは startService() を使った Service 利用の例の為、結果情報は受け取らない
    }

    // サービス停止ボタン
    public void onStopServiceClick(View v) {
        doStopService();
    }

    // IntentService 開始ボタン

    public void onIntentServiceClick(View v) {
        Intent intent = new Intent("org.jssec.android.service.publicservice.action.intentservice");

        // 利用先 Service が固定できる場合は明示的 Intent で Service を利用する
        intent.setClassName(TARGET_PACKAGE, TARGET_INTENT_CLASS);

        // ★ポイント 4★ センシティブな情報を送信してはならない
        intent.putExtra("PARAM", "センシティブではない情報");

        startService(intent);
```

(continues on next page)

(continued from previous page)

```
    }

    @Override
    public void onStop(){
        super.onStop();
        // サービスが終了していない場合は終了する
        doStopService();
    }

    // サービスを停止する
    private void doStopService() {
        Intent intent = new Intent("org.jssec.android.service.publicservice.action.startservice");

        // 利用先 Service が固定できる場合は明示的 Intent で Service を利用する
        intent.setClassName(TARGET_PACKAGE, TARGET_START_CLASS);

        stopService(intent);
    }
}
```

4.4.1.3 パートナー限定 Service

パートナー限定 Service は、特定のアプリだけから利用できる Service である。パートナー企業のアプリと自社アプリが連携してシステムを構成し、パートナーアプリとの間で扱う情報や機能を守るために利用される。

以下、AIDL bind 型の Service を使用した例を示す。

ポイント (Service を作る) :

1. Intent Filter を定義せず、exported=”true” を明示的に設定する
2. 利用元アプリの証明書がホワイトリストに登録されていることを確認する
3. onBind(onStartCommand,onHandleIntent) で呼び出し元がパートナーかどうか判別できない
4. パートナーアプリからの Intent であっても、受信 Intent の安全性を確認する
5. パートナーアプリに開示してよい情報に限り返送してよい

なお、ホワイトリストに指定する利用先アプリの証明書ハッシュ値の確認方法は「5.2.1.3. アプリの証明書のハッシュ値を確認する方法」を参照すること。

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="org.jssec.android.service.partnerservice.aidl" >

    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:allowBackup="false">

       <!-- AIDL を利用した Service -->
        <!-- ★ポイント 1★ Intent Filter を定義せず、exported="true"を明示的に設定する -->
```

(continues on next page)

(continued from previous page)

```
        <service
            android:name="org.jssec.android.service.partnerservice.aidl.PartnerAIDLService"
            android:exported="true" />
    </application>

</manifest>
```

今回の例では AIDL ファイルを２つ作成する。１つは、Service から Activity にデータを渡すためのコールバックインターフェースで、もう１つは Activity から Service にデータを渡し、情報を取得するインターフェースである。なお、AIDL ファイルに記述するパッケージ名は、java ファイルに記述するパッケージ名と同様に、AIDL ファイルを作成するディレクトリ階層に一致させる必要がある。

IPartnerAIDLServiceCallback.aidl

```
package org.jssec.android.service.partnerservice.aidl;

interface IPartnerAIDLServiceCallback {
    /**
     * 値が変わった時に呼び出される
     */
    void valueChanged(String info);
}
```

IPartnerAIDLService.aidl

```
package org.jssec.android.service.partnerservice.aidl;

import org.jssec.android.service.partnerservice.aidl.IExclusiveAIDLServiceCallback;

interface IPartnerAIDLService {

    /**
     * コールバックを登録する
     */
    void registerCallback(IPartnerAIDLServiceCallback cb);

    /**
     * 情報を取得する
     */
    String getInfo(String param);

    /**
     * コールバックを解除する
     */
    void unregisterCallback(IPartnerAIDLServiceCallback cb);
}
```

PartnerAIDLService.java

```
package org.jssec.android.service.partnerservice.aidl;

import org.jssec.android.shared.PkgCertWhitelists;
import org.jssec.android.shared.Utils;

import android.app.Service;
import android.content.Context;
```

(continues on next page)

(continued from previous page)

```
import android.content.Intent;
import android.os.Handler;
import android.os.IBinder;
import android.os.Message;
import android.os.RemoteCallbackList;
import android.os.RemoteException;
import android.widget.Toast;

public class PartnerAIDLService extends Service {
       private static final int REPORT_MSG = 1;
       private static final int GETINFO_MSG = 2;

    // Service からクライアントに通知する値
    private int mValue = 0;

    // ★ポイント 2★ 利用元アプリの証明書がホワイトリストに登録されていることを確認する
    private static PkgCertWhitelists sWhitelists = null;
    private static void buildWhitelists(Context context) {
        boolean isdebug = Utils.isDebuggable(context);
        sWhitelists = new PkgCertWhitelists();

        // パートナーアプリ org.jssec.android.service.partnerservice.aidluser の証明書ハッシュ値を登録
        sWhitelists.add("org.jssec.android.service.partnerservice.aidluser", isdebug ?
                // debug.keystore の "androiddebugkey"の証明書ハッシュ値
                "0EFB7236 328348A9 89718BAD DF57F544 D5CCB4AE B9DB34BC 1E29DD26 F77C8255" :
                // keystore の "partner key"の証明書ハッシュ値
                "1F039BB5 7861C27A 3916C778 8E78CE00 690B3974 3EB8259F E2627B8D 4C0EC35A");

        // 以下同様に他のパートナーアプリを登録...
    }

    private static boolean checkPartner(Context context, String pkgname) {
        if (sWhitelists == null) buildWhitelists(context);
        return sWhitelists.test(context, pkgname);
    }

    // コールバックを登録するオブジェクト。
    // RemoteCallbackList の提供するメソッドはスレッドセーフになっている。
    private final RemoteCallbackList<IPartnerAIDLServiceCallback> mCallbacks =
        new RemoteCallbackList<>();

    // コールバックに対して Service からデータを送信するための Handler
    protected static class ServiceHandler extends Handler{

        private Context mContext;
        private RemoteCallbackList<IPartnerAIDLServiceCallback> mCallbacks;
        private int mValue = 0;

        public ServiceHandler(Context context, RemoteCallbackList<IPartnerAIDLServiceCallback> callback,␣
↪int value){
            this.mContext = context;
            this.mCallbacks = callback;
            this.mValue = value;
        }
```

(continues on next page)

(continued from previous page)

```
        @Override
        public void handleMessage(Message msg) {
            switch (msg.what) {
                case REPORT_MSG: {
                    if(mCallbacks == null){
                        return;
                    }
                    // 通知を開始する
                    // beginBroadcast() は、getBroadcastItem() で取得可能なコピーを作成している
                    final int N = mCallbacks.beginBroadcast();
                    for (int i = 0; i < N; i++) {
                        IPartnerAIDLServiceCallback target = mCallbacks.getBroadcastItem(i);
                        try {
                            // ★ポイント 5★ パートナーアプリに開示してよい情報に限り送信してよい
                            target.valueChanged("パートナーアプリに開示してよい情報 (callback from Service)␣
↪No." + (++mValue));

                        } catch (RemoteException e) {
                            // RemoteCallbackList がコールバックを管理しているので、ここでは unregeister しない
                            // RemoteCallbackList.kill() によって全て解除される
                        }
                    }
                    // finishBroadcast() は、beginBroadcast() と対になる処理
                    mCallbacks.finishBroadcast();

                    // 10 秒後に繰り返す
                    sendEmptyMessageDelayed(REPORT_MSG, 10000);
                    break;
                }
                case GETINFO_MSG: {
                    if(mContext != null) {
                        Toast.makeText(mContext,
                                (String) msg.obj, Toast.LENGTH_LONG).show();
                    }
                    break;
                }
                default:
                    super.handleMessage(msg);
                    break;
            } // switch
        }
    }

    protected final ServiceHandler mHandler = new ServiceHandler(this, mCallbacks, mValue);

    // AIDL で定義したインターフェース
    private final IPartnerAIDLService.Stub mBinder = new IPartnerAIDLService.Stub() {
        private boolean checkPartner() {
            Context ctx = PartnerAIDLService.this;
            if (!PartnerAIDLService.checkPartner(ctx, Utils.getPackageNameFromUid(ctx,␣
↪getCallingUid()))) {
                mHandler.post(new Runnable() {
                    @Override
                    public void run() {
                        Toast.makeText(PartnerAIDLService.this, "利用元アプリはパートナーアプリではない。",␣
↪Toast.LENGTH_LONG).show();
```

(continues on next page)

(continued from previous page)

```
                }
            });
            return false;
        }
       return true;
    }
    public void registerCallback(IPartnerAIDLServiceCallback cb) {
        // ★ポイント 2★ 利用元アプリの証明書がホワイトリストに登録されていることを確認する
      if (!checkPartner()) {
            return;
        }
      if (cb != null) mCallbacks.register(cb);
    }
    public String getInfo(String param) {

        // ★ポイント 2★ 利用元アプリの証明書がホワイトリストに登録されていることを確認する
      if (!checkPartner()) {
            return null;
        }
        // ★ポイント 4★ パートナーアプリからの Intent であっても、受信 Intent の安全性を確認する
        // サンプルにつき割愛。「3.2 入力データの安全性を確認する」を参照。
      Message msg = new Message();
      msg.what = GETINFO_MSG;
      msg.obj = String.format("パートナーアプリからのメソッド呼び出し。「%s」を受信した。", param);
      PartnerAIDLService.this.mHandler.sendMessage(msg);

        // ★ポイント 5★ パートナーアプリに開示してよい情報に限り返送してよい
        return "パートナーアプリに開示してよい情報 (method from Service)";
    }

    public void unregisterCallback(IPartnerAIDLServiceCallback cb) {
        // ★ポイント 2★ 利用元アプリの証明書がホワイトリストに登録されていることを確認する
       if (!checkPartner()) {
            return;
        }

       if (cb != null) mCallbacks.unregister(cb);
    }
};

@Override
public IBinder onBind(Intent intent) {
    // ★ポイント 3★ onBind で呼び出し元がパートナーかどうか判別できない
    // AIDL で定義したメソッドの呼び出し毎にチェックが必要になる。

    return mBinder;
}

@Override
public void onCreate() {
    Toast.makeText(this, this.getClass().getSimpleName() + " - onCreate()", Toast.LENGTH_SHORT).
↪show();

    // Service が実行中の間は、定期的にインクリメントした数字を通知する
    mHandler.sendEmptyMessage(REPORT_MSG);
```

(continues on next page)

(continued from previous page)

```
    }

    @Override
    public void onDestroy() {
        Toast.makeText(this, this.getClass().getSimpleName() + " - onDestroy()", Toast.LENGTH_SHORT).
↪show();

        // コールバックを全て解除する
        mCallbacks.kill();

        mHandler.removeMessages(REPORT_MSG);
    }

}
```

PkgCertWhitelists.java

```
package org.jssec.android.shared;

import android.content.pm.PackageManager;
import java.util.HashMap;
import java.util.Map;
import android.content.Context;
import android.os.Build;

import static android.content.pm.PackageManager.CERT_INPUT_SHA256;

public class PkgCertWhitelists {
    private Map<String, String> mWhitelists = new HashMap<String, String>();

    public boolean add(String pkgname, String sha256) {
        if (pkgname == null) return false;
        if (sha256 == null) return false;

        sha256 = sha256.replaceAll(" ", "");
        if (sha256.length() != 64) return false;    // SHA-256 は 32 バイト
        sha256 = sha256.toUpperCase();
        if (sha256.replaceAll("[0-9A-F]+", "").length() != 0) return false; // 0-9A-F 以外の文字がある

        mWhitelists.put(pkgname, sha256);
        return true;
    }

    public boolean test(Context ctx, String pkgname) {
        // pkgname に対応する正解のハッシュ値を取得する
        String correctHash = mWhitelists.get(pkgname);
        android.util.Log.d("Partner", "hash=" + correctHash);
        // pkgname の実際のハッシュ値と正解のハッシュ値を比較する
        if (Build.VERSION.SDK_INT >= 28) {
            // ★ API Level >= 28 では Package Manager の API で直接検証が可能
            PackageManager pm = ctx.getPackageManager();
            return pm.hasSigningCertificate(pkgname, hex2Bytes(correctHash), CERT_INPUT_SHA256);
        } else {
            // API Level < 28 の場合は PkgCert の機能を利用する
```

(continues on next page)

(continued from previous page)

```
            return PkgCert.test(ctx, pkgname, correctHash);
        }
    }

    private byte[] hex2Bytes(String s) {
        int len = s.length();
        byte[] data = new byte[len / 2];
        for (int i = 0; i < len; i += 2) {
            data[i / 2] = (byte) ((Character.digit(s.charAt(i), 16) << 4)
                    + Character.digit(s.charAt(i+1), 16));
        }
        return data;
    }
}
```

PkgCert.java

```
package org.jssec.android.shared;

import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

import android.content.Context;
import android.content.pm.PackageInfo;
import android.content.pm.PackageManager;
import android.content.pm.PackageManager.NameNotFoundException;
import android.content.pm.Signature;

public class PkgCert {

    public static boolean test(Context ctx, String pkgname, String correctHash) {
        if (correctHash == null) return false;
        correctHash = correctHash.replaceAll(" ", "");
        return correctHash.equals(hash(ctx, pkgname));
    }

    public static String hash(Context ctx, String pkgname) {
        if (pkgname == null) return null;
        try {
            PackageManager pm = ctx.getPackageManager();
            PackageInfo pkginfo = pm.getPackageInfo(pkgname, PackageManager.GET_SIGNATURES);
            if (pkginfo.signatures.length != 1) return null;    // 複数署名は扱わない
            Signature sig = pkginfo.signatures[0];
            byte[] cert = sig.toByteArray();
            byte[] sha256 = computeSha256(cert);
            return byte2hex(sha256);
        } catch (NameNotFoundException e) {
            return null;
        }
    }

    private static byte[] computeSha256(byte[] data) {
        try {
            return MessageDigest.getInstance("SHA-256").digest(data);
        } catch (NoSuchAlgorithmException e) {
```

(continues on next page)

(continued from previous page)

```
            return null;
        }
    }

    private static String byte2hex(byte[] data) {
        if (data == null) return null;
        final StringBuilder hexadecimal = new StringBuilder();
        for (final byte b : data) {
            hexadecimal.append(String.format("%02X", b));
        }
        return hexadecimal.toString();
    }
}
```

次にパートナー限定 Service を利用する Activity のサンプルコードを示す。

ポイント (Service を利用する)：

6. 利用先パートナー限定 Service アプリの証明書がホワイトリストに登録されていることを確認する
7. 利用先パートナー限定アプリに開示してよい情報に限り送信してよい
8. 明示的 Intent によりパートナー限定 Service を呼び出す
9. パートナー限定アプリからの結果情報であっても、受信 Intent の安全性を確認する

PartnerAIDLUserActivity.java

```
package org.jssec.android.service.partnerservice.aidluser;

import org.jssec.android.service.partnerservice.aidl.IPartnerAIDLService;
import org.jssec.android.service.partnerservice.aidl.IPartnerAIDLServiceCallback;
import org.jssec.android.service.partnerservice.aidl.R;
import org.jssec.android.shared.PkgCertWhitelists;
import org.jssec.android.shared.Utils;

import android.app.Activity;
import android.content.ComponentName;
import android.content.Context;
import android.content.Intent;
import android.content.ServiceConnection;
import android.os.Bundle;

import android.os.Handler;
import android.os.IBinder;
import android.os.Message;
import android.os.RemoteException;
import android.view.View;
import android.widget.Toast;

public class PartnerAIDLUserActivity extends Activity {

    private boolean mIsBound;
    private Context mContext;

    private final static int MGS_VALUE_CHANGED = 1;
```

(continues on next page)

(continued from previous page)

```
    // ★ポイント 6★ 利用先パートナー限定 Service アプリの証明書がホワイトリストに登録されていることを確認する
    private static PkgCertWhitelists sWhitelists = null;

    private static void buildWhitelists(Context context) {
        boolean isdebug = Utils.isDebuggable(context);
        sWhitelists = new PkgCertWhitelists();

        // パートナー限定 Service アプリ org.jssec.android.service.partnerservice.aidl の証明書ハッシュ値を登録
        sWhitelists.add("org.jssec.android.service.partnerservice.aidl", isdebug ?
                // debug.keystore の "androiddebugkey"の証明書ハッシュ値
                "0EFB7236 328348A9 89718BAD DF57F544 D5CCB4AE B9DB34BC 1E29DD26 F77C8255" :
                // keystore の "my company key"の証明書ハッシュ値
                "D397D343 A5CBC10F 4EDDEB7C A10062DE 5690984F 1FB9E88B D7B3A7C2 42E142CA");

        // 以下同様に他のパートナー限定 Service アプリを登録...
    }

    private static boolean checkPartner(Context context, String pkgname) {
        if (sWhitelists == null) buildWhitelists(context);
        return sWhitelists.test(context, pkgname);
    }

    // 利用先のパートナー限定 Activity に関する情報
    private static final String TARGET_PACKAGE =  "org.jssec.android.service.partnerservice.aidl";
    private static final String TARGET_CLASS = "org.jssec.android.service.partnerservice.aidl.
↪PartnerAIDLService";

    private static class ReceiveHandler extends Handler{
        private Context mContext;

        public ReceiveHandler(Context context){
            this.mContext = context;
        }

        @Override
        public void handleMessage(Message msg) {
            switch (msg.what) {
                case MGS_VALUE_CHANGED: {
                    String info = (String)msg.obj;
                    Toast.makeText(mContext, String.format("コールバックで「%s」を受信した。", info), Toast.
↪LENGTH_SHORT).show();
                    break;
                }
                default:
                    super.handleMessage(msg);
                    break;
            } // switch
        }

    }

    private final ReceiveHandler mHandler = new ReceiveHandler(this);

    // AIDL で定義したインターフェース。Service からの通知を受け取る。
```

(continues on next page)

(continued from previous page)

```
    private final IPartnerAIDLServiceCallback.Stub mCallback =
        new IPartnerAIDLServiceCallback.Stub() {
            @Override
            public void valueChanged(String info) throws RemoteException {
                Message msg = mHandler.obtainMessage(MGS_VALUE_CHANGED, info);
                mHandler.sendMessage(msg);
            }
    };

    // AIDL で定義したインターフェース。Service へ通知する。
    private IPartnerAIDLService mService = null;

    // Service と接続する時に利用するコネクション。bindService で実装する場合は必要になる。
    private ServiceConnection mConnection = new ServiceConnection() {

        // Service に接続された場合に呼ばれる
        @Override
        public void onServiceConnected(ComponentName className, IBinder service) {
            mService = IPartnerAIDLService.Stub.asInterface(service);

            try{
                // Service に接続
                mService.registerCallback(mCallback);

            }catch(RemoteException e){
                // Service が異常終了した場合
            }

            Toast.makeText(mContext, "Connected to service", Toast.LENGTH_SHORT).show();
        }

        // Service が異常終了して、コネクションが切断された場合に呼ばれる
        @Override
        public void onServiceDisconnected(ComponentName className) {
            Toast.makeText(mContext, "Disconnected from service", Toast.LENGTH_SHORT).show();
        }
    };

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        setContentView(R.layout.partnerservice_activity);

        mContext = this;
    }

    // サービス開始ボタン
    public void onStartServiceClick(View v) {
        // bindService を実行する
        doBindService();
    }

    // 情報取得ボタン
    public void onGetInfoClick(View v) {
```

(continues on next page)

(continued from previous page)

```
        getServiceinfo();
    }

    // サービス停止ボタン
    public void onStopServiceClick(View v) {
        doUnbindService();
    }

    @Override
    public void onDestroy() {
        super.onDestroy();
        doUnbindService();
    }

    /**
     * Service に接続する
     */
    private void doBindService() {
        if (!mIsBound){
            // ★ポイント 6★ 利用先パートナー限定 Service アプリの証明書がホワイトリストに登録されていることを確認
する
            if (!checkPartner(this, TARGET_PACKAGE)) {
                Toast.makeText(this, "利用先 Service アプリはホワイトリストに登録されていない。", Toast.LENGTH_
↪LONG).show();
                return;
            }

            Intent intent = new Intent();

            // ★ポイント 7★ 利用先パートナー限定アプリに開示してよい情報に限り送信してよい
            intent.putExtra("PARAM", "パートナーアプリに開示してよい情報");

            // ★ポイント 8★ 明示的 Intent によりパートナー限定 Service を呼び出す
            intent.setClassName(TARGET_PACKAGE, TARGET_CLASS);

          bindService(intent, mConnection, Context.BIND_AUTO_CREATE);
          mIsBound = true;
        }
    }

    /**
     * Service への接続を切断する
     */
    private void doUnbindService() {
        if (mIsBound) {
            // 登録していたレジスタがある場合は解除
            if(mService != null){
                try{
                    mService.unregisterCallback(mCallback);
                }catch(RemoteException e){
                    // Service が異常終了していた場合
                    // サンプルにつき処理は割愛

                }
            }
```

(continues on next page)

(continued from previous page)

```
            unbindService(mConnection);

            Intent intent = new Intent();

             // ★ポイント 8★ 明示的 Intent によりパートナー限定 Service を呼び出す
            intent.setClassName(TARGET_PACKAGE, TARGET_CLASS);

            stopService(intent);

            mIsBound = false;
        }
    }

    /**
     * Service から情報を取得する
     */
    void getServiceinfo() {
        if (mIsBound && mService != null) {
            String info = null;

            try {
                // ★ポイント 7★ 利用先パートナー限定アプリに開示してよい情報に限り送信してよい
                info = mService.getInfo("パートナーアプリに開示してよい情報 (method from activity)");
            } catch (RemoteException e) {
                e.printStackTrace();
            }
            // ★ポイント 9★ パートナー限定アプリからの結果情報であっても、受信 Intent の安全性を確認する
            // サンプルにつき割愛。「3.2 入力データの安全性を確認する」を参照。
            Toast.makeText(mContext, String.format("サービスから「%s」を取得した。", info), Toast.LENGTH_
↪SHORT).show();
         }
    }
}
```

PkgCertWhitelists.java

```
package org.jssec.android.shared;

import android.content.pm.PackageManager;
import java.util.HashMap;
import java.util.Map;
import android.content.Context;
import android.os.Build;

import static android.content.pm.PackageManager.CERT_INPUT_SHA256;

public class PkgCertWhitelists {
    private Map<String, String> mWhitelists = new HashMap<String, String>();

    public boolean add(String pkgname, String sha256) {
        if (pkgname == null) return false;
        if (sha256 == null) return false;

        sha256 = sha256.replaceAll(" ", "");
```

(continues on next page)

(continued from previous page)

```
        if (sha256.length() != 64) return false;    // SHA-256 は 32 バイト
        sha256 = sha256.toUpperCase();
        if (sha256.replaceAll("[0-9A-F]+", "").length() != 0) return false; // 0-9A-F 以外の文字がある

        mWhitelists.put(pkgname, sha256);
        return true;
    }

    public boolean test(Context ctx, String pkgname) {
        // pkgname に対応する正解のハッシュ値を取得する
        String correctHash = mWhitelists.get(pkgname);
        android.util.Log.d("Partner", "hash=" + correctHash);
        // pkgname の実際のハッシュ値と正解のハッシュ値を比較する
        if (Build.VERSION.SDK_INT >= 28) {
            // ★ API Level >= 28 では Package Manager の API で直接検証が可能
            PackageManager pm = ctx.getPackageManager();
            return pm.hasSigningCertificate(pkgname, hex2Bytes(correctHash), CERT_INPUT_SHA256);
        } else {
            // API Level < 28 の場合は PkgCert の機能を利用する
            return PkgCert.test(ctx, pkgname, correctHash);
        }
    }

    private byte[] hex2Bytes(String s) {
        int len = s.length();
        byte[] data = new byte[len / 2];
        for (int i = 0; i < len; i += 2) {
            data[i / 2] = (byte) ((Character.digit(s.charAt(i), 16) << 4)
                    + Character.digit(s.charAt(i+1), 16));
        }
        return data;
    }
}
```

PkgCert.java

```
package org.jssec.android.shared;

import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

import android.content.Context;
import android.content.pm.PackageInfo;
import android.content.pm.PackageManager;
import android.content.pm.PackageManager.NameNotFoundException;
import android.content.pm.Signature;

public class PkgCert {

    public static boolean test(Context ctx, String pkgname, String correctHash) {
        if (correctHash == null) return false;
        correctHash = correctHash.replaceAll(" ", "");
        return correctHash.equals(hash(ctx, pkgname));
    }
```

(continues on next page)

(continued from previous page)

```
    public static String hash(Context ctx, String pkgname) {
        if (pkgname == null) return null;
        try {
            PackageManager pm = ctx.getPackageManager();
            PackageInfo pkginfo = pm.getPackageInfo(pkgname, PackageManager.GET_SIGNATURES);
            if (pkginfo.signatures.length != 1) return null;    // 複数署名は扱わない
            Signature sig = pkginfo.signatures[0];
            byte[] cert = sig.toByteArray();
            byte[] sha256 = computeSha256(cert);
            return byte2hex(sha256);
        } catch (NameNotFoundException e) {
            return null;
        }
    }

    private static byte[] computeSha256(byte[] data) {
        try {
            return MessageDigest.getInstance("SHA-256").digest(data);
        } catch (NoSuchAlgorithmException e) {
            return null;
        }
    }

    private static String byte2hex(byte[] data) {
        if (data == null) return null;
        final StringBuilder hexadecimal = new StringBuilder();
        for (final byte b : data) {
            hexadecimal.append(String.format("%02X", b));
        }
        return hexadecimal.toString();
    }
}
```

4.4.1.4 自社限定 Service

自社限定 Service は、自社以外のアプリから利用されることを禁止する Service である。複数の自社製アプリでシステムを構成し、自社アプリが扱う情報や機能を守るために利用される。

以下、Messenger bind 型の Service を使用した例を示す。

ポイント（Service を作る）：

1. 独自定義 Signature Permission を定義する
2. 独自定義 Signature Permission を要求宣言する
3. Intent Filter を定義せず、exported=”true” を明示的に設定する
4. 独自定義 Signature Permission が自社アプリにより定義されていることを確認する
5. 自社アプリからの Intent であっても、受信 Intent の安全性を確認する
6. 利用元アプリは自社アプリであるから、センシティブな情報を返送してよい
7. 利用元アプリと同じ開発者鍵で APK を署名する

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="org.jssec.android.service.inhouseservice.messenger" >

    <!-- ★ポイント 1★ 独自定義 Signature Permission を定義する -->
    <permission
        android:name="org.jssec.android.service.inhouseservice.messenger.MY_PERMISSION"
        android:protectionLevel="signature" />

    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:allowBackup="false" >

        <!-- Messenger を利用した Service -->
         <!-- ★ポイント 2★ 独自定義 Signature Permission を要求宣言する -->
         <!-- ★ポイント 3★ Intent Filter を定義せず、exported="true"を明示的に設定する -->
        <service
            android:name="org.jssec.android.service.inhouseservice.messenger.InhouseMessengerService"
            android:exported="true"
            android:permission="org.jssec.android.service.inhouseservice.messenger.MY_PERMISSION" />
    </application>

</manifest>
```

InhouseMessengerService.java

```
package org.jssec.android.service.inhouseservice.messenger;

import org.jssec.android.shared.SigPerm;
import org.jssec.android.shared.Utils;

import java.util.ArrayList;
import java.util.Iterator;

import android.app.Service;
import android.content.Context;
import android.content.Intent;
import android.os.Bundle;
import android.os.Handler;
import android.os.IBinder;
import android.os.Message;
import android.os.Messenger;
import android.os.RemoteException;
import android.widget.Toast;

public class InhouseMessengerService extends Service{
    // 自社の Signature Permission
    private static final String MY_PERMISSION = "org.jssec.android.service.inhouseservice.messenger.MY_
↪PERMISSION";

    // 自社の証明書のハッシュ値
    private static String sMyCertHash = null;
    private static String myCertHash(Context context) {
        if (sMyCertHash == null) {
```

(continues on next page)

(continued from previous page)

```
            if (Utils.isDebuggable(context)) {
                // debug.keystoreの"androiddebugkey"の証明書ハッシュ値
                sMyCertHash = "0EFB7236 328348A9 89718BAD DF57F544 D5CCB4AE B9DB34BC 1E29DD26 F77C8255";
            } else {
                // keystoreの"my company key"の証明書ハッシュ値
                sMyCertHash = "D397D343 A5CBC10F 4EDDEB7C A10062DE 5690984F 1FB9E88B D7B3A7C2 42E142CA";
            }
        }
        return sMyCertHash;
    }

    // Serviceのクライアント（データ送信先）をリストで管理する
    private ArrayList<Messenger> mClients = new ArrayList<Messenger>();

    // クライアントからのデータを受信するときに利用するMessenger
    private final Messenger mMessenger = new Messenger(new ServiceSideHandler(mClients));

    // クライアントから受け取ったMessageを処理するHandler
    private static class ServiceSideHandler extends Handler{

        private ArrayList<Messenger> mClients;

        public ServiceSideHandler(ArrayList<Messenger> clients){
            this.mClients = clients;
        }

        @Override
        public void handleMessage(Message msg){

            switch(msg.what){
            case CommonValue.MSG_REGISTER_CLIENT:
                // クライアントから受け取ったMessengerを追加
                mClients.add(msg.replyTo);
                break;
            case CommonValue.MSG_UNREGISTER_CLIENT:
                mClients.remove(msg.replyTo);
                break;
            case CommonValue.MSG_SET_VALUE:
                // クライアントにデータを送る
                sendMessageToClients(mClients);
                break;
            default:
                super.handleMessage(msg);
               break;
            }
        }
    }

    /**
     * クライアントにデータを送る
     */
    private static void sendMessageToClients(ArrayList<Messenger> mClients){

        // ★ポイント6★ 利用元アプリは自社アプリであるから、センシティブな情報を返送してよい
        String sendValue = "センシティブな情報（from Service)";
```

(continues on next page)

(continued from previous page)

```
        // 登録されているクライアントへ、順番に送信する
        // ループ途中で remove しても全てのデータにアクセスしたいので Iterator を利用する
        Iterator<Messenger> ite = mClients.iterator();
        while(ite.hasNext()){
            try {
                Message sendMsg = Message.obtain(null, CommonValue.MSG_SET_VALUE, null);

                Bundle data = new Bundle();
                data.putString("key", sendValue);
                sendMsg.setData(data);

                Messenger next = ite.next();
                next.send(sendMsg);

            } catch (RemoteException e) {
                // クライアントが存在しない場合は、リストから取り除く
                ite.remove();
            }
        }
    }

    @Override
    public IBinder onBind(Intent intent) {

        // ★ポイント 4★ 独自定義 Signature Permission が自社アプリにより定義されていることを確認する
        if (!SigPerm.test(this, MY_PERMISSION, myCertHash(this))) {
            Toast.makeText(this, "独自定義 Signature Permission が自社アプリにより定義されていない。", Toast.
↪LENGTH_LONG).show();
            return null;
        }

        // ★ポイント 5★ 自社アプリからの Intent であっても、受信 Intent の安全性を確認する
        // サンプルにつき割愛。「3.2 入力データの安全性を確認する」を参照。
        String param = intent.getStringExtra("PARAM");
        Toast.makeText(this, String.format("パラメータ「%s」を受け取った。", param), Toast.LENGTH_LONG).
↪show();

        return mMessenger.getBinder();
    }

    @Override
    public void onCreate() {
        Toast.makeText(this, "Service - onCreate()", Toast.LENGTH_SHORT).show();
    }

    @Override
    public void onDestroy() {
        Toast.makeText(this, "Service - onDestroy()", Toast.LENGTH_SHORT).show();
    }
}
```

SigPerm.java

```
package org.jssec.android.shared;
```

(continues on next page)

(continued from previous page)

```
import android.content.Context;
import android.content.pm.PackageManager;
import android.content.pm.PackageManager.NameNotFoundException;
import android.content.pm.PermissionInfo;
import android.os.Build;

import static android.content.pm.PackageManager.CERT_INPUT_SHA256;

public class SigPerm {

    public static boolean test(Context ctx, String sigPermName, String correctHash) {
        if (correctHash == null) return false;
        correctHash = correctHash.replaceAll(" ", "");
        try{
            // sigPermName を定義したアプリのパッケージ名を取得する
            PackageManager pm = ctx.getPackageManager();
            PermissionInfo pi = pm.getPermissionInfo(sigPermName, PackageManager.GET_META_DATA);
            String pkgname = pi.packageName;
            // 非 Signature Permission の場合は失敗扱い
            if (pi.protectionLevel != PermissionInfo.PROTECTION_SIGNATURE) return false;
            // pkgname の実際のハッシュ値と正解のハッシュ値を比較する
            if (Build.VERSION.SDK_INT >= 28) {
                // ★ API Level >= 28 では Package Manager の API で直接検証が可能
                return pm.hasSigningCertificate(pkgname, Utils.hex2Bytes(correctHash), CERT_INPUT_
↪SHA256);
            } else {
                // API Level < 28 の場合は PkgCert を利用し、ハッシュ値を取得して比較する
                return correctHash.equals(PkgCert.hash(ctx, pkgname));
            }
        } catch (NameNotFoundException e){
            return false;
        }
    }
}
```

PkgCert.java

```
package org.jssec.android.shared;

import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

import android.content.Context;
import android.content.pm.PackageInfo;
import android.content.pm.PackageManager;
import android.content.pm.PackageManager.NameNotFoundException;
import android.content.pm.Signature;

public class PkgCert {

    public static boolean test(Context ctx, String pkgname, String correctHash) {
        if (correctHash == null) return false;
        correctHash = correctHash.replaceAll(" ", "");
        return correctHash.equals(hash(ctx, pkgname));
```

(continues on next page)

(continued from previous page)

```
    }

    public static String hash(Context ctx, String pkgname) {
        if (pkgname == null) return null;
        try {
            PackageManager pm = ctx.getPackageManager();
            PackageInfo pkginfo = pm.getPackageInfo(pkgname, PackageManager.GET_SIGNATURES);
            if (pkginfo.signatures.length != 1) return null;    // 複数署名は扱わない
            Signature sig = pkginfo.signatures[0];
            byte[] cert = sig.toByteArray();
            byte[] sha256 = computeSha256(cert);
            return byte2hex(sha256);
        } catch (NameNotFoundException e) {
            return null;
        }
    }

    private static byte[] computeSha256(byte[] data) {
        try {
            return MessageDigest.getInstance("SHA-256").digest(data);
        } catch (NoSuchAlgorithmException e) {
            return null;
        }
    }

    private static String byte2hex(byte[] data) {
        if (data == null) return null;
        final StringBuilder hexadecimal = new StringBuilder();
        for (final byte b : data) {
            hexadecimal.append(String.format("%02X", b));
        }
        return hexadecimal.toString();
    }
}
```

★ポイント 7 ★ APK を Export するときに、利用元アプリと同じ開発者鍵で APK を署名する。

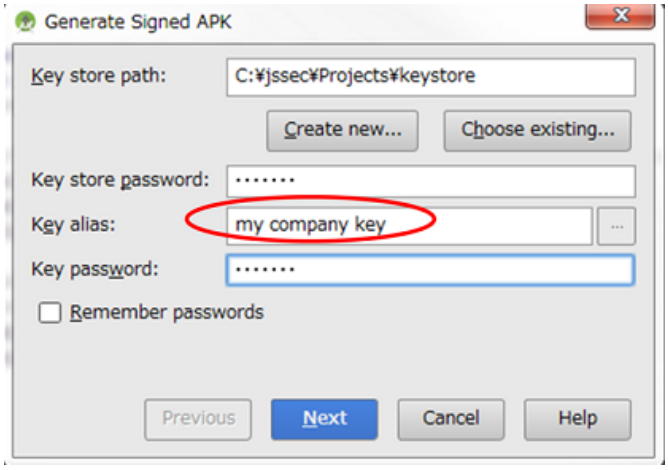

図 4.4.2　利用元アプリと同じ開発者鍵で APK を署名する

次に自社限定 Service を利用する Activity のサンプルコードを示す。

ポイント (Service を利用する)：

8. 独自定義 Signature Permission を利用宣言する
9. 独自定義 Signature Permission が自社アプリにより定義されていることを確認する
10. 利用先アプリの証明書が自社の証明書であることを確認する
11. 利用先アプリは自社アプリであるから、センシティブな情報を送信してもよい
12. 明示的 Intent により自社限定 Service を呼び出す
13. 自社アプリからの結果情報であっても、受信 Intent の安全性を確認する
14. 利用先アプリと同じ開発者鍵で APK を署名する

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="org.jssec.android.service.inhouseservice.messengeruser" >

    <!-- ★ポイント 8★ 独自定義 Signature Permission を利用宣言する -->
    <uses-permission
        android:name="org.jssec.android.service.inhouseservice.messenger.MY_PERMISSION" />

    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:allowBackup="false" >
        <activity
            android:name="org.jssec.android.service.inhouseservice.messengeruser.
↪InhouseMessengerUserActivity"
            android:label="@string/app_name"
            android:exported="true" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

InhouseMessengerUserActivity.java

```
package org.jssec.android.service.inhouseservice.messengeruser;

import org.jssec.android.shared.PkgCert;
import org.jssec.android.shared.SigPerm;
import org.jssec.android.shared.Utils;

import android.app.Activity;
import android.content.ComponentName;
import android.content.Context;
import android.content.Intent;
import android.content.ServiceConnection;
import android.os.Bundle;
```

(continues on next page)

(continued from previous page)

```
import android.os.Handler;
import android.os.IBinder;
import android.os.Message;
import android.os.Messenger;
import android.os.RemoteException;
import android.view.View;
import android.widget.Toast;

public class InhouseMessengerUserActivity extends Activity {

    private boolean mIsBound;
    private Context mContext;

    // 利用先の Activity 情報
    private static final String TARGET_PACKAGE =  "org.jssec.android.service.inhouseservice.messenger";
    private static final String TARGET_CLASS = "org.jssec.android.service.inhouseservice.messenger.
↪InhouseMessengerService";

    // 自社の Signature Permission
    private static final String MY_PERMISSION = "org.jssec.android.service.inhouseservice.messenger.MY_
↪PERMISSION";

    // 自社の証明書のハッシュ値
    private static String sMyCertHash = null;
    private static String myCertHash(Context context) {
        if (sMyCertHash == null) {
            if (Utils.isDebuggable(context)) {
                // debug.keystore の"androiddebugkey"の証明書ハッシュ値
                sMyCertHash = "0EFB7236 328348A9 89718BAD DF57F544 D5CCB4AE B9DB34BC 1E29DD26 F77C8255";
            } else {
                // keystore の"my company key"の証明書ハッシュ値
                sMyCertHash = "D397D343 A5CBC10F 4EDDEB7C A10062DE 5690984F 1FB9E88B D7B3A7C2 42E142CA";
            }
        }
        return sMyCertHash;
    }

    // Service からデータを受信するときに利用する Messenger
    private Messenger mServiceMessenger = null;

    // Service にデータを送信するときに利用する Messenger
    private final Messenger mActivityMessenger = new Messenger(new ActivitySideHandler());

    // Service から受け取った Message を処理する Handler
    private class ActivitySideHandler extends Handler {

        @Override
        public void handleMessage(Message msg) {
            switch (msg.what) {
                case CommonValue.MSG_SET_VALUE:
                    Bundle data = msg.getData();
                    String info = data.getString("key");
                    // ★ポイント 13★ 自社アプリからの結果情報であっても、値の安全性を確認する
                    // サンプルにつき割愛。「3.2 入力データの安全性を確認する」を参照。
                    Toast.makeText(mContext, String.format("サービスから「%s」を取得した。", info),
```

(continues on next page)

(continued from previous page)

```
                        Toast.LENGTH_SHORT).show();
                break;
            default:
                super.handleMessage(msg);
        }
    }
}

// Service と接続する時に利用するコネクション。bindService で実装する場合は必要になる。
private ServiceConnection mConnection = new ServiceConnection() {

    // Service に接続された場合に呼ばれる
    @Override
    public void onServiceConnected(ComponentName className, IBinder service) {
        mServiceMessenger = new Messenger(service);
        Toast.makeText(mContext, "Connect to service", Toast.LENGTH_SHORT).show();

        try {
            // Service に自分の Messenger を渡す
            Message msg = Message.obtain(null, CommonValue.MSG_REGISTER_CLIENT);

            msg.replyTo = mActivityMessenger;
            mServiceMessenger.send(msg);
        } catch (RemoteException e) {
            // Service が異常終了していた場合
        }
    }

    // Service が異常終了して、コネクションが切断された場合に呼ばれる
    @Override
    public void onServiceDisconnected(ComponentName className) {
        mServiceMessenger = null;
        Toast.makeText(mContext, "Disconnected from service", Toast.LENGTH_SHORT).show();
    }
};

@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    setContentView(R.layout.publicservice_activity);

    mContext = this;
}
// サービス開始ボタン
public void onStartServiceClick(View v) {
    // bindService を実行する
    doBindService();
}

// 情報取得ボタン
public void onGetInfoClick(View v) {
    getServiceinfo();
}
```

(continues on next page)

(continued from previous page)

```
    // サービス停止ボタン
    public void onStopServiceClick(View v) {
        doUnbindService();
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        doUnbindService();
    }



    /**
     * Service に接続する
     */
    void doBindService() {
        if (!mIsBound){
            // ★ポイント 9★ 独自定義 Signature Permission が自社アプリにより定義されていることを確認する
            if (!SigPerm.test(this, MY_PERMISSION, myCertHash(this))) {
                Toast.makeText(this, "独自定義 Signature Permission が自社アプリにより定義されていない。",␣
↪Toast.LENGTH_LONG).show();
                return;
            }

            // ★ポイント 10★ 利用先アプリの証明書が自社の証明書であることを確認する
            if (!PkgCert.test(this, TARGET_PACKAGE, myCertHash(this))) {
                Toast.makeText(this, "利用先サービスは自社アプリではない。", Toast.LENGTH_LONG).show();
                return;
            }

          Intent intent = new Intent();

            // ★ポイント 11★ 利用先アプリは自社アプリであるから、センシティブな情報を送信してもよい
          intent.putExtra("PARAM", "センシティブな情報");

            // ★ポイント 12★ 明示的 Intent により自社限定 Service を呼び出す
          intent.setClassName(TARGET_PACKAGE, TARGET_CLASS);

          bindService(intent, mConnection, Context.BIND_AUTO_CREATE);
          mIsBound = true;
        }
    }

    /**
     * Service への接続を切断する
     */
    void doUnbindService() {
        if (mIsBound) {
            unbindService(mConnection);
            mIsBound = false;
        }
    }

    /**
     * Service から情報を取得する
```

(continues on next page)

(continued from previous page)

```
     */
    void getServiceinfo() {
        if (mServiceMessenger != null) {
            try {
                  // 情報の送信を要求する
                Message msg = Message.obtain(null, CommonValue.MSG_SET_VALUE);
                mServiceMessenger.send(msg);
            } catch (RemoteException e) {
                // Service が異常終了していた場合
            }
         }
    }

}
```

SigPerm.java

```
package org.jssec.android.shared;

import android.content.Context;
import android.content.pm.PackageManager;
import android.content.pm.PackageManager.NameNotFoundException;
import android.content.pm.PermissionInfo;
import android.os.Build;

import static android.content.pm.PackageManager.CERT_INPUT_SHA256;

public class SigPerm {

    public static boolean test(Context ctx, String sigPermName, String correctHash) {
        if (correctHash == null) return false;
        correctHash = correctHash.replaceAll(" ", "");
        try{
            // sigPermName を定義したアプリのパッケージ名を取得する
            PackageManager pm = ctx.getPackageManager();
            PermissionInfo pi = pm.getPermissionInfo(sigPermName, PackageManager.GET_META_DATA);
            String pkgname = pi.packageName;
            // 非 Signature Permission の場合は失敗扱い
            if (pi.protectionLevel != PermissionInfo.PROTECTION_SIGNATURE) return false;
            // pkgname の実際のハッシュ値と正解のハッシュ値を比較する
            if (Build.VERSION.SDK_INT >= 28) {
                // ★ API Level >= 28 では Package Manager の API で直接検証が可能
                return pm.hasSigningCertificate(pkgname, Utils.hex2Bytes(correctHash), CERT_INPUT_
↪SHA256);
            } else {
                // API Level < 28 の場合は PkgCert を利用し、ハッシュ値を取得して比較する
                return correctHash.equals(PkgCert.hash(ctx, pkgname));
            }
        } catch (NameNotFoundException e){
            return false;
        }
    }
}
```

PkgCert.java

```java
package org.jssec.android.shared;

import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

import android.content.Context;
import android.content.pm.PackageInfo;
import android.content.pm.PackageManager;
import android.content.pm.PackageManager.NameNotFoundException;
import android.content.pm.Signature;

public class PkgCert {

    public static boolean test(Context ctx, String pkgname, String correctHash) {
        if (correctHash == null) return false;
        correctHash = correctHash.replaceAll(" ", "");
        return correctHash.equals(hash(ctx, pkgname));
    }

    public static String hash(Context ctx, String pkgname) {
        if (pkgname == null) return null;
        try {
            PackageManager pm = ctx.getPackageManager();
            PackageInfo pkginfo = pm.getPackageInfo(pkgname, PackageManager.GET_SIGNATURES);
            if (pkginfo.signatures.length != 1) return null;    // 複数署名は扱わない
            Signature sig = pkginfo.signatures[0];
            byte[] cert = sig.toByteArray();
            byte[] sha256 = computeSha256(cert);
            return byte2hex(sha256);
        } catch (NameNotFoundException e) {
            return null;
        }
    }

    private static byte[] computeSha256(byte[] data) {
        try {
            return MessageDigest.getInstance("SHA-256").digest(data);
        } catch (NoSuchAlgorithmException e) {
            return null;
        }
    }

    private static String byte2hex(byte[] data) {
        if (data == null) return null;
        final StringBuilder hexadecimal = new StringBuilder();
        for (final byte b : data) {
            hexadecimal.append(String.format("%02X", b));
        }
        return hexadecimal.toString();
    }
}
```

★ポイント 14 ★ APK を Export するときに、利用先アプリと同じ開発者鍵で APK を署名する。

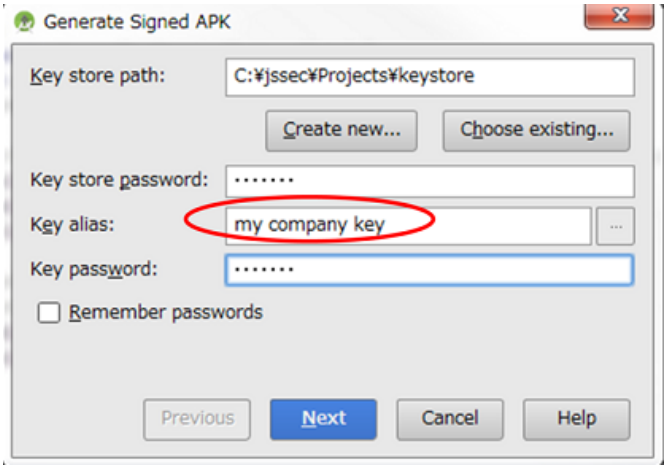

図 4.4.3　利用先アプリと同じ開発者鍵で APK を署名する

4.4.2 ルールブック

Service 実装時には以下のルールを守ること。

1. アプリ内でのみ使用する *Service* は非公開設定する （必須）
2. 受信データの安全性を確認する （必須）
3. 独自定義 *Signature Permission* は、自社アプリが定義したことを確認して利用する （必須）
4. 連携するタイミングで *Service* の機能を提供するかを判定する （必須）
5. 結果情報を返す場合には、返送先アプリからの結果情報漏洩に注意する （必須）
6. 利用先 *Service* が固定できる場合は明示的 *Intent* で *Service* を利用する （必須）
7. 他社の特定アプリと連携する場合は利用先 *Service* を確認する （必須）
8. 資産を二次的に提供する場合には、その資産の従来の保護水準を維持する （必須）
9. センシティブな情報はできる限り送らない （推奨）

4.4.2.1 アプリ内でのみ使用する Service は非公開設定する （必須）

アプリ内（または、同じ UID）でのみ使用される Service は非公開設定する。これにより、他のアプリから意図せず Intent を受け取ってしまうことがなくなり、アプリの機能を利用される、アプリの動作に異常をきたす等の被害を防ぐことができる。

実装上は AndroidManifest.xml で Service を定義する際に、exported 属性を false にするだけである。

```
AndroidManifest.xml
    <!-- 非公開 Service -->
    <!-- ★ポイント 1★ exported="false"により、明示的に非公開設定する -->
    <service android:name=".PrivateStartService" android:exported="false"/>
```

また、ケースは少ないと思われるが、同一アプリ内からのみ利用される Service であり、かつ Intent Filter を設置するような設計はしてはならない。Intent Filter の性質上、同一アプリ内の非公開 Service を呼び出すつもりでも、Intent Filter 経由で呼び出したときに意図せず他アプリの公開 Service を呼び出してしまう場合が存在するからである。

```
AndroidManifest.xml(非推奨)
    <!-- 非公開 Service -->
    <!-- ★ポイント 1★ exported="false"により、明示的に非公開設定する -->
    <service android:name=".PrivateStartService" android:exported="false">
        <intent-filter>
            <action android:name=" org.jssec.android.service.OPEN />
        </intent-filter>
    </service>
```

「4.4.3.1. *exported* 設定と *intent-filter* 設定の組み合わせ *(Service* の場合*)*」も参照すること。

4.4.2.2 受信データの安全性を確認する （必須）

Service も Activity と同様に、受信 Intent のデータを処理する際には、まず受信 Intent の安全性を確認しなければならない。Service を利用する側も Service からの結果 (として受信した）情報の安全性を確認する必要がある。Activity の「4.1.2.5. 受信 *Intent* の安全性を確認する （必須）」「4.1.2.9. 利用先 *Activity* からの戻り *Intent* の安全性を確認する （必須）」も参照すること。

Service においては、Intent 以外にもメソッドの呼び出しや Message によるデータの送受信などがあるため、それぞれ注意して実装を行わなければならない。

「3.2. 入力データの安全性を確認する」を参照すること。

4.4.2.3 独自定義 **Signature Permission** は、自社アプリが定義したことを確認して利用する （必須）

自社アプリだけから利用できる自社限定 Service を作る場合、独自定義 Signature Permission により保護しなければならない。AndroidManifest.xml での Permission 定義、Permission 要求宣言だけでは保護が不十分であるため、「5.2. *Permission* と *Protection Level*」の「5.2.1.2. 独自定義の *Signature Permission* で自社アプリ連携する方法」を参照すること。

4.4.2.4 連携するタイミングで **Service** の機能を提供するかを判定する （必須）

Intent パラメータの確認や独自定義 Signature Permission の確認といったセキュリティチェックを onCreate に入れてはいけない。その理由は、Service が起動中に新しい要求を受けたときに onCreate の処理が実施されないためである。したがって、startService によって開始される Service を実装する場合は、onStartCommand（IntentService を利用する場合は onHandleIntent）で判定を行わなければならない。bindService で開始する Service を実装する場合も同様のことが言えるので、onBind で判定をしなければならない。

4.4.2.5 結果情報を返す場合には、返送先アプリからの結果情報漏洩に注意する （必須）

Service のタイプによって結果情報の返送先（コールバックの呼び出し先や Message の送信先）アプリの信用度が異なる。返送先がマルウェアである可能性も考慮して十分に情報漏洩に対する配慮をしなければならない。

詳細は、Activity の「4.1.2.7. 結果情報を返す場合には、返送先アプリからの結果情報漏洩に注意する （必須）」を参照すること。

4.4.2.6 利用先 **Service** が固定できる場合は明示的 **Intent** で **Service** を利用する （必須）

暗黙的 Intent により Service を利用すると、Intent Filter の定義が同じ場合には先にインストールした Service に Intent が送信されてしまう。もし意図的に同じ Intent Filter を定義したマルウェアが先にインストールされていた場合、マルウェアに Intent が送信されてしまい、情報漏洩が生じる。一方、明示的 Intent により Service を利用すると、指定した Service 以外が Intent を受信することはなく比較的安全である。

ただし、別途考慮すべき点があるので、Activity の「4.1.2.8. 利用先 *Activity* が固定できる場合は明示的 *Intent* で *Activity* を利用する （必須）」を参照すること。

4.4.2.7 他社の特定アプリと連携する場合は利用先 **Service** を確認する （必須）

他社の特定アプリと連携する場合にはホワイトリストによる確認方法がある。自アプリ内に利用先アプリの証明書ハッシュを予め保持しておく。利用先の証明書ハッシュと保持している証明書ハッシュが一致するかを確認することで、なりすましアプリに Intent を発行することを防ぐことができる。具体的な実装方法についてはサンプルコードセクション「4.4.1.3. パートナー限定 *Service*」を参照すること。

4.4.2.8 資産を二次的に提供する場合には、その資産の従来の保護水準を維持する （必須）

Permission により保護されている情報資産および機能資産を他のアプリに二次的に提供する場合には、提供先アプリに対して同一の Permission を要求するなどして、その保護水準を維持しなければならない。Android の Permission セキュリティモデルでは、保護された資産に対するアプリからの直接アクセスについてのみ権限管理を行う。この仕様上の特性により、アプリに取得された資産がさらに他のアプリに、保護のために必要な Permission を要求することなく提供される可能性がある。このことは Permission を再委譲していることと実質的に等価なので、Permission の再委譲問題と呼ばれる。「5.2.3.4. *Permission* の再委譲問題」を参照すること。

4.4.2.9 センシティブな情報はできる限り送らない （推奨）

不特定多数のアプリと連携する場合にはセンシティブな情報を送ってはならない。

センシティブな情報を Service と受け渡しする場合、その情報の漏洩リスクを検討しなければならない。公開 Service に送付した情報は必ず漏洩すると考えなければならない。またパートナー限定 Service や自社限定 Service に送付した情報もそれら Service の実装に依存して情報漏洩リスクの大小がある。

センシティブな情報はできるだけ送付しないように工夫すべきである。送付する場合も、利用先 Service は信頼できる Service に限定し、情報が LogCat などに漏洩しないように配慮しなければならない。

4.4.3 アドバンスト

4.4.3.1 **exported** 設定と **intent-filter** 設定の組み合わせ **(Service** の場合**)**

このガイド文書では、Service の用途から非公開 Service、公開 Service、パートナー限定 Service、自社限定 Service の 4 タイプの Service について実装方法を述べている。各タイプに許されている AndroidManifest.xml の exported 属性と intent-filter 要素の組み合わせを次の表にまとめた。作ろうとしている Service のタイプと exported 属性および intent-filter 要素の対応が正しいことを確認すること。

表 4.4.2　exported 属性と intent-filter 要素の組み合わせ

	exported 属性の値		
	true	**false**	無指定
intent-filter 定義がある	公開	（使用禁止）	（使用禁止）
intent-filter 定義がない	公開、パートナー限定、自社限定	非公開	（使用禁止）

Service の exported 属性が無指定である場合にその Service が公開されるか非公開となるかは、intent-filter の定義の有無により決まるが[*13]、本ガイドでは Service の exported 属性を「無指定」にすることを禁止している。前述のような API のデフォルトの挙動に頼る実装をすることは避けるべきであり、exported 属性のようなセキュリティ上重要な設定を明示的に有効化する手段があるのであればそれを利用すべきであると考えられるためである。

「intent-filter 定義がある」&「exported=”false”」を使用禁止にしているのは、Android の振る舞いとして、同一アプリ内の非公開 Service を呼び出したつもりでも、意図せず他アプリの公開 Service を呼び出してしまう場合が存在するためである。

具体的には、Android は以下のような振る舞いをするのでアプリ設計時に検討が必要である。

- 複数の Service で同じ内容の intent-filter を定義した場合、先にインストールしたアプリ内の Service の定義が優先される
- 暗黙的 Intent を使った場合は、OS によって優先の Service が自動的に選ばれて、呼び出される。

以下の 3 つの図で Android の振る舞いによる意図せぬ呼び出しが起こる仕組みを説明する。図 4.4.4 は、同一アプリ内からしか非公開 Service(アプリ A) を暗黙的 Intent で呼び出せない正常な動作の例である。Intent-filter(図中 action=”X”) を定義しているのが、アプリ A しかいないので意図通りの動きとなっている。

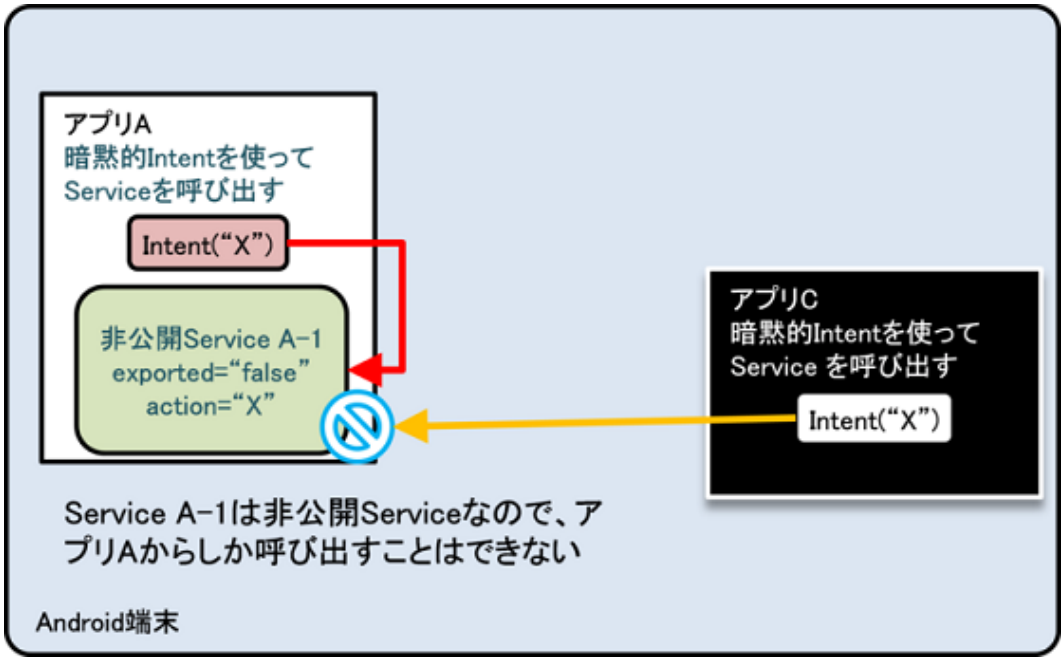

図 4.4.4　正常な動作の例

図 4.4.5 および 図 4.4.6 は、アプリ A に加えてアプリ B でも同じ intent-filter(図中 action=”X”) を定義している場合である。

[*13] intent-filter が定義されていれば公開 Service、定義されていなければ非公開 Service となる。
https://developer.android.com/guide/topics/manifest/service-element#exported

図 4.4.5 は、アプリ A→ アプリ B の順でインストールされた場合である。この場合、アプリ C が暗黙的 Intent を送信すると、非公開の Service(A-1) を呼び出そうとして失敗する。一方、アプリ A は暗黙的 Intent を使って意図通りに同一アプリ内の非公開 Service を呼び出せるので、セキュリティの (マルウェア対策の) 面では問題は起こらない。

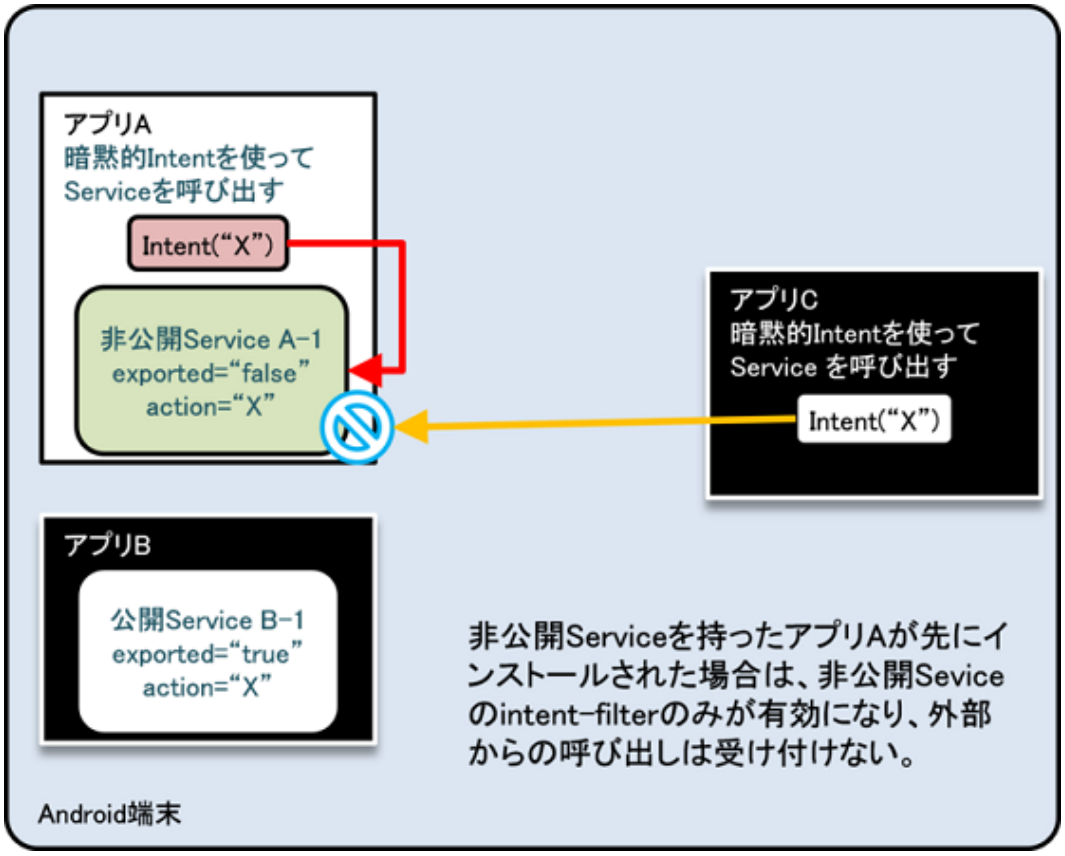

図 4.4.5　アプリ A → アプリ B の順でインストールされた場合

図 4.4.6 は、アプリ B→ アプリ A の順でインストールされた場合であり、セキュリティ面からみて問題がある。アプリ A が暗黙的 Intent を送信して同一アプリ内の非公開 Service を呼び出そうとするが、先にインストールしたアプリ B の公開 Activity(B-1) が呼び出されてしまう例を示している。これによりアプリ A からアプリ B に対してセンシティブな情報を送信する可能性が生じてしまう。アプリ B がマルウェアであれば、そのままセンシティブな情報の漏洩に繋がる。

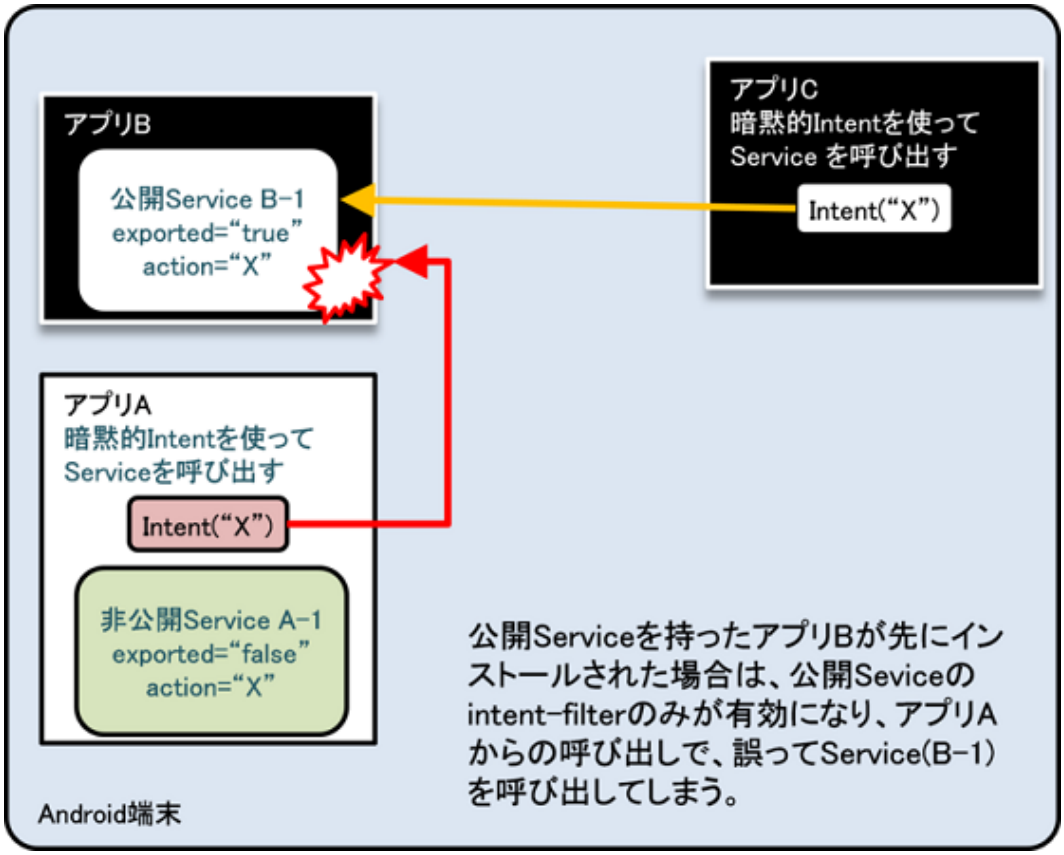

図 4.4.6　アプリ B → アプリ A の順でインストールされた場合

このように、Intent Filter を用いた非公開 Service の暗黙的 Intent 呼び出しは、意図せぬアプリの呼び出しや意図せぬアプリへのセンシティブな情報の送信を避けるためにも行うべきではない。

4.4.3.2 Service の実装方法について

Service の実装方法は多様であり、サンプルコードで分類したセキュリティ上のタイプとの相性もあるため簡単に特徴を示す。startService を利用する場合と bindService を利用する場合とに大きく分かれるが、startService と bindService の両方で利用できる Service を作成することも可能である。Service の実装方法を決定するために、次のような項目について検討を行うことになる。

- Service を別アプリに公開するか（Service の公開）
- 実行中にデータのやり取りを行うか (データの相互送受信)
- Service を制御するか（起動や終了など）
- 別プロセスとして実行するか（プロセス間通信）
- 複数の処理を同時に行うか (並行処理)

実装方法の分類と各々の項目の実現の可否を表にすると、表 4.4.3 のようになる。x は実現不可能かもしくは提供される機能とは別の枠組みが必要な場合を表す。

表 4.4.3: Service の実装方法の分類

分類	**Service** の公開	データの相互送受信	**Service** の制御 (起動・終了)	プロセス間通信	並行処理
startService 型	o	x	o	o	x
IntentService 型	o	x	x	o	x
localbind 型	x	o	o	x	x
Messen-gerbind 型	o	o	o	o	x
AIDLbind 型	o	o	o	o	o

startService 型

最も基本的な Service である。Service クラスを継承し、onStartCommand で処理を行う Service のことを指す。

利用する側は、Service を Intent で指定して startService を使用して呼び出す。Intent の送信元に対して、結果などのデータを直接返すことはできないため、Broadcast など別の方法を組み合わせて実現する必要がある。具体的な実装例は、「4.4.1.1. 非公開 *Service* を作る・利用する」を参照のこと。

セキュリティ上のチェックは onStartCommand で行う必要があるが、送信元のパッケージ名が取得できないためパートナー限定 Service には使用できない。

IntentService 型

IntentService は Service を継承して作られているクラスである。呼び出し方は、startService 型と同様である。通常の Service（startService 型）に比べて以下の特徴がある。

- Intent の処理は onHandleIntent で行う。（onStartCommand は使わない）
- 別スレッドで実行される
- 処理がキューイングされる

処理が別スレッドのため呼び出しは即座に返され、キューイング機構によりシーケンシャルに Intent に対する処理が行われる。各 Intent の並行処理はされないが、製品の要件によっては実装の簡素化の一つとして選択が可能である。Intent の送信元に対して、結果などのデータを直接返すことはできないため、Broadcast など別の方法を組み合わせて実現する必要がある。具体的な実装例は、「4.4.1.2. 公開 *Service* を作る・利用する」を参照のこと。

セキュリティ上のチェックは onHandleIntent で行う必要があるが、送信元のパッケージ名が取得できないためパートナー限定 Service には使用できない。

local bind 型

アプリと同じプロセス内でのみ動くローカル Service を実装するための方法を指す。Binder クラスから派生したクラスを定義して、Service で実装した機能（メソッド）を呼び出し元に提供できるようにする。

利用する側は、Service を Intent で指定して bindService を使用して呼び出す。Service を bind する方法の中では、最もシンプルな実装であるが、別プロセスでの起動や Service の公開ができないため用途は限定される。具体的な実装例は、サンプルコードに含まれるプロジェクト「Service PrivateServiceLocalBind」を参照のこと。

セキュリティ的には非公開 Service のみ実装可能である。

Messenger bind 型

Messenger の仕組みを利用して Service との連携を実現する方法を指す。

Service を利用する側からも Message の返信先として Messenger を渡すことができるため、双方でのデータのやり取りが比較的容易に実現可能である。また、処理はキューイングされるため、スレッドセーフに動作する特徴がある。各 Message の並行処理はされないが、製品の要件によっては実装の簡素化の一つとして選択が可能である。利用する側は、Service を Intent で指定して bindService を使用して呼び出す。具体的な実装例は、「4.4.1.4. 自社限定 *Service*」を参照のこと。

セキュリティ上のチェックは onBind や Message Handler で行う必要があるが、送信元のパッケージ名が取得できないためパートナー限定 Service には使用できない。

AIDL bind 型

AIDL の仕組みを利用して Service との連携を実現する方法を指す。AIDL によってインターフェースを定義し、Service の持つ機能をメソッドとして提供する。また、AIDL で定義したインターフェースを利用側で実装することで、コールバックを実現することもできる。マルチスレッド呼び出しは可能だが、排他処理はされないので Service 側で明示的に実装する必要がある。

利用する側は、Service を Intent で指定して bindService を使用して呼び出す。具体的な実装例は、「4.4.1.3. パートナー限定 *Service*」を参照のこと。

セキュリティ上のチェックは自社限定 Service では onBind で、パートナー限定 Service では AIDL で定義したインターフェースの各メソッドで行う必要がある。本文書で分類した全セキュリティタイプの Service に利用可能である。

4.5 SQLite を使う

本文書では SQLite を使用してデータベースの作成および操作を行う際にセキュリティ上で注意すべき点をまとめる。主なポイントは、データベースファイルのアクセス権の適切な設定と SQL インジェクションに対する対策である。ここでは、直接外部からデータベースファイルの読み書きを許す (複数アプリで共有する) ようなデータベースはここでは想定せず Content Provider のバックエンドやアプリ単体での使用を前提とする。また、ある程度センシティブな情報を扱っていることを想定しているが、そうでない場合も他アプリからの想定外の読み書きを避けるためにもここで挙げる対策を適用することをお勧めする。

4.5.1 サンプルコード

4.5.1.1 データベースの作成と操作

Android のアプリでデータベースを扱う場合、SQLiteOpenHelper を使用することでデータベースファイルの適切な配置およびアクセス権の設定（他のアプリがアクセスできない設定）ができる[*14]。ここでは、アプリ起動時にデータベースを作成し、UI 上からデータの検索・追加・変更・削除を行う簡単なアプリを例に、外部からの入力に対して不正な SQL が実行されないように SQL インジェクション対策したサンプルコードを示す。

[*14] ファイルの配置に関しては、SQLiteOpenHelper のコンストラクタの第 2 引数（name）にファイルの絶対パスも指定できる。そのため、誤って SD カードを直接指定した場合には他のアプリからの読み書きが可能になるので注意が必要である。

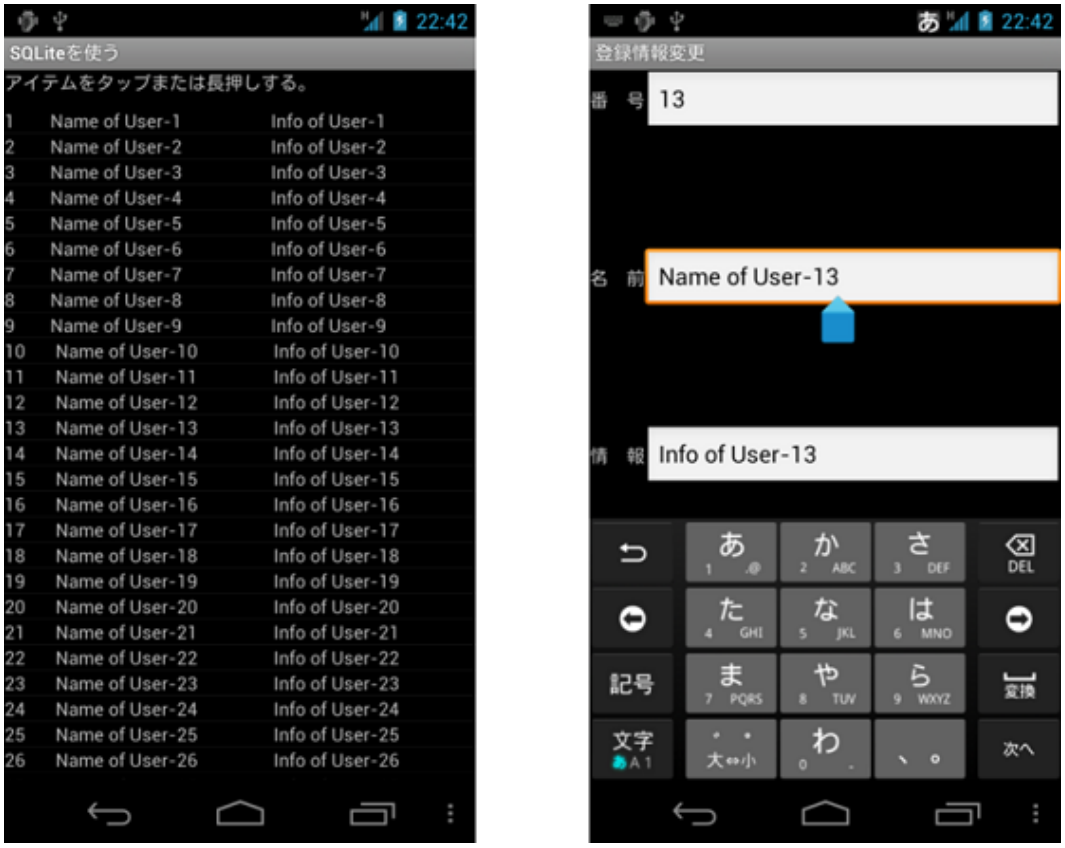

図 4.5.1　Android のアプリでデータベースを扱う

ポイント：

1. データベース作成には SQLiteOpenHelper を使用する
2. SQL インジェクションの対策として入力値を SQL 文に使用する場合にはプレースホルダを利用する
3. SQL インジェクションの保険的な対策としてアプリ要件に従って入力値をチェックする

SampleDbOpenHelper.java

```
package org.jssec.android.sqlite;

import android.content.Context;
import android.database.SQLException;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;
import android.util.Log;
import android.widget.Toast;

public class SampleDbOpenHelper extends SQLiteOpenHelper {
    private SQLiteDatabase           mSampleDb;              //取り扱うデータを格納するデータベース

    public static SampleDbOpenHelper newHelper(Context context)
    {
        //★ポイント 1★ DB 作成には SQLiteOpenHelper を使用する
        return new SampleDbOpenHelper(context);
    }

    public SQLiteDatabase getDb() {
        return mSampleDb;
```

(continues on next page)

(continued from previous page)

```
    }

    //Writable モードで DB を開く
    public void openDatabaseWithHelper() {
        try {
            if (mSampleDb != null && mSampleDb.isOpen()) {
                if (!mSampleDb.isReadOnly())// 既に読み書き可能でオープン済み
                    return;
                mSampleDb.close();
              }
            mSampleDb  = getWritableDatabase(); //この段階でオープンされる
        } catch (SQLException e) {
            //データベース構築に失敗した場合ログ出力
            Log.e(mContext.getClass().toString(), mContext.getString(R.string.DATABASE_OPEN_ERROR_
↪MESSAGE));
            Toast.makeText(mContext, R.string.DATABASE_OPEN_ERROR_MESSAGE, Toast.LENGTH_LONG).show();
        }
    }

    //ReadOnly モードで DB を開く
    public void openDatabaseReadOnly() {
        try {
            if (mSampleDb != null && mSampleDb.isOpen()) {
                if (mSampleDb.isReadOnly())// 既に ReadOnly でオープン済み
                    return;
                mSampleDb.close();
            }
            SQLiteDatabase.openDatabase(mContext.getDatabasePath(CommonData.DBFILE_NAME).getPath(), null,
↪ SQLiteDatabase.OPEN_READONLY);
        } catch (SQLException e) {
            //データベース構築に失敗した場合ログ出力
            Log.e(mContext.getClass().toString(), mContext.getString(R.string.DATABASE_OPEN_ERROR_
↪MESSAGE));
            Toast.makeText(mContext, R.string.DATABASE_OPEN_ERROR_MESSAGE, Toast.LENGTH_LONG).show();
        }
    }

    //Database Close
    public void closeDatabase() {
        try {
            if (mSampleDb != null && mSampleDb.isOpen()) {
                mSampleDb.close();
            }
        } catch (SQLException e) {
            //データベース構築に失敗した場合ログ出力
            Log.e(mContext.getClass().toString(), mContext.getString(R.string.DATABASE_CLOSE_ERROR_
↪MESSAGE));
            Toast.makeText(mContext, R.string.DATABASE_CLOSE_ERROR_MESSAGE, Toast.LENGTH_LONG).show();
        }
    }

    //Context を覚えておく
    private Context mContext;

    //テーブル作成コマンド
```

(continues on next page)

(continued from previous page)

```
    private static final String CREATE_TABLE_COMMANDS
            = "CREATE TABLE " + CommonData.TABLE_NAME + " ("
            + "_id INTEGER PRIMARY KEY AUTOINCREMENT, "
            + "idno INTEGER UNIQUE, "
            + "name VARCHAR(" + CommonData.TEXT_DATA_LENGTH_MAX + ") NOT NULL, "
            + "info VARCHAR(" + CommonData.TEXT_DATA_LENGTH_MAX + ")"
            + ");";

    public SampleDbOpenHelper(Context context) {
        super(context, CommonData.DBFILE_NAME, null, CommonData.DB_VERSION);
        mContext = context;
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
        try {
            db.execSQL(CREATE_TABLE_COMMANDS);  //DB構築コマンドの実行
        } catch (SQLException e) {
            //データベース構築に失敗した場合ログ出力
            Log.e(this.getClass().toString(), mContext.getString(R.string.DATABASE_CREATE_ERROR_
↪MESSAGE));
        }
    }

    @Override
    public void onUpgrade(SQLiteDatabase arg0, int arg1, int arg2) {
        // データベースのバージョンアップ時に実行される、データ移行などの処理を記述する
    }

}
```

DataSearchTask.java(SQLite Database プロジェクト)

```
package org.jssec.android.sqlite.task;

import org.jssec.android.sqlite.CommonData;
import org.jssec.android.sqlite.DataValidator;
import org.jssec.android.sqlite.MainActivity;
import org.jssec.android.sqlite.R;

import android.database.Cursor;
import android.database.SQLException;
import android.database.sqlite.SQLiteDatabase;
import android.os.AsyncTask;
import android.util.Log;


//データ検索タスク
public class DataSearchTask extends AsyncTask<String, Void, Cursor> {
    private MainActivity    mActivity;
    private SQLiteDatabase      mSampleDB;

    public DataSearchTask(SQLiteDatabase db, MainActivity activity) {
        mSampleDB = db;
        mActivity = activity;
```

(continues on next page)

(continued from previous page)

```
    }

    @Override
    protected Cursor doInBackground(String... params) {
        String  idno = params[0];
        String  name = params[1];
        String  info = params[2];
        String  cols[]  =   {"_id", "idno","name","info"};

        Cursor cur;

        //★ポイント 3★ アプリ要件に従って入力値をチェックする
        if (!DataValidator.validateData(idno, name, info))
        {
            return null;
        }

        //引数が全部 null だったら全件検索する)
        if ((idno == null || idno.length() == 0) &&
                (name == null || name.length() == 0) &&
                (info == null || info.length() == 0) ) {
            try {
                cur = mSampleDB.query(CommonData.TABLE_NAME, cols, null, null, null, null, null);
            } catch (SQLException e) {
                Log.e(DataSearchTask.class.toString(), mActivity.getString(R.string.SEARCHING_ERROR_
↪MESSAGE));
                return null;
            }
            return cur;
        }

        //No が指定されていたら No で検索
        if (idno != null && idno.length() > 0) {
            String selectionArgs[] = {idno};

            try {
                //★ポイント 2★ プレースホルダを使用する
                cur = mSampleDB.query(CommonData.TABLE_NAME, cols, "idno = ?", selectionArgs, null, null,
↪ null);
            } catch (SQLException e) {
                Log.e(DataSearchTask.class.toString(), mActivity.getString(R.string.SEARCHING_ERROR_
↪MESSAGE));
                return null;
            }
            return cur;
        }

        //Name が指定されていたら Name で完全一致検索
        if (name != null && name.length() > 0) {
            String selectionArgs[] = {name};
            try {
                //★ポイント 2★ プレースホルダを使用する
                cur = mSampleDB.query(CommonData.TABLE_NAME, cols, "name = ?", selectionArgs, null, null,
↪ null);
            } catch (SQLException e) {
```

(continues on next page)

(continued from previous page)

```
                Log.e(DataSearchTask.class.toString(), mActivity.getString(R.string.SEARCHING_ERROR_
↪MESSAGE));
                return null;
            }
            return cur;
        }

        //それ以外の場合は info を条件にして部分一致検索
        String argString = info.replaceAll("@", "@@"); //入力として受け取った info 内の$をエスケープ
        argString = argString.replaceAll("%", "@%"); //入力として受け取った info 内の % をエスケープ
        argString = argString.replaceAll("_", "@_"); //入力として受け取った info 内の_をエスケープ
        String selectionArgs[] = {argString};

        try {
            //★ポイント 2★ プレースホルダを使用する
            cur = mSampleDB.query(CommonData.TABLE_NAME, cols, "info LIKE '%' || ? || '%' ESCAPE '@'",␣
↪selectionArgs, null, null, null);
        } catch (SQLException e) {
            Log.e(DataSearchTask.class.toString(), mActivity.getString(R.string.SEARCHING_ERROR_
↪MESSAGE));
            return null;
        }
        return cur;
    }

    @Override
    protected void onPostExecute(Cursor resultCur) {
        mActivity.updateCursor(resultCur);
    }
}
```

DataValidator.java

```
package org.jssec.android.sqlite;

public class DataValidator {
    //入力値をチェックする
    //数字チェック
    public static boolean validateNo(String idno) {
        //null、空文字は OK
        if (idno == null || idno.length() == 0) {
           return true;
        }

        //数字であることを確認する
        try {
            if (!idno.matches("[1-9][0-9]*")) {
                //数字以外の時はエラー
                return false;
            }
        } catch (NullPointerException e) {
            //エラーを検出した
            return false;
        }
```

(continues on next page)

(continued from previous page)

```
        return true;
    }

    // 文字列の長さを調べる
    public static boolean validateLength(String str, int max_length) {
        //null、空文字は OK
        if (str == null || str.length() == 0) {
                return true;
        }

        //文字列の長さが MAX 以下であることを調べる
        try {
            if (str.length() > max_length) {
                //MAX より長い時はエラー
                return false;
            }
        } catch (NullPointerException e) {
            //バグ
            return false;
        }

        return true;
    }

   // 入力値チェック
    public static boolean validateData(String idno, String name, String info) {
        if (!validateNo(idno)) {
            return false;
         }
        if (!validateLength(name, CommonData.TEXT_DATA_LENGTH_MAX)) {
            return false;
         }else if (!validateLength(info, CommonData.TEXT_DATA_LENGTH_MAX)) {
            return false;
         }
        return true;
    }
}
```

4.5.2 ルールブック

SQLite を使用する際には以下のルールを守ること。

1. *DB* ファイルの配置場所、アクセス権を正しく設定する （必須）
2. 他アプリと *DB* データを共有する場合は *Content Provider* でアクセス制御する （必須）
3. *DB* 操作時に可変パラメータを扱う場合はプレースホルダを使用する （必須）)

4.5.2.1 DB ファイルの配置場所、アクセス権を正しく設定する （必須）

DB ファイルのデータの保護を考えた場合、DB ファイルの配置場所とアクセス権の設定は合わせて考慮すべき重要な要素である。

例えば、ファイルのアクセス権を正しく設定したつもりでも、SD カードなどアクセス権の設定を行えない場所に配置している場合には、誰からでもアクセス可能な DB ファイルになってしまう。また、アプリディレクトリに配置した場合でも、アクセス権を正しく設定しないと意図しないアクセスを許してしまうことになる。ここでは、配置場所とアクセス権設定について守るべき点を挙げた後、それを実現するための方法について説明する。

まず配置場所とアクセス権設定については、DB ファイル (データ) を保護する観点から考えると、以下の 2 点を実施する必要がある。

1. 配置場所

Context#getDatabasePath(String name) で取得できるファイルパスや場合によっては Context#getFilesDir で取得できるディレクトリの場所に配置する[*15]

2. アクセス権

MODE_PRIVATE（=ファイルを作成したアプリのみがアクセス可能）モードに設定する

この 2 点を実施することで、他のアプリからアクセスできない DB ファイルの作成を行うことができる。これらを実施するためには以下の方法が挙げられる。

1. SQLiteOpenHelper を使用する

2. Context#openOrCreateDatabase を使用する

DB ファイルの作成に際しては、SQLiteDatabase#openOrCreateDatabase を使用することもできる。しかし、このメソッドを使用した場合、Android スマートフォンの機種によっては、他のアプリから読み取り可能な DB ファイルが作成されることが分かっている。そのため、このメソッドの使用は避けて、他の方法を利用することを推奨する。上に挙げた 2 つの方法について、それぞれの特徴を以下で説明する。

SQLiteOpenHelper を使用する

SQLiteOpenHelper を使用する場合、開発者はあまり多くのことを考えなくてもよい。SQLiteOpenHelper を派生したクラスを作成し、コンストラクタの引数に DB の名前（ファイル名に使われる）[*16] を指定すれば、自動的に上記のセキュリティ要件を満たす DB ファイルを作成してくれる。

「4.5.1.1. データベースの作成と操作」に具体的な使用方法を示しているので参照すること。

Context#openOrCreateDatabase を使用する

Context#openOrCreateDatabase メソッドを使用して DB の作成を行う場合、ファイルのアクセス権をオプションで指定する必要があり、明示的に MODE_PRIVATE を指定する。

ファイルの配置に関しては、DB 名（ファイル名に使用される）の指定を SQLiteOpenHelper と同様に行えるので、自動的に前述のセキュリティ要件を満たすファイルパスにファイルが作成される。ただし、フルパスも指定できるので SD カードなどを指定した場合、MODE_PRIVATE を指定しても他アプリからアクセス可能になってしまうため注意が必要である。

DB に対して明示的にアクセス許可設定を行う例：MainActivity.java

[*15] どちらのメソッドも該当するアプリだけが読み書き権限を与えられ、他のアプリからはアクセスができないディレクトリ（パッケージディレクトリ）のサブディレクトリ以下のパスが取得できる。

[*16] （ドキュメントに記述はないが）SQLiteOpenHelper の実装では DB の名前にはファイルのフルパスを指定できるので、SD カードなどアクセス権の設定できない場所のパスが意図せず入力されないように注意が必要である。

```
public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        //データベースの構築
        try {
           //MODE_PRIVATE を設定して DB を作成
           db = Context.openOrCreateDatabase("Sample.db",
                                                 MODE_PRIVATE, null);
        } catch (SQLException e) {
            //データベース構築に失敗した場合ログ出力
            Log.e(this.getClass().toString(), getString(R.string.DATABASE_OPEN_ERROR_MESSAGE));
            return;
        }
        //省略 その他の初期化処理
    }
```

なお、アクセス権の設定は MODE_PRIVATE と合わせて以下の 3 種類があり、MODE_WORLD_READABLE と MODE_WORLD_WRITEABLE は OR 演算で同時指定することもできる。ただし、MODE_PRIVATE 以外は API Level 17 以降では deprecated となっており、API Level 24 以降ではセキュリティ例外が発生する。API Level 15 以降を対象とする場合でも、通常はこのフラグを使用しないことが望ましい[*17]。

- MODE_PRIVATE 作成アプリのみ読み書き可能
- MODE_WORLD_READABLE 作成アプリは読み書き可能、他は読み込みのみ
- MODE_WORLD_WRITEABLE 作成アプリは読み書き可能、他は書き込みのみ

4.5.2.2 他アプリと DB データを共有する場合は Content Provider でアクセス制御する （必須）

他のアプリと DB データを共有する手段として、DB ファイルを WORLD_READABLE、WORLD_WRITABLE として作成し、他のアプリから直接アクセスできるようにするという方法がある。しかし、この方法では DB にアクセスするアプリや DB への操作を制限できないため、意図しない相手（アプリ）にデータを読み書きされることもある。結果として、データの機密性や整合性に問題が生じたり、マルウェアの攻撃対象となったりする可能性も考えられる。

以上のことから、Android において DB データを他のアプリと共有する場合は、Content Provider を使うことを強くお勧めする。Content Provider を使うことにより、DB に対するアクセス制御を実現できるというセキュリティの観点からのメリットだけでなく、DB スキーマ構造を Content Provider 内に隠ぺいできるといった設計観点のメリットもある。

4.5.2.3 DB 操作時に可変パラメータを扱う場合はプレースホルダを使用する （必須）

SQL インジェクションを防ぐという意味で、任意の入力値を SQL 文に組み込む時はプレースホルダを使用するべきである。プレースホルダを使用した SQL の実行方法としては以下の 2 つの方法を挙げることができる。

1. SQLiteDatabase#compileStatement() を使用して SQLiteStatement を取得する。その後、SQLiteStatement#bindString()、bindLong() などを使用してパラメータをプレースホルダに配置する
2. SQLiteDatabese クラスの execSQL()、insert()、update()、delete()、query()、rawQuery()、replace() などを呼び出す際にプレースホルダを持った SQL 文を使用する

[*17] MODE_WORLD_READABLE および MODE_WORLD_WRITEABLE の性質と注意点については、「4.6.3.2. ディレクトリのアクセス権設定」を参照

なお、SQLiteDatabase#compileStatement() を使用して、SELECT コマンドを実行する場合、

「SELECT コマンドの結果として先頭の 1 要素 (1 行 1 列目) しか取得できない」

という制限があるので用途が限られる。

どちらの方式を使う場合でも、プレースホルダに与えるデータの内容は事前にアプリ要件に従ってチェックされていることが望ましい。以下で、それぞれの方法について説明する。

SQLiteDatabase#compileStatement() を使用する場合：

以下の手順でプレースホルダへデータを渡す。

1. SQLiteDatabase#compileStatement() を使用してプレースホルダを含んだ SQL 文を SQLiteStatement として取得する。
2. 作成した SQLiteStatement オブジェクトに対して、bindLong()、bindString() などのメソッドを使用してプレースホルダに設定する。
3. SQLiteStatement オブジェクトの execute() などのメソッドによって SQL を実行する。

プレースホルダ使用例：DataInsertTask.java（抜粋）

```
//データ追加タスク
public class DataInsertTask extends AsyncTask<String, Void, Void> {
    private MainActivity    mActivity;
    private SQLiteDatabase  mSampleDB;

    public DataInsertTask(SQLiteDatabase db, MainActivity activity) {
        mSampleDB = db;
        mActivity = activity;
    }

    @Override
    protected Void doInBackground(String... params) {
        String  idno = params[0];
        String  name = params[1];
        String  info = params[2];

        //★ポイント 3★ アプリケーション要件に従って入力値をチェックする
       if (!DataValidator.validateData(idno, name, info))
        {
            return null;
        }
        //データ追加処理
        //プレースホルダを使用する
        String commandString = "INSERT INTO " + CommonData.TABLE_NAME + " (idno, name, info) VALUES (?, ?
↪, ?)";
        SQLiteStatement sqlStmt = mSampleDB.compileStatement(commandString);
        sqlStmt.bindString(1, idno);
        sqlStmt.bindString(2, name);
        sqlStmt.bindString(3, info);
        try {
            sqlStmt.executeInsert();
        } catch (SQLException e) {
            Log.e(DataInsertTask.class.toString(), mActivity.getString(R.string.UPDATING_ERROR_MESSAGE));
```

(continues on next page)

(continued from previous page)

```
        } finally {
            sqlStmt.close();
        }
        return null;
    }

    // ～省略～
}
```

あらかじめ実行する SQL 文をオブジェクトとして作成しておきパラメータを当てはめる形である。実行する処理が確定しているので、SQL インジェクションが発生する余地はない。また、SQLiteStatement オブジェクトを再利用することで処理効率を高めることができるというメリットもある。

SQLiteDatabase が提供する各処理用のメソッドを使用する場合：

SQLiteDatabase が提供する DB 操作メソッドには、SQL 文を使用するものとそうでないものがある。SQL 文を使用するメソッドに SQLiteDatabase# execSQL()/rawQuery() などがあり、以下の手順で実行する。

1. プレースホルダを含んだ SQL 文を用意する。
2. プレースホルダに割り当てるデータを作成する。
3. SQL 文とデータを引数として渡して処理用メソッドを実行する。

一方、SQL 文を使用しないメソッドには、SQLiteDatabase#insert()/update()/delete()/query()/replace() などがある。これらを使用する場合には、以下の手順でデータを渡す。

1. DB に対して挿入/更新するデータがある場合には、ContentValues に登録する。
2. ContentValues を引数として渡して、各処理用メソッド（以下の例では SQLiteDatabase#insert()）を実行する。

各処理用メソッド（SQLiteDatabase#insert()）を使用する例

```
    private SQLiteDatabase  mSampleDB;
    private void addUserData(String idno, String name, String info) {

        //値の妥当性（型、範囲）チェック、エスケープ処理
        if (!validateInsertData(idno, name, info)) {
            //バリデーションを通過しなかった場合、ログ出力
            Log.e(this.getClass().toString(), getString(R.string.VALIDATION_ERROR_MESSAGE));
            return
        }

        //挿入するデータの準備
        ContentValues insertValues = new ContentValues();
        insertValues.put("idno", idno);
        insertValues.put("name", name);
        insertValues.put("info", info);

        //Insert 実行
        try {
            mSampleDb.insert("SampleTable", null, insertValues);
        } catch (SQLException e) {
            Log.e(this.getClass().toString(), getString(R.string.DB_INSERT_ERROR_MESSAGE));
```

(continues on next page)

(continued from previous page)

```
            return;
        }
    }
```

この例では、SQL コマンドを直接記述せず、SQLiteDatabase が提供する挿入用のメソッドを使用している。SQL コマンドを直接使用しないため、この方法も SQL インジェクションの余地はないと言える。

4.5.3 アドバンスト

4.5.3.1 SQL 文の LIKE 述語でワイルドカードを使用する際にエスケープ処理を施す

LIKE 述語のワイルドカード（%、_）を含む文字列をプレースホルダの入力値として使用した場合、そのままだとワイルドカードとして機能するため、必要に応じて事前にエスケープ処理を施す必要がある。必要なケースとしてはワイルドカードを単体の文字（”%” や”_”）として扱いたい場合が当てはまる。

実際のエスケープ処理は、以下のサンプルコードのように ESCAPE 句を使用して行うことができる。

LIKE を利用した場合のエスケープ処理の例

```
//データ検索タスク
public class DataSearchTask extends AsyncTask<String, Void, Cursor> {
    private MainActivity        mActivity;
    private SQLiteDatabase      mSampleDB;
    private ProgressDialog      mProgressDialog;

    public DataSearchTask(SQLiteDatabase db, MainActivity activity) {
        mSampleDB = db;
        mActivity = activity;
    }

    @Override
    protected Cursor doInBackground(String... params) {
        String  idno = params[0];
        String  name = params[1];
        String  info = params[2];
        String  cols[]  =   {"_id", "idno","name","info"};

        Cursor cur;

        // ～省略～

        //info を条件にして like 検索（部分一致）
        //ポイント：ワイルドカードに相当する文字はエスケープ処理する
        String argString = info.replaceAll("@", "@@"); //入力として受け取った info 内の@をエスケープ
        argString = argString.replaceAll("%", "@%"); //入力として受け取った info 内の %をエスケープ
        argString = argString.replaceAll("_", "@_"); //入力として受け取った info 内の_をエスケープ
        String selectionArgs[] = {argString};

        try {
            //ポイント：プレースホルダを使用する
            cur = mSampleDB.query("SampleTable", cols, "info LIKE '%' || ? || '%' ESCAPE '@'",
                                  selectionArgs, null, null, null);
        } catch (SQLException e) {
```

(continues on next page)

(continued from previous page)

```
            Toast.makeText(mActivity, R.string.SERCHING_ERROR_MESSAGE, Toast.LENGTH_LONG).show();
            return null;
        }
        return cur;
    }

    @Override
    protected void onPostExecute(Cursor resultCur) {
        mProgressDialog.dismiss();
        mActivity.updateCursor(resultCur);
    }
}
```

4.5.3.2 プレースホルダを使用できない SQL コマンドに対して外部入力を使う

テーブルの作成や削除などの DB オブジェクトを処理対象とした SQL 文を実行する場合、テーブル名などの値に対してプレースホルダを使うことはできない。基本的には、プレースホルダの使用できない値に対して、外部から入力された任意の文字列を使用するようなデータベースの設計はすべきでない。

仕様や機能上の制限でプレースホルダを使用できない場合は、入力値に危険が無いかどうか実行前に確認し、必要な処理を施すことが必須となる。

基本的には、

1. 文字列パラメータとして使用する場合、文字のエスケープやクォート処理を施す
2. 数値パラメータとして使用する場合、数字以外の文字が混入していないことを確認する
3. 識別子、コマンドとして使用する場合、1. に加え、使用できない文字が含まれていないことを確認する

を実施する。

> 参照：「安全な SQL の呼び出し方 (安全なウェブサイトの作り方別冊)」

4.5.3.3 不用意にデータベースの書き換えが行われないための対策を行う

SQLiteOpenHelper#getReadableDatabase、getWritableDatabase を使用して DB のインスタンスを取得した場合、どちらのメソッドを利用しても DB は読み書き可能な状態でオープンされる[*18]。また、Context#openOrCreateDatabase、SQLiteDatabase#openOrCreateDatabase なども同様である。

これは、アプリ操作や実装の不具合により意図せず DB の中身を書き換えてしまう（書き換えられてしまう）可能性を意味している。基本的にはアプリの仕様と実装の範囲で対応できると考えられるが、アプリの検索機能など、読み取りしか必要のない機能を実装する場合は、データベースを読み取り専用でオープンすることで、設計や検証の簡素化ひいてはアプリ品質の向上に繋がる場合があるので、状況に応じて検討をお勧めする。

具体的には、SQLiteDatabase#openDatabase に OPEN_READONLY を指定してデータベースをオープンする。

読み取り専用でデータベースをオープンする

[*18] getReableDatabase は基本的には getWritableDatabase で取得するのと同じオブジェクトを返す。ディスクフルなどの状況で書き込み可能オブジェクトを生成できない場合にリードオンリーのオブジェクトを返すという仕様である（getWritableDatabase はディスクフルなどの状況では実行エラーとなる）。

```
    // ～省略～

    // データベースのオープン（データベースは作成済みとする）
    SQLiteDatabase db
            = SQLiteDatabase.openDatabase(SQLiteDatabase.getDatabasePath("Sample.db"), null, OPEN_
↪READONLY);
```

参 照：https://developer.android.com/reference/android/database/sqlite/SQLiteOpenHelper.html - getReadableDatabase()

4.5.3.4 アプリの要件に従って DB の入出力データの妥当性をチェックする

SQLite は型に寛容なデータベースであり、DB 上で Integer として宣言されているカラムに対して文字型のデータを格納することが可能である。DB 内のデータは、数値型を含む全てのデータが平文の文字データとして DB 内に格納されている。このため、Integer 型のカラムに対して文字列型の検索（LIKE '%123%' など）を行うことも可能である。また、VARCHAR(100) のようにデータの最大長を記述してもそれ以上の長さのデータが入力可能であるなど、SQLite での値の制限（正当性確認）は期待できない。

このため、SQLite を使用するアプリは、このような DB の特性に注意して予期せぬデータを DB に格納したり取得したりしないようにアプリの要件に従って対処する必要がある。対処の方法としては次の 2 つがある。

1. データをデータベースに格納する際、型や長さなどの条件が一致しているか確認する
2. データベースから値を取得した際、データが想定外の型や長さでないか確認する

以下では、例として入力値が 1 以上の数字であることを検証するコードを示す。

例：入力データが 1 以上の数字であることを確認する（MainActivity.java より抜粋）

```
public class MainActivity extends Activity {

    // ～省略～

    //追加処理
    private void addUserData(String idno, String name, String info) {
        //No のチェック
        if (!validateNo(idno, CommonData.REQUEST_NEW)) {
            return;
        }

        //データ追加処理
        DataInsertTask task = new DataInsertTask(mSampleDb, this);
        task.execute(idno, name, info);
    }

    // ～省略～

    private boolean validateNo(String idno, int request) {
        if (idno == null || idno.length() == 0) {
            if (request == CommonData.REQUEST_SEARCH) {
                //検索処理の時は未指定を OK にする
                return true;
            } else {
                //検索処理以外の時は null、空文字はエラー
```

(continues on next page)

(continued from previous page)

```
                Toast.makeText(this, R.string.IDNO_EMPTY_MESSAGE, Toast.LENGTH_LONG).show();
                return false;
            }
        }

        //数字であることを確認する
        try {
            // 1 以上の値
            if (!idno.matches("[1-9][0-9]*")) {
                //数字以外の時はエラー
                Toast.makeText(this, R.string.IDNO_NOT_NUMERIC_MESSAGE, Toast.LENGTH_LONG).show();
                return false;
            }
        } catch (NullPointerException e) {
            //今回のケースではあり得ない
            return false;
        }

        return true;
    }

    // ～省略～
}
```

4.5.3.5 DB に格納するデータについての考察

SQLite では、データをファイルに格納する際に以下のような実装になっている。

- 数値型を含む全てのデータが平文の文字データとして DB ファイル内に格納される
- DB に対してデータの削除を行ってもデータ自体は DB ファイルから削除されない（削除マークが付くのみ）
- データを更新した場合も DB ファイル内には更新前のデータも削除されず残っている

よって、削除された「はず」の情報が DB ファイル内に残ったままの状態になっている可能性がある。この場合でも、本文書に従って対策を施し、Android のセキュリティ機能が有効であれば、他アプリを含む第三者からデータ・ファイルに直接アクセスされる心配はない。ただし、root 権限を奪取されるなど Android の保護機構を迂回してファイルを抜き出される可能性を考えると、ビジネスに大きな影響を与えるデータが格納されている場合には、Android 保護機構に頼らないデータ保護も検討しなければならない。

これらの理由により、端末の root 権限が奪取された場合でも守る必要があるような重要なデータは SQLite の DB にそのまま格納すべきではない。どうしても重要なデータを格納せざるを得ない場合には暗号化したデータを格納する、DB 全体を暗号化する、などの対策が必要となる。

実際に暗号化が必要な場合、暗号化に使う鍵の扱いやコードの難読化など本文書の範囲を超える課題が多いので、現時点でビジネスインパクトの大きなデータを扱うアプリの開発には専門家への相談をお勧めする。

参考として「4.5.3.6. *[参考]SQLite データベースを暗号化する (SQLCipher for Android)*」に、データベースを暗号化するライブラリを紹介しておく。

4.5.3.6 [参考]SQLite データベースを暗号化する (SQLCipher for Android)

SQLCipher は Zetetic LLC が開発した、SQLite データベースの透過的な 256 ビット AES による暗号化を提供するものである。C 言語で実装された SQLite 拡張ライブラリであり、暗号化には OpenSSL を使用している。また Obj-C、Java、Python 等の言語用の API も提供されている。商用バージョンの他にオープンソース版 (Community version と呼ばれている) が存在し、BSD ライセンスで商用利用可能である。Windows や Linux、macOS など様々なプラットフォームに対応しており、モバイルの世界では Android の他、ノキア/QT、アップルの iOS などで広く使用されている。

このうち Android 用にパッケージングされたのが SQLCipher for Android である[*19]。公開されているソースコードからコンパイルして作成することもできるが、aar フォーマットのライブラリ (android-database-sqlcipher-xxxx.aar) としても配布されており、単に利用するのであればこちらの方が便利である[*20]。標準の SQLite の API の一部を SQLCipher に合わせて変更することで、開発者は通常と同じコーディングで暗号化されたデータベースを利用できるようになっている。ここでは aar フォーマットのライブラリの利用方法について簡単に紹介する。

参照：https://www.zetetic.net/sqlcipher/

使い方

Android Studio では以下の手順で SQLCipher の利用が可能になる。

1. アプリの libs ディレクトリに android-database-sqlcipher-3.5.9.aar を配置する (https://www.zetetic.net/sqlcipher/open-source/)
2. app/gradle に依存関係を記載

```
dependencies {
    :
    implementation 'net.zetetic:android-database-sqlcipher:3.5.9@aar'
    :
}
```

3. 通常の android.database.sqlite.* のかわりに、net.sqlcipher.database.* をインポートする (android.database.Cursor はそのまま変更なしに利用できる)
4. データベースを利用する前にライブラリをロード・初期化し、データベースをオープンする際にパスワードを指定する

下に掲げたコードは、データベースを利用するための初期化処理を行うものである。ある Activity がデータベースを利用する前に SQLCipherInitializer.Initialize() を呼ぶ事を想定している。最初に SQLiteDatabase.loadLibs(this) を呼び出し、必要なライブラリをロードし初期化している。また、SQLiteDatabase.openOrCreateDatabase() でデータベースをオープンする際に、パスワード を渡している。データベースはここで与えたパスワードをベースとして生成された暗号鍵を用いて暗号化される。この場合注意すべきなのは平テキストで作成されたデータベースを、後で暗号化したものにすることはできない点であり、データベースを作成する時点でパスワードを指定しなければならない。

```
package android.jssec.org.samplesqlcipher;

import android.content.Context;
// 通常の android.database.sqlite.* のかわりに、net.sqlcipher.database.* をインポートする
```

(continues on next page)

[*19] https://github.com/sqlcipher/android-database-sqlcipher

[*20] ここで xxxx はライブラリのバージョン番号で、本記事執筆時点の最新バージョンは 3.5.9 である。以下ではこのバージョンを想定して記載している。

(continued from previous page)

```
import net.sqlcipher.database.SQLiteDatabase;
import java.io.File;

public class SQLCipherInitializer {
    static SQLiteDatabase Initialize(Context ctx, String dbName, String password) {
        // DB を利用する前に必要なライブラリをロードし初期化する
        SQLiteDatabase.loadLibs(ctx);
        // パッケージにローカルの DB 用ディレクトリにデータベースファイルを作成する
        File databaseFile = ctx.getDatabasePath(dbName);
        // DB をオープンする際に暗号化用のパスワードを指定する
        return SQLiteDatabase.openOrCreateDatabase(databaseFile, password, null);
    }
}
```

上は SQLiteDatabase.openOrCreateDatabase() を用いた例であるが、SQLiteOpenHelper#getWritableDatabase() や SQLiteOpenHelper#getReadableDatabase() API が変更になっており、引数としてパスワードを渡すことができるようになっている。いずれの場合もパスワードに null を指定した場合はデータベースは暗号化されず通常の SQLite データベースが作成される。

そのほか注意すべき点として、Context#openOrCreateDatabase() が使えないことである。そのためデータベースファイルの保護モードを設定したりパッケージのローカルディレクトリにデータベースを作成することを強制することはできない。従って SQLiteDatabase.openOrCreateDatabase() でデータベースを作成する場合は、最低限上の例のように getDatabasePath() により自身のパッケージのデータベース用ディレクトリにデータベースを作成することが推奨される。一方 SQLiteOpenHelper のコンストラクタの API には変更はなく、これを用いると android.database.sqlite.SQLiteOpenHelper と同様にパッケージのローカルディレクトリ内にデータベースが作成される。

4.6 ファイルを扱う

Android のセキュリティ設計思想に従うと、ファイルは情報を永続化又は一時保存 (キャッシュ) する目的にのみ利用し、原則非公開にするべきである。アプリ間の情報交換はファイルを直接アクセスさせるのではなく、ファイル内の情報を Content Provider や Service といったアプリ間連携の仕組みによって交換するべきである。これによりアプリ間のアクセス制御も実現できる。

SD カード等の外部記憶デバイスは十分なアクセス制御ができないため、容量の大きなファイルを扱う場合や別の場所 (PC など) への情報の移動目的など、機能上どうしても必要な場合のみに使用を限定するべきである。基本的に外部記憶デバイス上にはセンシティブな情報を含んだファイルを配置してはならない。もしセンシティブな情報を外部記憶デバイス上のファイルに保存しなければならない場合は暗号化等の対策が必要になるが、ここでは言及しない。

4.6.1 サンプルコード

前述のようにファイルは原則非公開にするべきである。しかしながらさまざまな事情によって、他のアプリにファイルを直接読み書きさせるべきときもある。セキュリティの観点から分類したファイルの種類と比較を 表 4.6.1 に示す。ファイルの格納場所や他アプリへのアクセス許可の組み合わせにより 4 種類のファイルに分類している。以降ではこのファイルの分類ごとにサンプルコードを示し説明を加えていく。

表 4.6.1: セキュリティ観点によるファイルの分類と比較

ファイルの分類	他アプリへのアクセス許可	格納場所	概要
非公開ファイル	なし	アプリディレクトリ内	• アプリ内でのみ読み書きできる。 • センシティブな情報を扱うことができる。 • ファイルは原則このタイプにするべき。
読み取り公開ファイル	読み取り	アプリディレクトリ内	• 他アプリおよびユーザーも読み取り可能。 • アプリ外部に公開（閲覧）可能な情報を扱う。
読み書き公開ファイル	読み取り･書き込み	アプリディレクトリ内	• 他アプリおよびユーザーも読み書き可能。 • セキュリティの観点からもアプリ設計の観点からも使用は避けるべき。
外部記憶ファイル	読み取り･書き込み	SD カードなどの外部記憶装置	• アクセス権のコントロールができない。 • 他アプリやユーザーによるファイルの読み書き・削除が常に可能。 • 使用は必要最小限にするべき。 • 比較的容量の大きなファイルを扱うことができる。

4.6.1.1 非公開ファイルを扱う

同一アプリ内でのみ読み書きされるファイルを扱う場合であり、安全なファイルの使い方である。ファイルに格納する情報が公開可能かどうかに関わらず、できるだけファイルは非公開の状態で保持し、他アプリとの必要な情報のやり取りは別の Android の仕組み（Content Provider、Service) を利用して行うことを原則とする。

ポイント：

1. ファイルは、アプリディレクトリ内に作成する
2. ファイルのアクセス権は、他のアプリが利用できないようにプライベートモードにする
3. センシティブな情報を格納することができる
4. ファイルに格納する (された) 情報に対しては、その入手先に関わらず内容の安全性を確認する

PrivateFileActivity.java

```
package org.jssec.android.file.privatefile;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
```

(continues on next page)

(continued from previous page)

```
import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.TextView;

public class PrivateFileActivity extends Activity {

    private TextView mFileView;

    private static final String FILE_NAME = "private_file.dat";

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.file);

        mFileView = (TextView) findViewById(R.id.file_view);
    }

    /**
     * ファイルの作成処理
     *
     * @param view
     */
    public void onCreateFileClick(View view) {
        FileOutputStream fos = null;
        try {
            // ★ポイント 1★ ファイルは、アプリディレクトリ内に作成する
            // ★ポイント 2★ ファイルのアクセス権は、他のアプリが利用できないようにプライベートモードにする
            fos = openFileOutput(FILE_NAME, MODE_PRIVATE);

            // ★ポイント 3★ センシティブな情報を格納することができる
            // ★ポイント 4★ ファイルに格納する情報に対しては、その入手先に関わらず内容の安全性を確認する
            // サンプルにつき割愛。「3.2 入力データの安全性を確認する」を参照。

            fos.write(new String("センシティブな情報 (File Activity)\n").getBytes());
        } catch (FileNotFoundException e) {
            mFileView.setText(R.string.file_view);
        } catch (IOException e) {
            android.util.Log.e("PrivateFileActivity", "ファイルの作成に失敗しました");
        } finally {
            if (fos != null) {
                try {
                    fos.close();
                } catch (IOException e) {
                    android.util.Log.e("PrivateFileActivity", "ファイルの終了に失敗しました");
                }
            }
        }

        finish();
    }

    /**
     * ファイルの読み込み処理
```

(continues on next page)

(continued from previous page)

```
     *
     * @param view
     */
    public void onReadFileClick(View view) {
        FileInputStream fis = null;
        try {
            fis = openFileInput(FILE_NAME);

            byte[] data = new byte[(int) fis.getChannel().size()];

            fis.read(data);

            String str = new String(data);

            mFileView.setText(str);
        } catch (FileNotFoundException e) {
            mFileView.setText(R.string.file_view);
        } catch (IOException e) {
            android.util.Log.e("PrivateFileActivity", "ファイルの読込に失敗しました");
        } finally {
            if (fis != null) {
                try {
                    fis.close();
                } catch (IOException e) {
                    android.util.Log.e("PrivateFileActivity", "ファイルの終了に失敗しました");
                }
            }
        }
    }

    /**
     * ファイルの削除処理
     *
     * @param view
     */
    public void onDeleteFileClick(View view) {

        File file = new File(this.getFilesDir() + "/" + FILE_NAME);
        file.delete();

        mFileView.setText(R.string.file_view);
    }
}
```

PrivateUserActivity.java

```
package org.jssec.android.file.privatefile;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

import android.app.Activity;
import android.content.Intent;
```

(continues on next page)

(continued from previous page)

```
import android.os.Bundle;
import android.view.View;
import android.widget.TextView;

public class PrivateUserActivity extends Activity {

    private TextView mFileView;

    private static final String FILE_NAME = "private_file.dat";

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.user);
        mFileView = (TextView) findViewById(R.id.file_view);
    }

    private void callFileActivity() {
        Intent intent = new Intent();
        intent.setClass(this, PrivateFileActivity.class);

        startActivity(intent);
    }

    /**
     * ファイル Activity の呼び出し処理
     *
     * @param view
     */
    public void onCallFileActivityClick(View view) {
        callFileActivity();
    }

    /**
     * ファイルの読み込み処理
     *
     * @param view
     */
    public void onReadFileClick(View view) {
        FileInputStream fis = null;
        try {
            fis = openFileInput(FILE_NAME);

            byte[] data = new byte[(int) fis.getChannel().size()];

            fis.read(data);

            // ★ポイント 4★ ファイルに格納された情報に対しては、その入手先に関わらず内容の安全性を確認する
            // サンプルにつき割愛。「3.2 入力データの安全性を確認する」を参照。
            String str = new String(data);
            mFileView.setText(str);
        } catch (FileNotFoundException e) {
            mFileView.setText(R.string.file_view);
        } catch (IOException e) {
            android.util.Log.d("PrivateFileActivity", "ファイルの読込に失敗しました");
```

(continues on next page)

(continued from previous page)

```
        } finally {
            if (fis != null) {
                try {
                    fis.close();
                } catch (IOException e) {
                    android.util.Log.d("PrivateFileActivity", "ファイルの終了に失敗しました");
                }
            }
        }
    }

    /**
     * ファイルの追記処理
     *
     * @param view
     */
    public void onWriteFileClick(View view) {
        FileOutputStream fos = null;
        try {
            // ★ポイント 1★ ファイルは、アプリケーションディレクトリ内に作成する
            // ★ポイント 2★ ファイルのアクセス権は、他のアプリが利用できないようにプライベートモードにする
            fos = openFileOutput(FILE_NAME, MODE_APPEND);

            // ★ポイント 3★ センシティブな情報を格納することができる
            // ★ポイント 4★ ファイルに格納する情報に対しては、その入手先に関わらず内容の安全性を確認する
            // サンプルにつき割愛。「3.2 入力データの安全性を確認する」を参照。
            fos.write(new String("センシティブな情報 (User Activity)\n").getBytes());
        } catch (FileNotFoundException e) {
            mFileView.setText(R.string.file_view);
        } catch (IOException e) {
            android.util.Log.d("PrivateFileActivity", "ファイルの作成に失敗しました");
        } finally {
            if (fos != null) {
                try {
                    fos.close();
                } catch (IOException e) {
                    android.util.Log.d("PrivateFileActivity", "ファイルの終了に失敗しました");
                }
            }
        }

        callFileActivity();
    }
}
```

4.6.1.2 読み取り公開ファイルを扱う

不特定多数のアプリに対して内容を公開するためのファイルである。以下のポイントに気を付けて実装すれば、比較的安全なファイルの使い方になる。ただし、公開ファイルを作成するための、MODE_WORLD_READABLE 変数は API Level17 以降では deprecated となっており、API Level 24 以降ではセキュリティ例外が発生するため、Content Provider によるファイル共有方法が望ましい。

ポイント：

1. ファイルは、アプリディレクトリ内に作成する
2. ファイルのアクセス権は、他のアプリに対しては読み取り専用モードにする
3. センシティブな情報は格納しない
4. ファイルに格納する (された) 情報に対しては、その入手先に関わらず内容の安全性を確認する

PublicFileActivity.java

```
package org.jssec.android.file.publicfile.readonly;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.TextView;

public class PublicFileActivity extends Activity {

    private TextView mFileView;

    private static final String FILE_NAME = "public_file.dat";

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.file);

        mFileView = (TextView) findViewById(R.id.file_view);
    }

    /**
     * ファイルの作成処理
     *
     * @param view
     */
    public void onCreateFileClick(View view) {
        FileOutputStream fos = null;
        try {
            // ★ポイント 1★ ファイルは、アプリディレクトリ内に作成する
            // ★ポイント 2★ ファイルのアクセス権は、他のアプリに対しては読み取り専用モードにする
            //  (読み取り専用モードの MODE_WORLDREADABLE は API LEVEL 17 で deplicated となったため、
            //   極力使用せず、ContentProvider などによるデータのやり取りをすること)
            fos = openFileOutput(FILE_NAME, MODE_WORLD_READABLE);

            // ★ポイント 3★ センシティブな情報は格納しない
            // ★ポイント 4★ ファイルに格納する情報に対しては、その入手先に関わらず内容の安全性を確認する
            // サンプルにつき割愛。「3.2 入力データの安全性を確認する」を参照。
            fos.write(new String("センシティブでない情報 (Public File Activity)\n")
                    .getBytes());
        } catch (FileNotFoundException e) {
            mFileView.setText(R.string.file_view);
```

(continues on next page)

(continued from previous page)

```
        } catch (IOException e) {
            android.util.Log.e("PublicFileActivity", "ファイルの作成に失敗しました");
        } finally {
            if (fos != null) {
                try {
                    fos.close();
                } catch (IOException e) {
                    android.util.Log.e("PublicFileActivity", "ファイルの終了に失敗しました");
                }

            }
        }

        finish();
    }

    /**
     * ファイルの読み込み処理
     *
     * @param view
     */
    public void onReadFileClick(View view) {
        FileInputStream fis = null;
        try {
            fis = openFileInput(FILE_NAME);

            byte[] data = new byte[(int) fis.getChannel().size()];

            fis.read(data);

            String str = new String(data);

            mFileView.setText(str);
        } catch (FileNotFoundException e) {
            mFileView.setText(R.string.file_view);
        } catch (IOException e) {
            android.util.Log.e("PublicFileActivity", "ファイルの読込に失敗しました");
        } finally {
            if (fis != null) {
                try {
                    fis.close();
                } catch (IOException e) {
                    android.util.Log.e("PublicFileActivity", "ファイルの終了に失敗しました");
                }
            }
        }
    }

    /**
     * ファイルの削除処理
     *
     * @param view
     */
    public void onDeleteFileClick(View view) {
```

(continues on next page)

(continued from previous page)

```
        File file = new File(this.getFilesDir() + "/" + FILE_NAME);
        file.delete();

        mFileView.setText(R.string.file_view);
    }
}
```

PublicUserActivity.java

```
package org.jssec.android.file.publicuser.readonly;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

import android.app.Activity;
import android.content.ActivityNotFoundException;
import android.content.Context;
import android.content.Intent;
import android.content.pm.PackageManager.NameNotFoundException;
import android.os.Bundle;
import android.view.View;
import android.widget.TextView;

public class PublicUserActivity extends Activity {

    private TextView mFileView;

    private static final String TARGET_PACKAGE = "org.jssec.android.file.publicfile.readonly";
    private static final String TARGET_CLASS = "org.jssec.android.file.publicfile.readonly.
↪PublicFileActivity";

    private static final String FILE_NAME = "public_file.dat";

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.user);

        mFileView = (TextView) findViewById(R.id.file_view);
    }

    private void callFileActivity() {
        Intent intent = new Intent();
        intent.setClassName(TARGET_PACKAGE, TARGET_CLASS);

        try {
            startActivity(intent);
        } catch (ActivityNotFoundException e) {
            mFileView.setText("(File Activity がありませんでした)");
        }
    }
```

(continues on next page)

(continued from previous page)

```
    /**
     * ファイル Activity の呼び出し処理
     *
     * @param view
     */
    public void onCallFileActivityClick(View view) {
        callFileActivity();
    }

    /**
     * ファイルの読み込み処理
     *
     * @param view
     */
    public void onReadFileClick(View view) {
        FileInputStream fis = null;
        try {
            File file = new File(getFilesPath(FILE_NAME));
            fis = new FileInputStream(file);

            byte[] data = new byte[(int) fis.getChannel().size()];

            fis.read(data);

            // ★ポイント 4★ ファイルに格納された情報に対しては、その入手先に関わらず内容の安全性を確認する
            // サンプルにつき割愛。「3.2 入力データの安全性を確認する」を参照。
            String str = new String(data);

            mFileView.setText(str);
        } catch (FileNotFoundException e) {
            android.util.Log.e("PublicUserActivity", "ファイルがありません");
        } catch (IOException e) {
            android.util.Log.e("PublicUserActivity", "ファイルの読込に失敗しました");
        } finally {
            if (fis != null) {
                try {
                    fis.close();
                } catch (IOException e) {
                    android.util.Log.e("PublicUserActivity", "ファイルの終了に失敗しました");
                }
            }
        }
    }

    /**
     * ファイルの追記処理
     *
     * @param view
     */
    public void onWriteFileClick(View view) {
        FileOutputStream fos = null;
        boolean exception = false;
        try {
            File file = new File(getFilesPath(FILE_NAME));
            // 書き込みは失敗する。FileNotFoundException が発生
```

(continues on next page)

(continued from previous page)

```
            fos = new FileOutputStream(file, true);

            fos.write(new String("センシティブでない情報（Public User Activity）\n")
                    .getBytes());
        } catch (IOException e) {
            mFileView.setText(e.getMessage());
            exception = true;
        } finally {
            if (fos != null) {
                try {
                    fos.close();
                } catch (IOException e) {
                    exception = true;
                }
            }
        }


        if (!exception)
            callFileActivity();
    }

    private String getFilesPath(String filename) {
        String path = "";

        try {
            Context ctx = createPackageContext(TARGET_PACKAGE,
                    Context.CONTEXT_RESTRICTED);
            File file = new File(ctx.getFilesDir(), filename);
            path = file.getPath();
        } catch (NameNotFoundException e) {
            android.util.Log.e("PublicUserActivity", "ファイルがありません");
        }
        return path;
    }
}
```

4.6.1.3 読み書き公開ファイルを扱う

不特定多数のアプリに対して、読み書き権限を許可するファイルの使い方である。

不特定多数のアプリが読み書き可能ということは、マルウェアも当然内容の書き換えが可能であり、データの信頼性も安全性も全く保証されない。また、悪意のない場合でもファイル内のデータの形式や書き込みを行うタイミングなど制御が困難であり、そのようなファイルは機能面からも実用性が無いに等しい。

以上のように、セキュリティの観点からもアプリ設計の観点からも、読み書き公開ファイルを安全に運用することは不可能であり、読み書き公開ファイルの使用は避けなければならない。

ポイント：

1. 他アプリから読み書き可能なアクセス権を設定したファイルは作らない

4.6.1.4 外部記憶 (読み書き公開) ファイルを扱う

SD カードのような外部記憶デバイス上にファイルを格納する場合である。比較的容量の大きな情報を格納する (Web からダウンロードしたファイルを置くなどの) 場合や外部に情報を持ち出す (バックアップなどの) 場合に利用することが想定される。

「外部記憶 (読み書き公開) ファイル」は不特定多数のアプリに対して「読み取り公開ファイル」と同等の性質を持つ。さらに android.permission.WRITE_EXTERNAL_STORAGE Permission を利用宣言している不特定多数のアプリに対しては「読み書き公開ファイル」と同等の性質を持つ。そのため、外部記憶 (読み書き公開) ファイルの使用は必要最小限にとどめるべきである。

Android アプリの慣例として、バックアップファイルは外部記憶デバイス上に作成されることが多い。しかし外部記憶デバイス上のファイルは前述のようにマルウェアを含む他のアプリから改ざんや削除されてしまうリスクがある。ゆえにバックアップを出力するアプリでは「バックアップファイルは速やかに PC 等の安全な場所にコピーしてください」といった警告表示をするなど、アプリの仕様や設計面でのリスク最小化の工夫も必要となる。

ポイント：

1. センシティブな情報は格納しない
2. アプリ毎にユニークなディレクトリにファイルを配置する
3. ファイルに格納する (された) 情報に対しては、その入手先に関わらず内容の安全性を確認する
4. 利用側のアプリで書き込みを行わない仕様にする

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="org.jssec.android.file.externalfile" >

    <!-- android.permission.WRITE_EXTERNAL_STORAGE Permission を利用宣言する -->
    <!-- Android 4.4 (API Level 19) 以降では、外部ストレージのアプリデータ領域を
    読み書きする際に Permission が不要なため、maxSdkVersion を宣言する -->
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"
        android:maxSdkVersion="18"/>

    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:allowBackup="false" >
        <activity
            android:name=".ExternalFileActivity"
            android:label="@string/app_name"
            android:exported="true" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

ExternalFileActivity.java

```
package org.jssec.android.file.externalfile;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.TextView;

public class ExternalFileActivity extends Activity {

    private TextView mFileView;

    private static final String TARGET_TYPE = "external";

    private static final String FILE_NAME = "external_file.dat";

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.file);

        mFileView = (TextView) findViewById(R.id.file_view);
    }


    /**
     * ファイルの作成処理
     *
     * @param view
     */
    public void onCreateFileClick(View view) {
        FileOutputStream fos = null;
        try {
            // ★ポイント 1★ センシティブな情報は格納しない
            // ★ポイント 2★ アプリ毎にユニークなディレクトリにファイルを配置する
            File file = new File(getExternalFilesDir(TARGET_TYPE), FILE_NAME);
            fos = new FileOutputStream(file, false);

            // ★ポイント 3★ ファイルに格納する情報に対しては、その入手先に関わらず内容の安全性を確認する
            // サンプルにつき割愛。「3.2 入力データの安全性を確認する」を参照。
            fos.write(new String("センシティブでない情報 (External File Activity)\n")
                    .getBytes());
        } catch (FileNotFoundException e) {
            mFileView.setText(R.string.file_view);
        } catch (IOException e) {
            android.util.Log.e("ExternalFileActivity", "ファイルの読込に失敗しました");
        } finally {
            if (fos != null) {

                try {
```

(continues on next page)

(continued from previous page)

```
                fos.close();
            } catch (IOException e) {
                android.util.Log.e("ExternalUserActivity", "ファイルの終了に失敗しました");
            }
        }
    }

    finish();
}

/**
 * ファイルの読み込み処理
 *
 * @param view
 */
public void onReadFileClick(View view) {
    FileInputStream fis = null;
    try {
        File file = new File(getExternalFilesDir(TARGET_TYPE), FILE_NAME);
        fis = new FileInputStream(file);

        byte[] data = new byte[(int) fis.getChannel().size()];

        fis.read(data);

        // ★ポイント 3★ ファイルに格納された情報に対しては、その入手先に関わらず内容の安全性を確認する
        // サンプルにつき割愛。「3.2 入力データの安全性を確認する」を参照。
        String str = new String(data);

        mFileView.setText(str);
    } catch (FileNotFoundException e) {
        mFileView.setText(R.string.file_view);
    } catch (IOException e) {
        android.util.Log.e("ExternalFileActivity", "ファイルの読込に失敗しました");
    } finally {
        if (fis != null) {
            try {
                fis.close();
            } catch (IOException e) {
                android.util.Log.e("ExternalFileActivity", "ファイルの終了に失敗しました");
            }
        }
    }
}

/**
 * ファイルの削除処理
 *
 * @param view
 */
public void onDeleteFileClick(View view) {

    File file = new File(getExternalFilesDir(TARGET_TYPE), FILE_NAME);
    file.delete();
```

(continues on next page)

(continued from previous page)

```
        mFileView.setText(R.string.file_view);
    }
}
```

利用側のサンプルコード

ExternalUserActivity.java

```
package org.jssec.android.file.externaluser;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

import android.app.Activity;
import android.app.AlertDialog;
import android.content.ActivityNotFoundException;
import android.content.Context;
import android.content.DialogInterface;
import android.content.Intent;
import android.content.pm.PackageManager.NameNotFoundException;
import android.os.Bundle;
import android.view.View;
import android.widget.TextView;

public class ExternalUserActivity extends Activity {

    private TextView mFileView;

    private static final String TARGET_PACKAGE = "org.jssec.android.file.externalfile";
    private static final String TARGET_CLASS = "org.jssec.android.file.externalfile.ExternalFileActivity
↪";
    private static final String TARGET_TYPE = "external";

    private static final String FILE_NAME = "external_file.dat";

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.user);
        mFileView = (TextView) findViewById(R.id.file_view);
    }

    private void callFileActivity() {
        Intent intent = new Intent();
        intent.setClassName(TARGET_PACKAGE, TARGET_CLASS);

        try {
            startActivity(intent);
        } catch (ActivityNotFoundException e) {
            mFileView.setText("(File Activity がありませんでした)");
        }
    }

    /**
```

(continues on next page)

(continued from previous page)

```
     * ファイル Activity の呼び出し処理
     *
     * @param view
     */
    public void onCallFileActivityClick(View view) {
        callFileActivity();
    }

    /**
     * ファイルの読み込み処理
     *
     * @param view
     */
    public void onReadFileClick(View view) {
        FileInputStream fis = null;
        try {
            File file = new File(getFilesPath(FILE_NAME));
            fis = new FileInputStream(file);

            byte[] data = new byte[(int) fis.getChannel().size()];

            fis.read(data);

            // ★ポイント 3★ ファイルに格納された情報に対しては、その入手先に関わらず内容の安全性を確認する
            // サンプルにつき割愛。「3.2 入力データの安全性を確認する」を参照。
            String str = new String(data);

            mFileView.setText(str);
        } catch (FileNotFoundException e) {
            mFileView.setText(R.string.file_view);
        } catch (IOException e) {
            android.util.Log.e("ExternalUserActivity", "ファイルの読込に失敗しました");
        } finally {
            if (fis != null) {
                try {
                    fis.close();
                } catch (IOException e) {
                    android.util.Log.e("ExternalUserActivity", "ファイルの終了に失敗しました");
                }
            }
        }
    }

    /**
     * ファイルの追記処理
     *
     * @param view
     */
    public void onWriteFileClick(View view) {

        // ★ポイント 4★ 利用側のアプリで書き込みを行わない仕様にする
        // ただし、悪意のあるアプリが上書き・削除などを行うことを想定してアプリの設計を行うこと

        final AlertDialog.Builder alertDialogBuilder = new AlertDialog.Builder(
                this);
```

(continues on next page)

(continued from previous page)

```
        alertDialogBuilder.setTitle("ポイント 4");
        alertDialogBuilder.setMessage("利用側のアプリで書き込みを行わないこと");
        alertDialogBuilder.setPositiveButton("OK", new DialogInterface.OnClickListener() {

                    @Override
                    public void onClick(DialogInterface dialog, int which) {
                        callFileActivity();
                    }
                });

        alertDialogBuilder.create().show();

    }

    private String getFilesPath(String filename) {
        String path = "";

        try {
            Context ctx = createPackageContext(TARGET_PACKAGE,
                    Context.CONTEXT_IGNORE_SECURITY);
            File file = new File(ctx.getExternalFilesDir(TARGET_TYPE), filename);
            path = file.getPath();
        } catch (NameNotFoundException e) {
            android.util.Log.e("ExternalUserActivity", "ファイルがありません");
        }
        return path;
    }
}
```

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="org.jssec.android.file.externaluser" >

    <!-- Android 4.0.3 (API Level 14) 以降では、外部ストレージを読むための Permission が
    定義されたので利用宣言をする。実際には Android 4.4(API Level 19) 以降で
    他のアプリデータ領域を読む場合に必須となる -->
    <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"/>

    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:allowBackup="false" >

        <activity
            android:name=".ExternalUserActivity"
            android:label="@string/app_name"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
```

(continues on next page)

(continued from previous page)

```
    </application>
</manifest>
```

4.6.2 ルールブック

ファイルを扱う場合には以下のルールを守ること。

1. ファイルは原則非公開ファイルとして作成する （必須）
2. 他のアプリから読み書き権限でアクセス可能なファイルは作成しない （必須）
3. *SD* カードなど外部記憶デバイスに格納するファイルの利用は必要最小限にする （必須）
4. ファイルの生存期間を考慮してアプリの設計を行う （必須）

4.6.2.1 ファイルは原則非公開ファイルとして作成する （必須）

「4.6. ファイルを扱う」「4.6.1.1. 非公開ファイルを扱う」で述べたように、格納する情報の内容に関わらずファイルは原則非公開にするべきである。Android のセキュリティ設計の観点からも、情報のやり取りとそのアクセス制御は Content Provider や Service などの Android の仕組みの中で行うべきであり、できない事情がある場合のみファイルのアクセス権で代用することを検討することになる。

各ファイルタイプのサンプルコードや以下のルールの項も参照のこと。

4.6.2.2 他のアプリから読み書き権限でアクセス可能なファイルは作成しない （必須）

「4.6.1.3. 読み書き公開ファイルを扱う」で述べたように、他のアプリに対してファイルの読み書きを許可すると、ファイルに格納される情報の制御ができない。そのため、セキュリティ的な観点からも機能・設計的な観点からも読み書き公開ファイルを利用した情報の共有を考えるべきではない。

4.6.2.3 SD カードなど外部記憶デバイスに格納するファイルの利用は必要最小限にする （必須）

「4.6.1.4. 外部記憶 *(*読み書き公開*)* ファイルを扱う」で述べたように、SD カードをはじめとする外部記憶デバイスにファイルを置くことは、セキュリティおよび機能の両方の観点から潜在的な問題を抱えることに繋がる。一方で、SD カードはアプリディレクトリより生存期間の長いファイルを扱え、アプリ外部にデータを持ち出すのに常時使える唯一のストレージなので、アプリの仕様によっては使用せざるを得ないケースも多いと考えられる。

外部記憶デバイスにファイルを格納する場合、不特定多数のアプリおよびユーザーが読み・書き・削除できることを考慮して、サンプルコードで述べたポイントを含めて以下のようなポイントに気をつけてアプリの設計を行う必要がある。

- 原則としてセンシティブな情報は外部記憶デバイス上のファイルに保存しない
- もしセンシティブな情報を外部記憶デバイス上のファイルに保存する場合は暗号化する
- 他アプリやユーザーに改ざんされては困る情報を外部記憶デバイス上のファイルに保存する場合は電子署名も一緒に保存する
- 外部記憶デバイス上のファイルを読み込む場合、読み込むデータの安全性を確認してからデータを利用する

- 他のアプリやユーザーによって外部記憶デバイス上のファイルはいつでも削除されることを想定してアプリを設計しなければならない

「4.6.2.4. ファイルの生存期間を考慮してアプリの設計を行う （必須）」も参照すること。

4.6.2.4 ファイルの生存期間を考慮してアプリの設計を行う （必須）

アプリディレクトリに保存されたデータは以下のユーザー操作により消去される。アプリの生存期間と一致する、またはアプリの生存期間より短いのが特徴である。

- アプリのアンインストール
- 各アプリのデータおよびキャッシュの消去（「設定」→「アプリケーション」→「アプリケーションの管理」）

SD カード等の外部記憶デバイス上に保存されたファイルは、アプリの生存期間よりファイルの生存期間が長いことが特徴である。さらに次の状況も想定する必要がある。

- ユーザーによるファイルの消去
- SD カードの抜き取り・差し替え・アンマウント
- マルウェアによるファイルの消去

このようにファイルの保存場所によってファイルの生存期間が異なるため、本節で説明したようなセンシティブな情報を保護する観点だけでなく、アプリとして正しい動作を実現する観点でもファイルの保存場所を正しく選択する必要がある。

4.6.3 アドバンスト

4.6.3.1 ファイルディスクリプタ経由のファイル共有

他のアプリに公開ファイルを直接アクセスさせるのではなく、ファイルディスクリプタ経由でファイル共有する方法がある。Content Provider と Service でこの方法が使える。Content Provider や Service の中で非公開ファイルをオープンし、そのファイルディスクリプタを相手のアプリに渡す。相手アプリはファイルディスクリプタ経由でファイルを読み書きできる。

他のアプリにファイルを直接アクセスさせるファイル共有方法とファイルディスクリプタ経由のファイル共有方法の比較を 表 4.6.2 に示す。アクセス権のバリエーションとアクセス許可するアプリの範囲でメリットがある。特にアクセスを許可するアプリを細かく制御できるところがセキュリティ観点ではメリットが大きい。

表 4.6.2　アプリ間ファイル共有方法の比較

ファイル共有方法	アクセス権設定のバリエーション	アクセスを許可するアプリの範囲
他のアプリにファイルを直接アクセスさせるファイル共有	• 読み取り • 書き込み • 読み取り＋書き込み	すべてのアプリに対して一律アクセス許可してしまう
ファイルディスクリプタ経由のファイル共有	• 読み取り • 書き込み • 追記のみ • 読み取り＋書き込み • 読み取り＋追記のみ	Content Provider や Service にアクセスしてくるアプリに対して個別に一時的にアクセス許可・不許可を制御できる

上記ファイル共有方法のどちらにも共通することであるが、他のアプリにファイルの書き込みを許可するとファイル内容の完全性が保証しづらくなる。特に複数のアプリから同時に書き込みが行われると、ファイル内容のデータ構造が壊れてしまいアプリが正常に動作しなくなるリスクがある。他のアプリとのファイル共有においては、読み込み権限だけを許可するのが望ましい。

以下では、Content Provider でファイルを共有する実装例 (非公開 Provider の場合) をサンプルコードとして掲載する。

ポイント

1. 利用元アプリは自社アプリであるから、センシティブな情報を保存してよい
2. 自社限定 Content Provider アプリからの結果であっても、結果データの安全性を確認する

InhouseProvider.java

```
package org.jssec.android.file.inhouseprovider;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

import org.jssec.android.shared.SigPerm;
import org.jssec.android.shared.Utils;

import android.content.ContentProvider;
import android.content.ContentValues;
import android.content.Context;
import android.database.Cursor;
import android.net.Uri;
import android.os.ParcelFileDescriptor;

public class InhouseProvider extends ContentProvider {

    private static final String FILENAME = "sensitive.txt";

    // 自社の Signature Permission
    private static final String MY_PERMISSION = "org.jssec.android.file.inhouseprovider.MY_PERMISSION";
```

(continues on next page)

(continued from previous page)

```
    // 自社の証明書のハッシュ値
    private static String sMyCertHash = null;

    private static String myCertHash(Context context) {
        if (sMyCertHash == null) {
            if (Utils.isDebuggable(context)) {
                // debug.keystore の"androiddebugkey"の証明書ハッシュ値
                sMyCertHash = "0EFB7236 328348A9 89718BAD DF57F544 D5CCB4AE B9DB34BC 1E29DD26 F77C8255";
            } else {
                // keystore の"my company key"の証明書ハッシュ値
                sMyCertHash = "D397D343 A5CBC10F 4EDDEB7C A10062DE 5690984F 1FB9E88B D7B3A7C2 42E142CA";
            }
        }
        return sMyCertHash;
    }

    @Override
    public boolean onCreate() {
        File dir = getContext().getFilesDir();
        FileOutputStream fos = null;
        try {
            fos = new FileOutputStream(new File(dir, FILENAME));

            // ★ポイント 1★ 利用元アプリは自社アプリであるから、センシティブな情報を保存してよい
            fos.write(new String("センシティブな情報").getBytes());

        } catch (IOException e) {
            android.util.Log.e("InHouseProvider", "ファイル保存に失敗しました");
        } finally {
            try {
                fos.close();
            } catch (IOException e) {
                android.util.Log.e("InHouseProvider", "ファイル終了に失敗しました");
            }
        }

        return true;
    }

    @Override
    public ParcelFileDescriptor openFile(Uri uri, String mode)
            throws FileNotFoundException {

        // 独自定義 Signature Permission が自社アプリにより定義されていることを確認する
        if (!SigPerm
                .test(getContext(), MY_PERMISSION, myCertHash(getContext()))) {
            throw new SecurityException(
                    "独自定義 Signature Permission が自社アプリにより定義されていない。");
        }

        File dir = getContext().getFilesDir();
        File file = new File(dir, FILENAME);

        // サンプルのため読み取り専用を常に返す
        int modeBits = ParcelFileDescriptor.MODE_READ_ONLY;
```

(continues on next page)

(continued from previous page)

```
        return ParcelFileDescriptor.open(file, modeBits);
    }

    @Override
    public String getType(Uri uri) {
        return "";
    }

    @Override
    public Cursor query(Uri uri, String[] projection, String selection,
            String[] selectionArgs, String sortOrder) {
        return null;
    }

    @Override
    public Uri insert(Uri uri, ContentValues values) {
        return null;
    }

    @Override
    public int update(Uri uri, ContentValues values, String selection,
            String[] selectionArgs) {
        return 0;
    }

    @Override
    public int delete(Uri uri, String selection, String[] selectionArgs) {
        return 0;
    }
}
```

InhouseUserActivity.java

```
package org.jssec.android.file.inhouseprovideruser;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

import org.jssec.android.shared.PkgCert;
import org.jssec.android.shared.SigPerm;
import org.jssec.android.shared.Utils;

import android.app.Activity;
import android.content.Context;
import android.content.pm.PackageManager;
import android.content.pm.ProviderInfo;
import android.net.Uri;
import android.os.Bundle;
import android.os.ParcelFileDescriptor;
import android.view.View;
import android.widget.TextView;

public class InhouseUserActivity extends Activity {
```

(continues on next page)

(continued from previous page)

```
    // 利用先の Content Provider 情報
    private static final String AUTHORITY = "org.jssec.android.file.inhouseprovider";

    // 自社の Signature Permission
    private static final String MY_PERMISSION = "org.jssec.android.file.inhouseprovider.MY_PERMISSION";

    // 自社の証明書のハッシュ値
    private static String sMyCertHash = null;

    private static String myCertHash(Context context) {
        if (sMyCertHash == null) {
            if (Utils.isDebuggable(context)) {
                // debug.keystore の"androiddebugkey"の証明書ハッシュ値
                sMyCertHash = "0EFB7236 328348A9 89718BAD DF57F544 D5CCB4AE B9DB34BC 1E29DD26 F77C8255";
            } else {
                // keystore の"my company key"の証明書ハッシュ値
                sMyCertHash = "D397D343 A5CBC10F 4EDDEB7C A10062DE 5690984F 1FB9E88B D7B3A7C2 42E142CA";
            }
        }
        return sMyCertHash;
    }

    // 利用先 Content Provider のパッケージ名を取得
    private static String providerPkgname(Context context, String authority) {
        String pkgname = null;
        PackageManager pm = context.getPackageManager();
        ProviderInfo pi = pm.resolveContentProvider(authority, 0);
        if (pi != null)
            pkgname = pi.packageName;
        return pkgname;
    }

    public void onReadFileClick(View view) {

        logLine("[ReadFile]");

        // 独自定義 Signature Permission が自社アプリにより定義されていることを確認する
        if (!SigPerm.test(this, MY_PERMISSION, myCertHash(this))) {
            logLine("  独自定義 Signature Permission が自社アプリにより定義されていない。");
            return;
        }

        // 利用先 Content Provider アプリの証明書が自社の証明書であることを確認する
        String pkgname = providerPkgname(this, AUTHORITY);
        if (!PkgCert.test(this, pkgname, myCertHash(this))) {
            logLine("  利用先 Content Provider は自社アプリではない。");
            return;
        }

        // 自社限定 Content Provider アプリに開示してよい情報に限りリクエストに含めてよい
        ParcelFileDescriptor pfd = null;
        try {
            pfd = getContentResolver().openFileDescriptor(
                    Uri.parse("content://" + AUTHORITY), "r");
        } catch (FileNotFoundException e) {
```

(continues on next page)

(continued from previous page)

```
            android.util.Log.e("InHouseUserActivity", "ファイルがありません");
        }

        if (pfd != null) {
            FileInputStream fis = new FileInputStream(pfd.getFileDescriptor());

            if (fis != null) {
                try {
                    byte[] buf = new byte[(int) fis.getChannel().size()];
                    fis.read(buf);
                    // ★ポイント 2★ 自社限定 Content Provider アプリからの結果であっても、結果データの安全性を確
認する
                    // サンプルにつき割愛。「3.2 入力データの安全性を確認する」を参照。
                    logLine(new String(buf));
                } catch (IOException e) {
                    android.util.Log.e("InHouseUserActivity", "ファイルの読み込みに失敗しました");
                } finally {
                    try {
                        fis.close();
                    } catch (IOException e) {
                        android.util.Log.e("InHouseUserActivity", "ファイルの終了に失敗しました");
                    }
                }
            }
            try {
                pfd.close();
            } catch (IOException e) {
                android.util.Log.e("InHouseUserActivity", "ファイルディスクリプタの終了に失敗しました");
            }

        } else {
            logLine("  null file descriptor");
        }
    }

    private TextView mLogView;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        mLogView = (TextView) findViewById(R.id.logview);
    }

    private void logLine(String line) {
        mLogView.append(line);
        mLogView.append("\n");
    }
}
```

4.6.3.2 ディレクトリのアクセス権設定

これまでファイルに着目してセキュリティの考慮点を説明してきた。ファイルのコンテナであるディレクトリについてもセキュリティの考慮が必要である。ここではディレクトリのアクセス権設定についてセキュリティ上の考慮ポイント

を説明する。

Android には、アプリディレクトリ内にサブディレクトリを取得・作成するメソッドがいくつか用意されている。主なものを 表 4.6.3 に示す。

表 4.6.3: アプリディレクトリ配下のサブディレクトリ取得・作成メソッド

	他アプリに対するアクセス権の指定	ユーザーによる削除
Context#getFilesDir()	不可 (実行権限のみ)	「設定」→「アプリ」→ アプリケーションを選択 →「データを消去」
Context#getCacheDir()	不可 (実行権限のみ)	「設定」→「アプリ」→ アプリケーションを選択 →「キャッシュを消去」(※「データを消去」でも削除される)
Context#getDir(String name, int mode)	mode として、MODE_PRIVATE、MODE_WORLD_READABLE、MODE_WORLD_WRITABLE を指定可能	「設定」→「アプリ」→ アプリケーションを選択 →「データを消去」

ここで特に気を付けるのは Context#getDir() によるアクセス権の設定である。ファイルの作成でも説明しているように、Android のセキュリティ設計の観点からディレクトリも基本的には非公開にするべきであり、アクセス権の設定によって情報の共有を行うと思わぬ副作用があるので、情報の共有には他の手段を考えるべきである。

MODE_WORLD_READABLE

すべてのアプリに対してディレクトリの読み取り権限を与えるフラグである。すべてのアプリがディレクトリ内のファイル一覧や個々のファイルの属性情報を取得可能になる。このディレクトリ配下に秘密のファイルを配置することはできないため、通常はこのフラグを使用してはならない[*21]。

MODE_WORLD_WRITEABLE

他アプリに対してディレクトリの書き込み権限を与えるフラグである。すべてのアプリがディレクトリ内のファイルを作成、移動[*22]、リネーム、削除が可能になる。これらの操作はファイル自体のアクセス権設定（読み取り、書き込み、実行）とは無関係であり、ディレクトリの書き込み権限があるだけで可能となる操作であることに注意が必要だ。他のアプリから勝手にファイルを削除されたり置き換えられたりするため、通常はこのフラグを使用してはならない[*21]。

表 4.6.3 の「ユーザーによる削除」に関しては、「4.6.2.4. ファイルの生存期間を考慮してアプリの設計を行う （必須）」を参照のこと。

4.6.3.3 Shared Preference やデータベースファイルのアクセス権設定

Shared Preference やデータベースもファイルで構成される。アクセス権設定についてはファイルと同じことが言える。したがって Shared Preference もデータベースもファイルと同様に基本的には非公開ファイルとして作成し、内容の共有は Android のアプリ間連携の仕組みによって実現するべきである。

[*21] MODE_WORLD_READABLE および MODE_WORLD_WRITEABLE は API Level17 以降では deprecated となっており、API Level 24 以降ではセキュリティ例外が発生するため使用できなくなっている。

[*22] 内部ストレージから外部記憶装置 (SD カードなど) への移動などマウントポイントを超えた移動はできない。そのため、読み取り権限のない内部ストレージファイルが外部記憶装置に移動されて読み書き可能になるようなことはない。

Shared Preference の使用例を次に示す。MODE_PRIVATE により非公開ファイルとして Shared Preference を作成している。

Shared Preference ファイルにアクセス制限を設定する例

```
import android.content.SharedPreferences;
import android.content.SharedPreferences.Editor;

// ～省略～

        // Shared Preference を取得する（なければ作成される）
        // ポイント：基本的に MODE_PRIVATE モードを指定する
        SharedPreferences preference = getSharedPreferences(
                PREFERENCE_FILE_NAME, MODE_PRIVATE);

        // 値が文字列のプリファレンスを書き込む例
        Editor editor = preference.edit();
        editor.putString("prep_key", "prep_value");// key:"prep_key", value:"prep_value"
        editor.commit();
```

データベースについては「4.5. *SQLite* を使う」を参照すること。

4.6.3.4 Android 4.4 (API Level 19) における外部ストレージへのアクセスに関する仕様変更について

Android 4.4 (API Level 19) 以降の端末において、外部ストレージへのアクセスに関して以下のように仕様が変更された。

(1) 外部ストレージ上のアプリ固有ディレクトリに読み書きする場合は、WRITE_EXTERNAL_STORAGE/READ_EXTERNAL_STORAGE Permission が不要である (変更箇所)

(2) 外部ストレージ上のアプリ固有ディレクトリ以外の場所にあるファイルを読み込む場合は、READ_EXTERNAL_STORAGE Permission が必要である (変更箇所)

(3) プライマリ外部ストレージ上のアプリ固有ディレクトリ以外の場所にファイルを書き込む場合は、WRITE_EXTERNAL_STORAGE Permission が必要である

(4) セカンダリ以降の外部ストレージにはアプリ固有ディレクトリ以外の場所に書き込みは出来ない

この仕様では、Android OS のバージョンによって Permission の利用宣言の要・不要が変わっているため、Android 4.4 (API Level 19) をまたいで端末のサポートが必要なアプリの場合は、インストールする端末のバージョンによって不要な Permission をユーザーに要求することになり、好ましい状況とは言えない。よって、上記仕様 (1) のみに該当するアプリの場合は、<uses-permission>タグの maxSdkVersion 属性を以下のように記述して対応することをお薦めする。

```
AndroidManifest.xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="org.jssec.android.file.externalfile" >

    <!-- android.permission.WRITE_EXTERNAL_STORAGE Permission を利用宣言する -->
    <!-- Android 4.4 (API Level 19) 以降では、外部ストレージのアプリデータ領域を
    読み書きする際に Permission が不要なため、maxSdkVersion を宣言する -->
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"
        android:maxSdkVersion="18"/>

    <application
```

(continues on next page)

(continued from previous page)

```
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:allowBackup="false" >
        <activity
            android:name=".ExternalFileActivity"
            android:label="@string/app_name"
            android:exported="true" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

4.6.3.5 Android 7.0（API Level 24）における外部ストレージの特定ディレクトリへのアクセスに関する仕様変更について

Android 7.0（API Level 24）以降の端末において、外部ストレージの特定ディレクトリに対し、Permission の利用宣言なしにアクセスできるための仕組みとして、「Scoped Directory Access」が導入された。

Scoped Directory Access では、StorageVolume#createAccessIntent メソッドの引数に Enviroment クラスで定義されたディレクトリを指定し、Intent を生成する。生成された Intent を startActivityForResult で送信することで、画面上にアクセスの許可を求めるダイアログが表示され、ユーザーが許可をするとストレージボリュームごとの指定されたディレクトリへアクセスが可能となる。

表 4.6.4　Scoped Directory Access によってアクセスできるディレクトリ

DIRECTORY_MUSIC	一般的な音楽ファイルの標準ディレクトリ
DIRECTORY_PODCASTS	ポッドキャストの標準ディレクトリ
DIRECTORY_RINGTONES	着信音の標準ディレクトリ
DIRECTORY_ALARMS	アラーム音の標準ディレクトリ
DIRECTORY_NOTIFICATIONS	通知音の標準ディレクトリ
DIRECTORY_PICTURES	画像ファイルの標準ディレクトリ
DIRECTORY_MOVIES	動画ファイルの標準ディレクトリ
DIRECTORY_DOWNLOADS	ユーザーがダウンロードしたファイルの標準ディレクトリ
DIRECTORY_DCIM	カメラによる画像・動画ファイルの標準ディレクトリ
DIRECTORY_DOCUMENTS	ユーザーによって作られたドキュメントの標準ディレクトリ

アプリがアクセスする必要のある領域が上記のディレクトリであるならば、Android 7.0 以上の端末で動作させる場合は、以下の理由により Scoped Directory Access 機能を利用することを推奨する。Android7.0 をまたいで端末のサポートが必要なアプリの場合は、「4.6.3.4. *Android 4.4 (API Level 19)* における外部ストレージへのアクセスに関する仕様変更について」で掲載した AndroidManifest の記述例を参照すること。

- 外部ストレージにアクセスできる Permission を付与した場合、アプリの目的外のディレクトリにもアクセスできてしまう。
- Storage Access Framework でアクセスできるディレクトリをユーザーに選択させる場合、その都度ピッカー上で

煩雑な操作を求められる。また、外部ストレージのルートディレクトリにアクセス許可を与えた場合は、外部ストレージ全体にアクセスできる。

4.7 Browsable Intent を利用する

ブラウザから Web ページのリンクに対応して起動するようにアプリを作ることができる。Browsable Intent という機能である。アプリは、URI スキームを Manifest ファイルで指定することで、その URI スキームを持つリンクへの移動 (ユーザーのタップなど) に反応し、リンクをパラメータとして起動することが可能になる。

また、URI スキームを利用することでブラウザから対応するアプリを起動する方法は、Android のみならず iOS 他のプラットフォームでも対応しており、Web アプリとの外部アプリ連携などに一般的に使われている。例えば、Twitter アプリや Facebook アプリでは次のような URI スキームが定義されており、Android でも iOS でもブラウザから対応するアプリが起動するようになっている。

表 4.7.1 URI スキームと対応するアプリ

URI スキーム	対応するアプリ
fb://	Facebook
twitter://	Twitter

このように連携や利便性を考えた便利な機能であるが、悪意ある第三者に悪用される危険性も潜んでいる。悪意のある Web サイトを用意してリンクの URL に不正なパラメータを仕込むことでアプリの機能を悪用したり、同じ URI スキームに対応したマルウェアをインストールさせて URL に含まれる情報を横取りしたりするなどが考えられる。

このような危険性に対応するために、利用する際にはいくつかのポイントに気をつけなければならない。

4.7.1 サンプルコード

以下に、Browsable Intent を利用したアプリのサンプルコードを示す。

ポイント：

1. (Web ページ側) 対応する URI スキーマを使ったリンクのパラメータにセンシティブな情報を含めない
2. URL のパラメータを利用する前に値の安全性を確認する

```
Starter.html
<html>
    <body>
<!-- ★ポイント 1★ URL にセンシティブな情報を含めない -->
<!-- URL パラメータとして渡す文字列は、UTF-8 で、かつ URI エンコードしておくこと -->
        <a href="secure://jssec?user=user_id"> Login </a>
    </body>
</html>
```

```
AndroidManifest.xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="org.jssec.android.browsableintent" >
```

(continues on next page)

(continued from previous page)

```
    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:allowBackup="false" >
        <activity
            android:name=".BrowsableIntentActivity"
            android:label="@string/title_activity_browsable_intent"
            android:exported="true" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>

            <intent-filter>
                <action android:name="android.intent.action.VIEW" />
                // 暗黙的 intent を受け付ける
                <category android:name="android.intent.category.DEFAULT" />
                // Browsable intent を受け付ける
                <category android:name="android.intent.category.BROWSABLE" />
                // URI 'secure://jssec' を受け付ける
                <data android:scheme="secure" android:host="jssec"/>
            </intent-filter>
        </activity>
    </application>

</manifest>
```

BrowsableIntentActivity.java

```
package org.jssec.android.browsableintent;

import android.app.Activity;
import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;
import android.widget.TextView;

public class BrowsableIntentActivity extends Activity {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_browsable_intent);

        Intent intent = getIntent();
        Uri uri = intent.getData();
        if (uri != null) {
            // URL パラメータで渡されたユーザー ID を取得する
            // ★ポイント 2★ URL のパラメータを利用する前に値の安全性を確認する
            // サンプルにつき割愛。「3.2 入力データの安全性を確認する」を参照。
            String userID = "User ID = " +  uri.getQueryParameter("user");
            TextView tv = (TextView)findViewById(R.id.text_userid);
            tv.setText(userID);
        }
    }
```

(continues on next page)

(continued from previous page)

```
}
```

4.7.2 ルールブック

Browsable Intent を利用する場合には以下のルールを守ること。

1. （*Web* ページ側）対応するリンクのパラメータにセンシティブな情報を含めない （必須）
2. *URL* のパラメータを利用する前に値の安全性を確認する （必須）

4.7.2.1 （Web ページ側）対応するリンクのパラメータにセンシティブな情報を含めない （必須）

ブラウザ上でリンクをタップした際、data（Intent#getData にて取得) に URL の値が入った Intent が発行され、システムにより該当する Intent Filter を持つアプリが起動する。

この時、同じ URI スキームを受け付けるよう Intent Filter が設定されたアプリが複数存在する場合は、通常の暗黙的 Intent による起動と同様にアプリ選択のダイアログが表示され、ユーザーの選択したアプリが起動することになる。仮に、アプリ選択画面の選択肢としてマルウェアが存在していた場合は、ユーザーが誤ってマルウェアを起動させてしまう危険性があり、パラメータがマルウェアに渡ることになる。

このように Web ページのリンク URL に含めたパラメータはすべてマルウェアに渡る可能性があるので、一般の Web ページのリンクを作るときと同様に、URL のパラメータに直接センシティブな情報を含めることは避けなければならない。

URL にユーザー ID とパスワードが入っている例

```
insecure://sample/login?userID=12345&password=abcdef
```

また、URL のパラメータがユーザー ID などセンシティブでない情報のみの場合でも、アプリ起動時のパスワード入力をアプリ側でさせるような仕様では、ユーザーが気付かずにマルウェアを起動してしまい、マルウェアに対してパスワードを入力してしまう危険性もある。そのため、一連のログイン処理自体はアプリ側で完結するような仕様を検討すべきである。Browsable Intent によるアプリ起動はあくまで暗黙的 Intent によるアプリ起動であり、意図したアプリが起動される保証がないことを念頭に置いたアプリ・サービス設計を心がける必要がある。

4.7.2.2 URL のパラメータを利用する前に値の安全性を確認する （必須）

URI スキーマに合わせたリンクは、アプリ開発者に限らず誰でも作成可能なので、アプリに渡された URL のパラメータが正規の Web ページから送られてくるとは限らない。また、渡された URL のパラメータが正規の Web ページから送られてきたかどうかを調べる方法もない。

そのため、渡された URL のパラメータを利用する前に、パラメータに想定しない値が入っていないかなど、値の安全性を確認する必要がある。

4.8 LogCat にログ出力する

Android は LogCat と呼ばれるシステムログ機構があり、システムのログ情報だけでなくアプリのログ情報も LogCat に出力される。LogCat のログ情報は同じ端末内の他のアプリからも読み取り可能[*23] であるため、センシティブな情報を LogCat にログ出力してしまうアプリには情報漏洩の脆弱性があるとされる。LogCat にはセンシティブな情報をログ出力すべきではない。

セキュリティ観点ではリリース版アプリでは一切ログ出力しないことが望ましい。しかし様々な理由によりリリース版アプリでもログ出力するケースがある。ここではリリース版アプリにおいてもログ出力しつつ、センシティブな情報はログ出力しない方法を紹介する。また「4.8.3.1. リリース版アプリにおけるログ出力の 2 つの考え方」も参照すること。

4.8.1 サンプルコード

ここでは ProGuard を利用してリリース版アプリでの LogCat へのログ出力を制御する方法を紹介する。ProGuard は使用されていないメソッド等、実質的に不要なコードを自動削除する最適化ツールの一つである。

Android の android.util.Log クラスには 5 種類のログ出力メソッド Log.e()、Log.w()、Log.i()、Log.d()、Log.v() がある。ログ情報は、リリース版アプリで出力することを意図したログ情報（以下、運用ログ情報と呼ぶ）と、リリース版アプリで出力してはならない（たとえばデバッグ用の）ログ情報（以下、開発ログ情報と呼ぶ）を区別するべきである。運用ログ出力のためには Log.e()/w()/i() を使用し、開発ログ出力のためには Log.d()/v() を使用するとよい。5 種類のログ出力メソッドの使い分けの詳細については「4.8.3.2. ログレベルとログ出力メソッドの選択基準」を参照すること。また、「4.8.3.3. *DEBUG* ログと *VERBOSE* ログは自動的に削除されるわけではない」も参照すること。

次ページ以降で、Log.d()/v() で出力する開発ログ情報を開発版アプリではログ出力し、リリース版アプリではログ出力しないサンプルコードを紹介する。このサンプルコードでは Log.d()/v() 呼び出しコードを自動削除するために、ProGuard を使用している。

ポイント：

1. センシティブな情報は Log.e()/w()/i()、System.out/err で出力しない
2. センシティブな情報をログ出力する場合は Log.d()/v() で出力する
3. Log.d()/v() の呼び出しでは戻り値を使用しない (代入や比較)
4. リリースビルドでは Log.d()/v() の呼び出しが自動削除される仕組みを導入する
5. リリース版アプリの APK ファイルはリリースビルドで作成する

ProGuardActivity.java

```
package org.jssec.android.log.proguard;

import android.app.Activity;
import android.os.Bundle;
import android.util.Log;

public class ProGuardActivity extends Activity {

    final static String LOG_TAG = "ProGuardActivity";
```

(continues on next page)

[*23] LogCat に出力されたログ情報は、READ_LOGS Permission を利用宣言したアプリであれば読み取り可能である。ただし Android 4.1 以降では LogCat に出力された他のアプリのログ情報は読み取り不可となった。また、スマートフォンユーザーであれば、ADB 経由で LogCat のログ情報を参照することも可能である。

(continued from previous page)

```
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_proguard);

        // ★ポイント 1★ センシティブな情報は Log.e()/w()/i()、System.out/err で出力しない
        Log.e(LOG_TAG, "センシティブではない情報 (ERROR)");
        Log.w(LOG_TAG, "センシティブではない情報 (WARN)");
        Log.i(LOG_TAG, "センシティブではない情報 (INFO)");

        // ★ポイント 2★ センシティブな情報をログ出力する場合は Log.d()/v() で出力する
        // ★ポイント 3★ Log.d()/v() の呼び出しでは戻り値を使用しない (代入や比較)
        Log.d(LOG_TAG, "センシティブな情報 (DEBUG)");
        Log.v(LOG_TAG, "センシティブな情報 (VERBOSE)");
    }
}
```

```
proguard-project.txt
# クラス名、メソッド名等の変更を防ぐ
-dontobfuscate

# ★ポイント 4★ リリースビルドでは Log.d()/v() の呼び出しが自動削除される仕組みを導入する
-assumenosideeffects class android.util.Log {
    public static int d(...);
    public static int v(...);
}
```

★ポイント 5 ★ リリース版アプリの APK ファイルはリリースビルドで作成する

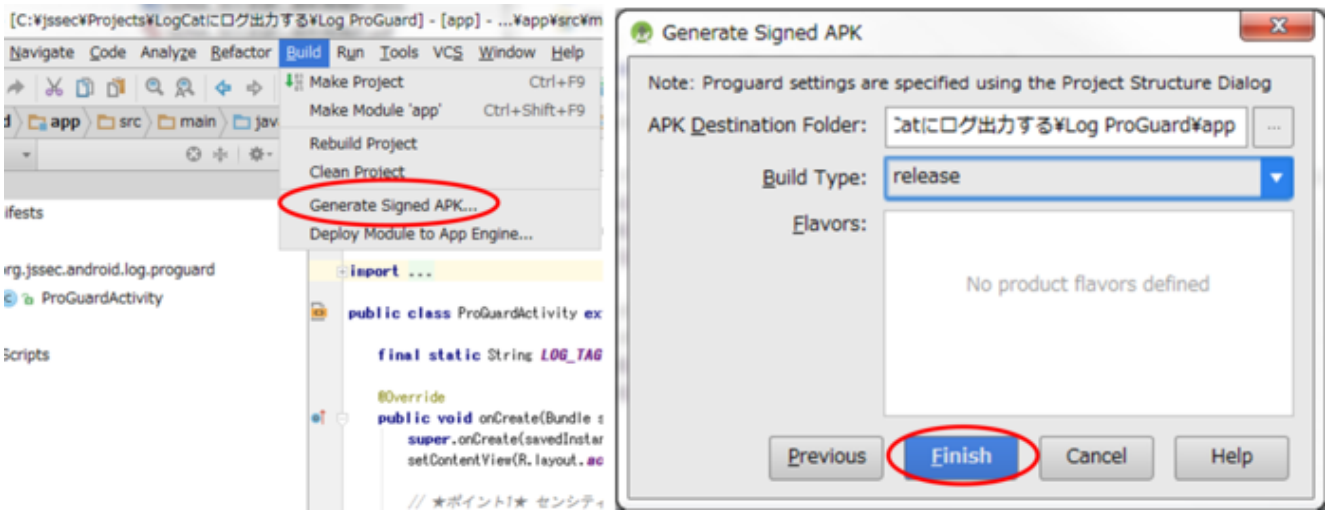

図 4.8.1　リリース版アプリを作成する方法（Export する）

開発版アプリ（デバッグビルド）とリリース版アプリ（リリースビルド）の LogCat 出力の違いを 図 4.8.2 に示す。

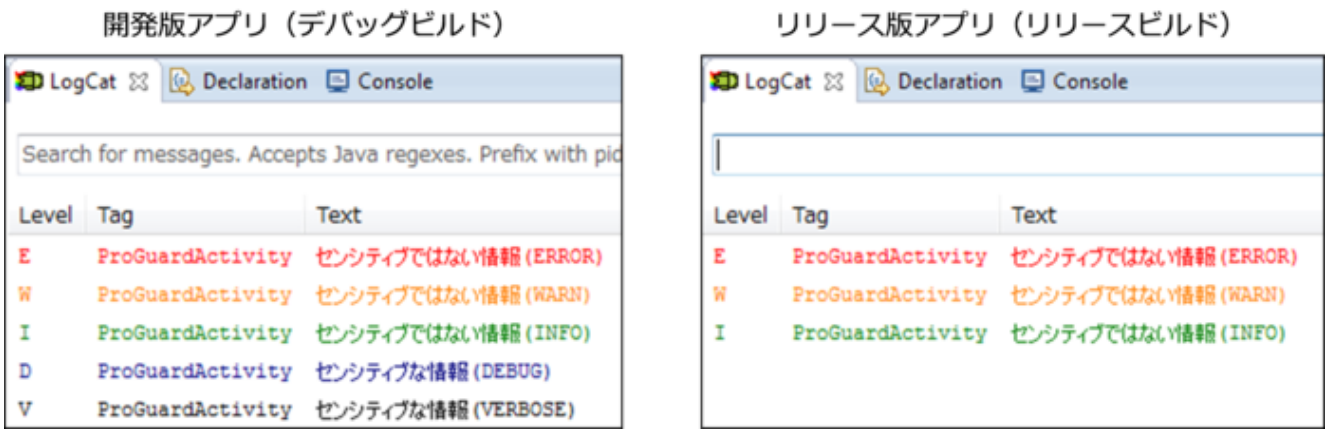

図 4.8.2　Log メソッドの開発版アプリとリリース版アプリの LogCat 出力の違い

4.8.2 ルールブック

LogCat にログを出力する際は、以下のルールを守ること。

1. 運用ログ情報にセンシティブな情報を含めない （必須）
2. 開発ログ情報を出力するコードをリリースビルド時に自動削除する仕組みを導入する （推奨）
3. *Throwable* オブジェクトをログ出力するときは *Log.d()/v()* メソッドを使う （推奨）
4. ログ出力には *android.util.Log* クラスのメソッドのみ使用する （推奨）

4.8.2.1 運用ログ情報にセンシティブな情報を含めない （必須）

LogCat に出力したログは他のアプリから読むことができるので、リリース版アプリがユーザーのログイン情報などのセンシティブな情報をログ出力することがあってはならない。開発中にセンシティブな情報をログ出力するコードを書かないようにするか、あるいは、リリース前にそのようなコードを全て削除することが必要である。

このルールを順守するためには、運用ログ情報にセンシティブな情報を含めないこと。さらに、センシティブな情報を出力するコードをリリースビルド時に削除する仕組みを導入することを強く推奨する。「4.8.2.2. 開発ログ情報を出力するコードをリリースビルド時に自動削除する仕組みを導入する （推奨）」を参照すること。

4.8.2.2 開発ログ情報を出力するコードをリリースビルド時に自動削除する仕組みを導入する （推奨）

アプリ開発中は、複雑なロジックの処理過程の中間的な演算結果、プログラム内部の状態情報、通信プロトコルの通信データ構造など、処理内容の確認やデバッグ用でセンシティブな情報をログ出力させたいことがある。アプリ開発時にセンシティブな情報をデバッグログとして出力するのは構わないが、この場合は、「4.8.2.1. 運用ログ情報にセンシティブな情報を含めない （必須）」で述べたように、リリース前に必ず該当するログ出力コードを削除すること。

リリースビルド時に開発ログ情報を出力するコードを確実に削除するために、何らかのツールを用いてコード削除を自動化する仕組みを導入すべきである。そのためのツールに「4.8.1. サンプルコード」で紹介した ProGuard がある。以下では、ProGuard を使ったコード削除の仕組みを導入する際の注意を説明する。ここでは、「4.8.3.2. ログレベルとログ出力メソッドの選択基準」に準拠し、開発ログ情報を Log.d()/v() のいずれかのみで出力しているアプリに対して仕組みを適用することを想定している。

ProGuard は使用されていないメソッド等、実質的に不要なコードを自動削除する。Log.d()/v() を-assumenosideeffects オ

プションの引数に指定することにより、Log.d()、Log.v() の呼び出しが実質的に不要なコードとみなされ、自動削除される。

Log.d()/v() を-assumenosideeffects と指定することで、自動削除の対象にする

```
-assumenosideeffects class android.util.Log {
    public static int d(...);
    public static int v(...);
}
```

この自動削除の仕組みを利用する場合は、Log.v(), Log.d() の戻り値を使用してしまうと Log.v()/d() のコードが削除されない点に注意が必要である。よって、Log.v(), Log.d() の戻り値を使用してはならない。たとえば、次の実験コードにおいては、Log.v() が削除されない。

削除指定した Log.v() が削除されない実験コード

```
int i = android.util.Log.v("tag", "message");
System.out.println(String.format("Log.v() が %d を返した。", i));  // 実験のため Log.v() の戻り値を使用。
```

また、上記 ProGuard 設定により、Log.d() 及び Log.v() が自動削除されることを前提としたソースコードがあったとする。もしそのソースコードを ProGuard 設定がされていない他のプロジェクトで再利用してしまうと、Log.d() 及び Log.v() が削除されないため、センシティブな情報が漏洩してしまう危険性がある。ソースコードを再利用する際は、ProGuard 設定を含めたプロジェクト環境の整合性を確保すること。

4.8.2.3 Throwable オブジェクトをログ出力するときは Log.d()/v() メソッドを使う （推奨）

「4.8.1. サンプルコード」および「4.8.3.2. ログレベルとログ出力メソッドの選択基準」に示した通り、Log.e()/w()/i() ではセンシティブな情報をログ出力してはならない。一方で、開発者がプログラムの異常を詳細にログ出力するために、例外発生時に Log.e(..., Throwable tr)/w(..., Throwable tr)/i(..., Throwable tr) でスタックトレースを LogCat にログ出力しているケースがみられる。しかしながら、スタックトレースはプログラムの内部構造を詳細に出力してしまうので、アプリケーションによってはセンシティブな情報が含まれてしまう場合がある。例えば、SQLiteException をそのまま出力してしまうと、どのような SQL ステートメントが発行されたかが明らかになるので、SQL インジェクション攻撃の手がかりを与えてしまうことがある。よって、Throwable オブジェクトをログ出力する際には、Log.d()/Log.v() メソッドのみを使用することを推奨する。

4.8.2.4 ログ出力には android.util.Log クラスのメソッドのみ使用する （推奨）

開発中にアプリが想定通りに動作していることを確認するために、System.out/err でログを出力することがあるだろう。もちろん System.out/err の print()/println() メソッドでも LogCat にログを出力することは可能だが、以下の理由からログ出力には android.util.Log クラスのメソッドのみを使用することを強く推奨する。

ログを出力するときは、一般には情報の緊急度に応じて出力メソッドを使い分け、出力を制御する。たとえば、深刻なエラー、警告、単なるアプリ情報通知などの区分が使われる。この区分を System.out/err に適用する手段の一つには、エラーと警告は System.err、それ以外は System.out で出力する方法がある。しかし、この場合、リリース時にも出力する必要のあるセンシティブでない情報 (運用ログ情報) とセンシティブな情報が含まれている可能性のある情報 (開発ログ情報) が同じメソッドによって出力されてしまう。よって、センシティブな情報を出力するコードを削除する際に、削除漏れが発生するおそれがある。

また、ログ出力に android.util.Log と System.out/err を使う場合は、android.util.Log のみを使う場合と比べて、ログ出力コードを削除する際に考慮することが増えるため、削除漏れなどのミスが生じるおそれがある。

上記のようなミスが生じる危険を減らすために、android.util.Log クラスのメソッドのみ使用することを推奨する。

4.8.3 アドバンスト

4.8.3.1 リリース版アプリにおけるログ出力の 2 つの考え方

リリース版 Android アプリにおけるログ出力の考え方には大きく分けて、一切ログ出力すべきではないという考え方と、後の解析のために必要な情報をログ出力すべきという考え方の 2 つがある。セキュリティ観点ではリリース版アプリでは一切ログ出力しないことが望ましい。しかし様々な理由によりリリース版アプリでもログ出力するケースがある。ここでは両者のそれぞれの考え方について述べる。

1 つ目は、リリース版アプリにおいてログ出力することにはあまり価値がなく、しかもセンシティブな情報を漏洩してしまうリスクがあるので、「一切ログ出力すべきではない」という考え方である。この考え方は、多くの Web アプリ運用環境などと違い、Android アプリ運用環境ではリリース後のアプリのログ情報を開発者が収集する手段が用意されていないことによるものである。この考え方に基づくと、開発中に使用したログ出力コードを最終版のソースコードから削除してリリース版アプリを作成するという運用がなされる。

2 つ目は、カスタマーサポート等でアプリの不具合解析を行う最終手段として、「後の解析のために必要な情報をログ出力すべき」という考え方である。この考え方に基づくと、リリース版アプリではセンシティブな情報を誤ってログ出力してしまわないよう細心の注意が必要となるため、サンプルコードセクションで紹介したような人為的ミスを排除する運用が必要となる。なお、下記の Google の Code Style Guideline も 2 つ目の考え方に基づいている。

Code Style Guidelines for Contributors / Log Sparingly

https://source.android.com/setup/contribute/code-style#log-sparingly

4.8.3.2 ログレベルとログ出力メソッドの選択基準

Android の android.util.Log クラスには ERROR, WARN, INFO, DEBUG, VERBOSE の 5 段階のログレベルが定義されている。出力したいログ情報のログレベルに応じて、適切な android.util.Log クラスのログ出力メソッドを選択する必要がある。選択基準を 表 4.8.1 にまとめた。

表 4.8.1　ログレベルとログ出力メソッドの選択基準

ログレベル	メソッド	出力するログ情報の趣旨	アプリリリース時の注意
ERROR	Log.e()	アプリが致命的な状況に陥ったときに出力するログ情報。	左記のログ情報はユーザーも参照することが想定される情報であるため、開発版アプリとリリース版アプリの両方でログ出力されるべき情報である。そのためこのログレベルではセンシティブな情報をログ出力してはならない。
WARN	Log.w()	アプリが深刻な予期せぬ状況に遭遇したときに出力するログ情報。	
INFO	Log.i()	上記以外で、アプリの注目すべき状態の変化や結果を知らせる目的で出力するログ情報。	
DEBUG	Log.d()	アプリ開発時に特定のバグの原因究明のために一時的にログ出力したいプログラム内部の状態情報。	左記のログ情報はアプリ開発者専用の情報であるため、リリース版アプリではログ出力されてはならない情報である。開発版アプリではセンシティブな情報を出力しても構わないが、リリース版アプリでは絶対にセンシティブな情報をログ出力してはならない。
VERBOSE	Log.v()	以上のいずれにも該当しないログ情報。アプリ開発者がさまざまな目的で出力するログ情報が該当する。サーバーとの通信データをダンプ出力したい場合など。	

より詳細なログ出力の作法については下記 URL を参照すること。

Code Style Guidelines for Contributors / Log Sparingly

https://source.android.com/setup/contribute/code-style#log-sparingly

4.8.3.3 DEBUG ログと VERBOSE ログは自動的に削除されるわけではない

Developer Reference の android.util.Log クラスの解説[*24] には次のような記載がある。

The order in terms of verbosity, from least to most is ERROR, WARN, INFO, DEBUG, VERBOSE. Verbose should never be compiled into an application except during development. Debug logs are compiled in but stripped at runtime. Error, warning and info logs are always kept.

開発者の中には、この文章から Log クラスの動作を次のように誤った解釈をしている人がいる。

- Log.v() 呼び出しはリリースビルド時にはコンパイルされず、VERBOSE ログが出力されることがなくなる
- Log.d() 呼び出しはコンパイルされるが、実行時には DEBUG ログが出力されることはない

しかし実際には Log クラスはこのようには動作せず、デバッグビルド、リリースビルドを問わず全てのログを出力してしまう。よく読んでみるとわかるが、この英文は Log クラスの動作について語っているのではなく、ログ情報とはこうあるべきということを説明しているだけである。

この記事のサンプルコードでは、ProGuard を使って上記英文のような動作を実現する方法を紹介している。

4.8.3.4 ログ情報の組み立て処理を削除する

下記ソースコードを ProGuard でリリースビルドして Log.d() を削除した場合、Log.d() の呼び出し処理（下記コードの 2 行目）は削除されるものの、その前段でセンシティブな情報を組み立てる処理（下記コードの 1 行目）は削除されないこ

[*24] https://developer.android.com/intl/ja/reference/android/util/Log.html

とに注意が必要である。

```
    String debug_info = String.format("%s:%s", "センシティブな情報 1", "センシティブな情報 2");
    if (BuildConfig.DEBUG) android.util.Log.d(TAG, debug_info);
```

上記ソースコードをリリースビルドした APK ファイルを逆アセンブルすると次のようになる。確かに Log.d() の呼び出し処理は存在しないが、” センシティブな情報 1” といった文字列定数定義と String#format() メソッドの呼び出し処理が削除されず残っていることが分かる。

```
    const-string v1, "%s:%s"
    const/4 v2, 0x2
    new-array v2, v2, [Ljava/lang/Object;
    const/4 v3, 0x0
    const-string v4, "センシティブな情報 1"
    aput-object v4, v2, v3
    const/4 v3, 0x1
    const-string v4, "センシティブな情報 2"
    aput-object v4, v2, v3
    invoke-static {v1, v2}, Ljava/lang/String;->format(Ljava/lang/String;[Ljava/lang/Object;)Ljava/lang/
↪String;
    move-result-object v0
```

実際には APK ファイルを逆アセンブルして、上記のようにログ出力情報を組み立てている箇所を発見するのは容易なことではない。しかし非常に機密度の高い情報を扱っているアプリにおいては、このような処理が APK ファイルに残ってしまってはならない場合もあり得る。

もし上記のようなログ出力情報の組み立て処理も削除してしまいたい場合には、次のように記述するとよい[*25]。リリースビルド時にはコンパイラの最適化処理によって、下記サンプルコードの処理は丸ごと削除される。

```
    if (BuildConfig.DEBUG) {
        String debug_info = String.format("%s:%s", "センシティブな情報 1", "センシティブな情報 2");
        if (BuildConfig.DEBUG) android.util.Log.d(TAG, debug_info);
    }
```

なお、下記ソースコードに ProGuard を適用した場合も、同様にログ情報の組み立て処理（”result:”+ value の部分）が残ってしまう。

```
    Log.d(TAG, "result:" + value);
```

この場合も下記のように対処すればよい。

```
    if (BuildConfig.DEBUG) Log.d(TAG, "result:" + value);
```

4.8.3.5 Intent の内容が LogCat に出力される

Activity を利用する際に ActivityManager が Intent の内容を LogCat に出力するため、注意が必要である。「4.1.3.5. *Activity* 利用時のログ出力について」を参照すること。

[*25] 前述のサンプルコードを、条件式に BuildConfig.DEBUG を用いた if 文で囲った。Log.d() 呼び出し前の if 文は不要であるが、前述のサンプルコードと対比させるため、そのまま残した。

4.8.3.6 System.out/err に出力されるログの抑制

System.out/err の出力先は LogCat である。System.out/err に出力されるのは、開発者がデバッグのために出力したログに限らない。例えば、次の場合、スタックトレースは System.err に出力される。

- Exception#printStackTrace() を使った場合
- 暗黙的に System.err に出力される場合（例外をアプリでキャッチしていない場合、システムが Exception#printStackTrace() に渡すため。）

スタックトレースにはアプリ固有の情報が含まれるため、例外は開発者が正しくハンドリングすべきである。

保険的対策として、System.out/err の出力先を LogCat 以外に変更する方法がある。以下に、リリースビルド時に System.out/err の出力先を変更し、どこにもログ出力しないようにする実装例を挙げる。ただし、この対応は System.out/err の出力先をアプリの実行時に一時的に書き換えるので、アプリやシステムの誤動作に繋がらないかどうかを充分に検討する必要がある。また、この対策はアプリ自身のプロセスには有効であるが、システムプロセスが生成するエラーログを抑制することはできない。すべてのエラーを抑制できるわけではないことに注意すること。

OutputRedirectApplication.java

```
package org.jssec.android.log.outputredirection;

import java.io.IOException;
import java.io.OutputStream;
import java.io.PrintStream;

import android.app.Application;

public class OutputRedirectApplication extends Application {

    // どこにも出力しない PrintStream
    private final PrintStream emptyStream = new PrintStream(new OutputStream() {
        public void write(int oneByte) throws IOException {
            // do nothing
        }
    });

    @Override
    public void onCreate() {
        // リリースビルド時に System.out/err をどこにも出力しない PrintStream にリダイレクトする

        // System.out/err の本来のストリームを退避する
        PrintStream savedOut = System.out;
        PrintStream savedErr = System.err;

        // 一旦、System.out/err をどこにも出力しない PrintStream にリダイレクトする
        System.setOut(emptyStream);
        System.setErr(emptyStream);

        // デバッグ時のみ本来のストリームに戻す（リリースビルドでは下記 1 行が ProGuard により削除される）
        resetStreams(savedOut, savedErr);
    }

    // リリース時は ProGuard により下記メソッドがまるごと削除される
    private void resetStreams(PrintStream savedOut, PrintStream savedErr) {
        System.setOut(savedOut);
        System.setErr(savedErr);
```

(continues on next page)

(continued from previous page)

```
    }
}
```

AndroidManifest.xml

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="org.jssec.android.log.outputredirection" >

    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:name=".OutputRedirectApplication"
        android:allowBackup="false" >
        <activity
            android:name=".LogActivity"
            android:label="@string/app_name"
            android:exported="true" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

proguard-project.txt

```
# クラス名、メソッド名等の変更を防ぐ
-dontobfuscate

# リリースビルド時に Log.d()/v() の呼び出しを自動的に削除する
-assumenosideeffects class android.util.Log {
    public static int d(...);
    public static int v(...);
}

# リリースビルド時に resetStreams() を自動的に削除する
-assumenosideeffects class org.jssec.android.log.outputredirection.OutputRedirectApplication {
    private void resetStreams(...);
}
```

開発版アプリ（デバッグビルド）とリリース版アプリ（リリースビルド）の LogCat 出力の違いを 図 4.8.3 に示す。

図 4.8.3　System.out/err の開発版アプリとリリース版アプリの LogCat 出力の違い

4.9 WebView を使う

Web サイトや HTML ファイルを閲覧する機能を実装する方法として、WebView を使用することができる。WebView は HTML をレンダリングする、JavaScript を実行するなど、この目的のために有用な機能を提供する。

4.9.1 サンプルコード

WebView を使用することにより容易に Web サイト、HTML ファイル閲覧機能を実現することができるが、アクセスするコンテンツの特性によって WebView が抱えるリスクや適切な防衛手段が異なってくる。

特に気をつけなければいけないのは JavaScript の使用である。WebView のデフォルト設定では JavaScript の機能が無効になっているが、WebSettings#setJavaScriptEnabled() メソッドにより有効にすることが可能である。JavaScript を使用することでインタラクティブなコンテンツの表示が可能になるが、悪意のある第三者により端末の情報を取得される、あるいは端末を操作されるという被害が発生する可能性がある。

WebView を用いてコンテンツにアクセスするアプリを開発する際は、次の原則に従うこと[*26]。

1. 自社が管理しているコンテンツにのみアクセスする場合に限り JavaScript を有効にしてよい
2. 上記以外の場合には、JavaScript を有効にしてはならない

開発しているアプリがアクセスするコンテンツの特性を踏まえ、図 4.9.1 に従いサンプルコードを選択することが必要である。

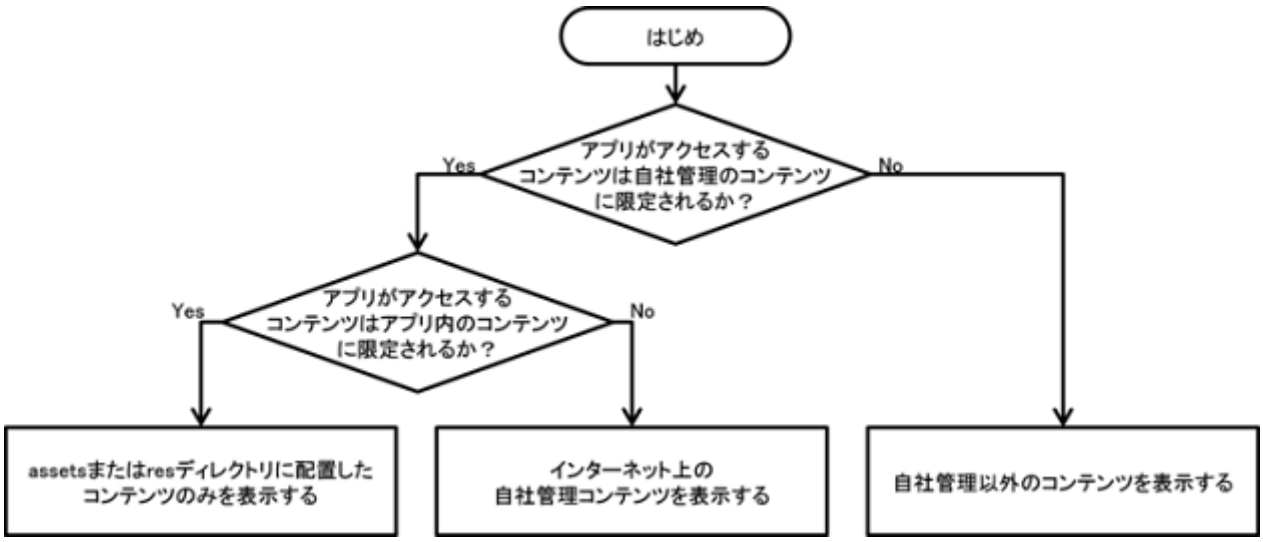

図 4.9.1　WebView のサンプルコードを選択するフローチャート

4.9.1.1 assets または res ディレクトリに配置したコンテンツのみを表示する

端末内のローカルコンテンツを WebView で表示するアプリに関しては、アプリの APK に含まれる assets あるいは res ディレクトリ内のコンテンツにアクセスする場合に限り JavaScript を有効にしてもよい。

以下に WebView を使用して assets ディレクトリ内にある HTML ファイルを表示するサンプルコードを示す。

[*26] 厳密に言えば安全性を保証できるコンテンツであれば JavaScript を有効にしてよい。自社管理のコンテンツであれば自社の努力で安全性を確保できるし責任も取れる。では信頼できる提携会社のコンテンツは安全だろうか？ これは会社間の信頼関係により決まる。信頼できる提携会社のコンテンツを安全であると信頼して JavaScript を有効にしてもよいが、万一の場合は自社責任も伴うため、ビジネス責任者の判断が必要となる。

ポイント：

1. assets と res ディレクトリ以外の場所に配置したファイルへのアクセスを禁止にする
2. JavaScript を有効にしてよい

```
WebViewAssetsActivity.java
package org.jssec.webview.assets;

import android.app.Activity;
import android.os.Bundle;
import android.webkit.WebSettings;
import android.webkit.WebView;

public class WebViewAssetsActivity extends Activity {
    /**
     * assets 内のコンテンツを表示する
     */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        WebView webView = (WebView) findViewById(R.id.webView);
        WebSettings webSettings = webView.getSettings();

        // ★ポイント 1★ assets と res ディレクトリ以外の場所に配置したファイルへのアクセスを禁止にする
        webSettings.setAllowFileAccess(false);

        // ★ポイント 2★ JavaScript を有効にしてよい
        webSettings.setJavaScriptEnabled(true);

        // assets 内に配置したコンテンツを表示する
        webView.loadUrl("file:///android_asset/sample/index.html");
    }
}
```

4.9.1.2 インターネット上の自社管理コンテンツを表示する

自社の管理するサービス上のコンテンツを表示する場合、サービス側、アプリ側の双方で適切な対策を施し、安全が確保できるならば、JavaScript を有効にしてもよい。

- サービス側の対策

図 4.9.2 に示すように、サービス側に用意するコンテンツは自社の管理していないコンテンツを参照してはならない。加えて、サービスに適切なセキュリティ対策が施されていることも必要である。その理由は、サービスを構成するコンテンツへの攻撃コードの埋め込みや改ざんを防止することにある。「4.9.2.1. *JavaScript* を有効にするのはコンテンツを自社が管理している場合に限定する （必須）」を参照すること。

- アプリ側の対策

次にアプリ側での対策を述べる。アプリ側では、接続先が自社管理サービスであることを確認することが必要である。そのために、通信プロトコルは HTTPS を使用し、証明書が信頼できる場合のみ接続するように実装する。

以下では、アプリ側での実装の例として、WebView を使って自社管理コンテンツを表示する Activity の例を示す。

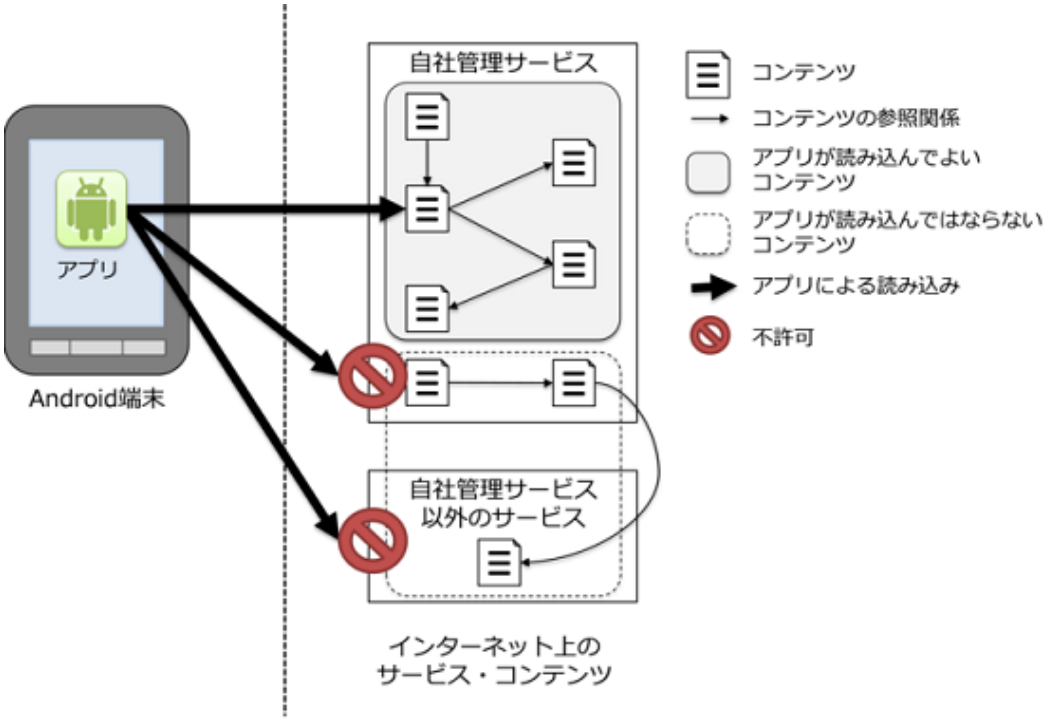

図 4.9.2　アプリが読み込んでよい自社管理コンテンツ

ポイント：

1. WebView の SSL 通信エラーを適切にハンドリングする
2. WebView の JavaScript を有効にしてもよい
3. WebView で表示する URL を HTTPS プロトコルだけに限定する
4. WebView で表示する URL を自社管理コンテンツだけに限定する

```
WebViewTrustedContentsActivity.java
package org.jssec.webview.trustedcontents;

import android.app.Activity;
import android.app.AlertDialog;
import android.content.DialogInterface;
import android.net.http.SslCertificate;
import android.net.http.SslError;
import android.os.Bundle;
import android.webkit.SslErrorHandler;
import android.webkit.WebView;
import android.webkit.WebViewClient;

import java.text.SimpleDateFormat;

public class WebViewTrustedContentsActivity extends Activity {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        WebView webView = (WebView) findViewById(R.id.webView);
```

(continues on next page)

(continued from previous page)

```
        webView.setWebViewClient(new WebViewClient() {
            @Override
            public void onReceivedSslError(WebView view,
                    SslErrorHandler handler, SslError error) {
                // ★ポイント 1★ WebView の SSL 通信エラーを適切にハンドリングする
                // SSL エラーが発生した場合には、SSL エラーが発生した旨をユーザに通知する
                AlertDialog dialog = createSslErrorDialog(error);
                dialog.show();

                // ★ポイント 1★ WebView の SSL 通信エラーを適切にハンドリングする
                // SSL エラーが発生した場合、有効期限切れなど証明書に不備があるか、
                // もしくは中間者攻撃を受けている可能性があるので、安全のために接続を中止する。
                handler.cancel();
            }
        });


        // ★ポイント 2★ WebView の JavaScript を有効にしてもよい
        // 以下のコードでは、loadUrl() で自社管理コンテンツを読みこむことを想定している。
        webView.getSettings().setJavaScriptEnabled(true);

        // ★ポイント 3★ WebView で表示する URL を HTTPS プロトコルだけに限定する
        // ★ポイント 4★ WebView で表示する URL を自社管理コンテンツだけに限定する
        webView.loadUrl("https://url.to.your.contents/");
    }

    private AlertDialog createSslErrorDialog(SslError error) {
        // ダイアログに表示するエラーメッセージ
        String errorMsg = createErrorMessage(error);
        // ダイアログの OK ボタン押下時の挙動
        DialogInterface.OnClickListener onClickOk = new DialogInterface.OnClickListener() {
            @Override
            public void onClick(DialogInterface dialog, int which) {
                setResult(RESULT_OK);
            }
        };
        // ダイアログの作成
        AlertDialog dialog = new AlertDialog.Builder(
                WebViewTrustedContentsActivity.this).setTitle("SSL 接続エラー")
                .setMessage(errorMsg).setPositiveButton("OK", onClickOk)
                .create();
        return dialog;
    }

    private String createErrorMessage(SslError error) {
        SslCertificate cert = error.getCertificate();
        SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy/MM/dd HH:mm:ss");
        StringBuilder result = new StringBuilder()
        .append("サイトのセキュリティ証明書が信頼できません。接続を終了しました。\n\n エラーの原因\n");
        switch (error.getPrimaryError()) {
        case SslError.SSL_EXPIRED:
            result.append("証明書の有効期限が切れています。\n\n 終了時刻=")
            .append(dateFormat.format(cert.getValidNotAfterDate()));
            return result.toString();
        case SslError.SSL_IDMISMATCH:
```

(continues on next page)

(continued from previous page)

```
            result.append("ホスト名が一致しません。\n\nCN=")
            .append(cert.getIssuedTo().getCName());
            return result.toString();
        case SslError.SSL_NOTYETVALID:
            result.append("証明書はまだ有効ではありません\n\n 開始時刻=")
            .append(dateFormat.format(cert.getValidNotBeforeDate()));
            return result.toString();
        case SslError.SSL_UNTRUSTED:
            result.append("証明書を発行した認証局が信頼できません\n\n 認証局\n")
            .append(cert.getIssuedBy().getDName());
            return result.toString();
        default:
            result.append("原因不明のエラーが発生しました");
            return result.toString();
        }
    }
}
```

4.9.1.3 自社管理以外のコンテンツを表示する

自社で管理していないコンテンツを WebView で接続・表示する場合は、JavaScript を有効にしてはならない。攻撃者が用意したコンテンツに接続する可能性があるからである。

以下のサンプルコードは WebView を使用して自社管理以外のコンテンツを表示するアプリである。このアプリは、アドレスバーに入力した URL の指す HTML ファイルなどのコンテンツを読み込み、画面に表示する。安全の確保のために JavaScript を無効化しているほか、HTTPS で通信していて SSL エラーが発生した場合は接続を中止する実装となっている。SSL エラーは「4.9.1.2. インターネット上の自社管理コンテンツを表示する」と同様の方法によりハンドリングしている。HTTPS 通信についての詳細は、「5.4. *HTTPS* で通信する」を参照すること。

ポイント：

1. HTTPS 通信の場合には SSL 通信のエラーを適切にハンドリングする
2. JavaScript を有効にしない

WebViewUntrustActivity.java

```
package org.jssec.webview.untrust;

import android.app.Activity;
import android.app.AlertDialog;
import android.content.DialogInterface;
import android.graphics.Bitmap;
import android.net.http.SslCertificate;
import android.net.http.SslError;
import android.os.Bundle;
import android.view.View;
import android.webkit.SslErrorHandler;
import android.webkit.WebView;
import android.webkit.WebViewClient;
import android.widget.Button;
import android.widget.EditText;
```

(continues on next page)

(continued from previous page)

```
import java.text.SimpleDateFormat;

public class WebViewUntrustActivity extends Activity {
    /*
     * 自社管理以外のコンテンツを表示する（簡易ブラウザとして機能するサンプルプログラム）
     */
    private EditText textUrl;
    private Button buttonGo;
    private WebView webView;


    // この Activity が独自に URL リクエストをハンドリングできるようにするために定義
    private class WebViewUnlimitedClient extends WebViewClient {

        @Override
        public boolean shouldOverrideUrlLoading(WebView webView, String url) {
            webView.loadUrl(url);
            textUrl.setText(url);
            return true;
        }

        // Web ページの読み込み開始処理
        @Override
        public void onPageStarted(WebView webview, String url, Bitmap favicon) {
            buttonGo.setEnabled(false);
            textUrl.setText(url);
        }

        // SSL 通信で問題があるとエラーダイアログを表示し、
        // 接続を中止する
        @Override
        public void onReceivedSslError(WebView webview,
                SslErrorHandler handler, SslError error) {
            // ★ポイント 1★ HTTPS 通信の場合には SSL 通信のエラーを適切にハンドリングする
            AlertDialog errorDialog = createSslErrorDialog(error);
            errorDialog.show();
            handler.cancel();
            textUrl.setText(webview.getUrl());
            buttonGo.setEnabled(true);
        }

        // Web ページの load が終わったら表示されたページの URL を EditText に表示させる
        @Override
        public void onPageFinished(WebView webview, String url) {
            textUrl.setText(url);
            buttonGo.setEnabled(true);
        }
    }

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        webView = (WebView) findViewById(R.id.webview);
```

(continues on next page)

(continued from previous page)

```
        webView.setWebViewClient(new WebViewUnlimitedClient());

        // ★ポイント 2★ JavaScript を有効にしない
        // デフォルトの設定で JavaScript 無効となっているが、明示的に無効化する
        webView.getSettings().setJavaScriptEnabled(false);

        webView.loadUrl(getString(R.string.texturl));
        textUrl = (EditText) findViewById(R.id.texturl);
        buttonGo = (Button) findViewById(R.id.go);
    }

    public void onClickButtonGo(View v) {
        webView.loadUrl(textUrl.getText().toString());
    }

    private AlertDialog createSslErrorDialog(SslError error) {
        // ダイアログに表示するエラーメッセージ
        String errorMsg = createErrorMessage(error);
        // ダイアログの OK ボタン押下時の挙動
        DialogInterface.OnClickListener onClickOk = new DialogInterface.OnClickListener() {
            @Override
            public void onClick(DialogInterface dialog, int which) {
                setResult(RESULT_OK);
            }
        };
        // ダイアログの作成
        AlertDialog dialog = new AlertDialog.Builder(
                WebViewUntrustActivity.this).setTitle("SSL 接続エラー")
                .setMessage(errorMsg).setPositiveButton("OK", onClickOk)
                .create();
        return dialog;
    }

    private String createErrorMessage(SslError error) {
        SslCertificate cert = error.getCertificate();
        SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy/MM/dd HH:mm:ss");
        StringBuilder result = new StringBuilder()
        .append("サイトのセキュリティ証明書が信頼できません。接続を終了しました。\n\n エラーの原因\n");
        switch (error.getPrimaryError()) {
        case SslError.SSL_EXPIRED:
            result.append("証明書の有効期限が切れています。\n\n 終了時刻=")
            .append(dateFormat.format(cert.getValidNotAfterDate()));
            return result.toString();
        case SslError.SSL_IDMISMATCH:
            result.append("ホスト名が一致しません。\n\nCN=")
            .append(cert.getIssuedTo().getCName());
            return result.toString();
        case SslError.SSL_NOTYETVALID:
            result.append("証明書はまだ有効ではありません\n\n 開始時刻=")
            .append(dateFormat.format(cert.getValidNotBeforeDate()));
            return result.toString();
        case SslError.SSL_UNTRUSTED:
            result.append("証明書を発行した認証局が信頼できません\n\n 認証局\n")
            .append(cert.getIssuedBy().getDName());
            return result.toString();
```

(continues on next page)

(continued from previous page)

```
        default:
            result.append("原因不明のエラーが発生しました");
            return result.toString();
        }
    }
}
```

4.9.2 ルールブック

WebView を使用する際には以下のルールを守ること。

1. *JavaScript* を有効にするのはコンテンツを自社が管理している場合に限定する （必須）
2. 自社管理サービスとの通信には *HTTPS* を使用する （必須）
3. *Intent* 経由など、他から受け取った *URL* は *JavaScript* が有効な *WebView* には表示しない （必須）
4. *SSL* 通信のエラーを適切にハンドリングする （必須）

4.9.2.1 JavaScript を有効にするのはコンテンツを自社が管理している場合に限定する （必須）

WebView を用いてコンテンツやサービスにアクセスするアプリを開発する際に、セキュリティの面で最も注意しなければならない点は JavaScript を有効にするかどうかである。原則的には、自社が管理しているサービスにのみアプリがアクセスする場合に限り JavaScript を有効にしてもよい。しかし、そうでないサービスにアクセスする可能性が少しでもある場合には、JavaScript を有効にしてはならない。

自社で管理しているサービス

自社で作成あるいは、運用、管理に責任を持つサービスは、自社が安全を保証できる。例として、自社管理サーバー上の自社開発コンテンツにアプリがアクセスする場合を考える。各コンテンツがサーバー内部のコンテンツのみを参照しており、かつ、自社管理サーバーに対して適切なセキュリティ対策が施されているならば、このサービスは自社以外が内容を書き換えていることはないとみなせる。この場合、自社管理サービスにアクセスするアプリの JavaScript を有効にしてもよい。「4.9.1.2. インターネット上の自社管理コンテンツを表示する」を参照すること。また、他のアプリによる書き換えが不可能な端末内コンテンツ (APK の assets または res ディレクトリ内に配置されたコンテンツやアプリディレクトリ下のコンテンツ) にアクセスするアプリの場合も同様に考え、JavaScript を有効にしてもよい。「4.9.1.1. *assets* または *res* ディレクトリに配置したコンテンツのみを表示する」を参照すること。

自社で管理していないサービス

自社で管理していないコンテンツ・サービスは自社が安全を保証できると考えてはならない。それゆえ、アプリの JavaScript を無効にしなければならない。「4.9.1.3. 自社管理以外のコンテンツを表示する」を参照すること。加えて、SD カードのような端末の外部記憶装置に配置されたコンテンツは他のアプリによる書き換えが可能なので、自社が管理しているとは言えない。そのようなコンテンツにアクセスするアプリについても JavaScript を無効にしなければならない。

4.9.2.2 自社管理サービスとの通信には **HTTPS** を使用する （必須）

自社管理サービスにアクセスするアプリは、悪意ある第三者によるサービスのなりすましによる被害を防ぎ、対象サービスへ確実に接続する必要がある。そのためには、サービスとの通信に HTTPS を使用する。

詳細は「4.9.2.4. *SSL* 通信のエラーを適切にハンドリングする （必須）」、「5.4. *HTTPS* で通信する」を参照すること。

4.9.2.3 **Intent** 経由など、他から受け取った **URL** は **JavaScript** が有効な **WebView** には表示しない （必須）

他のアプリから Intent を受信し、その Intent のパラメータで渡された URL を WebView に表示する実装が多くのアプリで見られる。ここで WebView の JavaScript が有効である場合、悪意ある Web ページの URL を WebView で表示してしまい、悪意ある JavaScript が WebView 上で実行されて何らかの被害が生じる可能性がある。この実装の問題点は、安全を保証できない不特定の URL を JavaScript が有効な WebView で表示してしまうことである。

サンプルコード「4.9.1.2. インターネット上の自社管理コンテンツを表示する」では、固定 URL 文字列定数で自社管理コンテンツを指定することで、WebView で表示するコンテンツを自社管理コンテンツに限定し安全を確保している。

もし Intent 等で受け取った URL を JavaScript が有効な WebView で表示したい場合は、その URL が自社管理コンテンツであることを保証しなければならない。あらかじめアプリ内に自社管理コンテンツ URL のホワイトリストを正規表現等で保持しておき、このホワイトリストと照合して合致した URL だけを WebView で表示することで安全を確保することができる。この場合も、ホワイトリスト登録する URL は HTTPS でなければならないことにも注意が必要だ。

4.9.2.4 **SSL** 通信のエラーを適切にハンドリングする （必須）

HTTPS 通信で SSL エラーが発生した場合は、エラーが発生した旨をダイアログ表示するなどの方法でユーザーに通知して、通信を終了しなければならない。

SSL エラーの発生は、サーバー証明書に不備がある可能性、あるいは中間者攻撃を受けている可能性を示唆する。しかし、WebView には、サービスとの通信時に発生した SSL エラーに関する情報をユーザーに通知する仕組みが備わっていない。そこで、SSL エラーが発生した場合にはその旨をダイアログなどで表示することで、脅威にさらされている可能性があることをユーザーに通知する必要がある。エラー通知の例は、「4.9.1.2. インターネット上の自社管理コンテンツを表示する」のサンプルコードあるいは「4.9.1.3. 自社管理以外のコンテンツを表示する」のサンプルコードを参照すること。

また、エラーの通知に加えて、アプリはサービスとの通信を終了しなければならない。特に、次のような実装を行ってはならない。

- 発生したエラーを無視してサービスとの通信を継続する
- HTTP などの非暗号化通信を使ってサービスと改めて通信する

HTTP 通信/HTTPS 通信の詳細は「5.4. *HTTPS* で通信する」を参照すること。

SSL エラーが発生した際には対象のサーバーと接続を行わないことが WebView のデフォルトの挙動である。よって、WebView のデフォルトの挙動に SSL エラーの通知機能を実装することで適切に通信エラーを取り扱うことができる。

4.9.3 アドバンスト

4.9.3.1 **Android 4.2** 未満の端末における **addJavascriptInterface()** に起因する脆弱性について

Android 4.2（API Level 17）未満の端末には addJavascriptInterface() に起因する脆弱性があり、JavaScript から Java のリフレクションを行うことにより任意の Java メソッドが実行できてしまう問題が存在する。

そのため、「4.9.2.1. *JavaScript* を有効にするのはコンテンツを自社が管理している場合に限定する （必須）」で解説した通り、自社で管理していないコンテンツ・サービスにアクセスする可能性がある場合は、JavaScript を無効にする必要がある。

Android 4.2（API Level 17）以降の端末では、Java のソースコード上で@JavascriptInterface というアノテーションが指定されたメソッドしか JavaScript から操作できないように API が仕様変更され、脆弱性の対策がされた。ただし、自社で管理していないコンテンツ・サービスにアクセスする可能性がある場合は、コンテンツ・サービス提供者が悪意ある JavaScript を送信する恐れがあるため、JavaScript を無効化する対策は引き続き必要である。

4.9.3.2 file スキームに起因する問題について

WebView をデフォルト設定で使用している場合、file スキームを利用してアクセスすると当該アプリがアクセス可能なすべてのファイルにアクセスすることが可能になる。この動作を悪用された場合、例えば、JavaScript から file スキーム使ったリクエストすることで、アプリの専用フォルダに保存したファイル等を攻撃者に取得されてしまう可能性がある。

対策としては、「4.9.2.1. *JavaScript* を有効にするのはコンテンツを自社が管理している場合に限定する （必須）」で解説した通り、自社で管理していないコンテンツ・サービスにアクセスする可能性がある場合は JavaScript を無効にする。この対策により意図しない file スキームによるリクエストが送信されないようにする。

また、Android 4.1（API Level 16）以降の場合、setAllowFileAccessFromFileURLs() および setAllowUniversalAccessFromFileURLs() を利用することで file スキームによるアクセスを禁止することができる。

file スキームの無効化

```
        webView = (WebView) findViewById(R.id.webview);
        webView.setWebViewClient(new WebViewUnlimitedClient());
        WebSettings settings = webView.getSettings();
        settings.setAllowUniversalAccessFromFileURLs(false);
        settings.setAllowFileAccessFromFileURLs(false);
```

4.9.3.3 Web Messaging 利用時の送信先オリジン指定について

Android 6.0(API Level 23) において、HTML5 Web Messaging を実現するための API が追加された。Web Messaging は異なるブラウジング・コンテキスト間でデータを送受信するための仕組みであり、HTML5 で定義されている[*27]。

WebView クラスに追加された postWebMessage() は Web Messaging で定義されている Cross-domain messaging によるデータ送信を処理するメソッドである。このメソッドは第１引数で指定されたメッセージオブジェクトを WebView に読み込んでいるブラウジング・コンテキストに対して送信するのだが、その際第２引数として送信先のオリジンを指定する必要がある。指定されたオリジン[*28] が送信先コンテキストのオリジンと一致しない限りメッセージは送信されない。送信先オリジンを制限することで、意図しない送信先にメッセージを渡してしまうことを防いでいるのである。

ただし、postWebMessage() メソッドではオリジンとしてワイルドカードを指定できることに注意が必要である[*29]。ワイルドカードを指定するとメッセージの送信先オリジンがチェックされず、どのようなオリジンに対してもメッセージを送信してしまう。もし WebView に悪意のあるコンテンツが読み込まれている状況でオリジンの制限なしに重要なメッセージを送信してしまうと何らかの被害につながる可能性も生じる。WebView を用いて Web messaging を行う際は、postWebMessage() メソッドに特定のオリジンを明示的に指定するべきである。

[*27] https://www.w3.org/TR/webmessaging/
[*28] オリジンとは、URL のスキーム、ホスト名、ポート番号の組み合わせのこと。詳細な定義は https://tools.ietf.org/html/rfc6454 を参照。
[*29] Uri.EMPTY および Uri.parse("") がワイルドカードとして機能する (2016 年 9 月 1 日版執筆時)

4.9.3.4 WebView の Safe Browsing について

Safe Browsing とは Google が提供するサービスで、マルウェアページやフィッシングサイトなどの安全でない Web ページにユーザーがアクセスしようとした際に警告ページを表示させる仕組みである。

図 4.9.3　Chrome for Android で安全でない Web ページにアクセスした際に表示される警告ページ

現在、Chrome for Android などのブラウザアプリに限らず、アプリで使われる WebView においても Safe Browsing の機能を利用することが可能となっている。しかし、端末の Android OS バージョンにより WebView として用いられるコンポーネントが異なり、Safe Browsing の対応状況が異なるため注意が必要である。端末の Android OS バージョンと標準 WebView の対応と Safe Browsing の対応を次の表にまとめた。

表 4.9.1: 端末の Android OS バージョンと標準 WebView の対応

Android OS バージョン	**Android** 標準 **WebView**	**OS** との関係	**Safe Browsing** 対応
Android 7.0 以降	Chrome for Android (Chromium ベース)	独立	○
Android 5.0 - 6.0	Android System WebView (Chromium ベース)	独立	○
Android 4.4	OS 組み込み WebView (Chromium ベース)	一体	×
Android 4.3 以前	OS 組み込み WebView	一体	×

Android 4.3 (API Level 18) 以前は、Safe Browsing 機能が搭載されていない WebView が OS に組み込まれていたが、Android 4.4 (API Level 19) にて Safe Browsing 機能を持っているものに変更された。しかし、バージョンが古く、アプリ

の WebView での Safe Browsing 機能の利用には対応していないため注意すること。

Safe Browsing 機能をアプリの WebView で利用できるのは、WebView が OS から切り離されアプリとして更新されるようになった Android 5.0 (API Level 21) からである。

WebView 66 以降では Safe Browsing がデフォルトで有効になっており、アプリ側で特別何か設定を行う必要はない。しかし、ユーザーが WebView の更新を行っていなかった場合や、開発者向けオプション「WebView の実装」項目で標準 WebView をデフォルトのものから変更している場合、その WebView のバージョンによっては Safe Browsing がデフォルトで有効とならない可能性が考えられる。そのため、Safe Browsing を利用するのならば下記のように Safe Browsing を明示的に有効化しておくべきである。

AndroidManifest.xml での Safe Browsing の有効化設定

```
<?xml version="1.0" encoding="utf-8"?>
<manifest package="...">
    <application>
        ...
        <!-- アプリプロセス内の WebView の Safe Browsing 機能を明示的に有効化する -->
        <meta-data
            android:name="android.webkit.WebView.EnableSafeBrowsing"
            android:value="true" />
    </application>
</manifest>
```

また、Android 8.0 (API Level 26) において、Safe Browsing に関する API が幾つか追加された。

WebSettings クラスに追加された setSafeBrowsingEnabled(boolean enabled) は WebView のインスタンス単位で動的に有効・無効を設定するメソッドである。Android 8.0 (API Level 26) 未満では Safe Browsing 機能の有効・無効を AndroidManifest で設定していたが、これではアプリの全 WebView に対する一括の設定しかできなかった。setSafeBrowsingEnabled(boolean enabled) を用いることで WebView のインスタンス単位で動的に有効・無効を切り替えることが可能となった。

```
if (url == IN_HOUSE_MANAGEMENT_CONTENT_URL) {
    // 例）自社管理コンテンツが Safe Browsing で検出されてしまうので一時的に無効化する
    webView.getSettings().setSafeBrowsingEnabled(false);
} else {
    // 通常は有効化しておくべき
    webView.getSettings().setSafeBrowsingEnabled(true);
}
```

Android 8.1 (API Level 27) においても、Safe Browsing に関するクラス・API が追加された。これにより Safe Browsing の初期化処理の記述や、安全でない Web ページにアクセスした際の挙動の設定、特定のサイトを Safe Browsing の対象から外すためのホワイトリストの設定などが可能となった。

WebView クラスに追加された startSafeBrowsing() はアプリ内 WebView で使われる WebView コンポーネントの Safe Browsing 機能初期化処理を呼び出すメソッドである。第 2 引数で渡す Callback オブジェクトに初期化の結果が渡されるので、初期化が失敗し Callback オブジェクトに false が渡された場合、WebView を無効化する・URL をロードしないなどの対応が推奨される。

```
// Android 8.1 未満の端末ではサポートされないため、実際の実装時には端末の Android OS バージョンで処理を分ける必要がある
WebView.startSafeBrowsing(this, new ValueCallback<Boolean>() {
    @Override
```

(continues on next page)

(continued from previous page)

```
    public void onReceiveValue(Boolean result) {
        mSafeBrowsingIsInitialized = true;
        if (result) {
            Log.i("WebView SafeBrowsing", "Initialized SafeBrowsing!");
        } else {
            Log.w("WebView SafeBrowsing", "SafeBrowsing initialization failed...");
            // 初期化処理が失敗したため、Safe Browsing がうまく動作しない可能性がある
            // この場合 WebView を無効化することが望ましい
        }
    }
});
```

同じく WebView クラスに追加された setSafeBrowsingWhitelist() は Safe Browsing の対象から外したいホスト名・IP アドレスをホワイトリスト形式で設定するメソッドである。Safe Browsing の対象から外したい Host 名・IP アドレスのリストを引数で渡すとそれらにアクセスする際に Safe Browsing による検証が行われなくなる。

```
// Safe Browsing 機能を適用させない Host 名・IP アドレスのホワイトリストを設定する
// 例）自社管理コンテンツが Safe Browsing で検出されてしまうのでホワイトリストに登録する
WebView.setSafeBrowsingWhitelist(new ArrayList<>(Arrays.asList( IN_HOUSE_MANAGEMENT_CONTENT_HOSTNAME )),
        new ValueCallback<Boolean>() {
            @Override
            public void onReceiveValue(Boolean aBoolean) {
                Log.i("WebView SafeBrowsing", "Whitelisted " + aBoolean.toString());
            }
        });
```

WebClient クラスに追加された onSafeBrowsingHit() は、Safe Browsing が有効化された WebView でアクセスした URL が安全でない Web ページだと判定された際に Callback される Callback 関数である。第 1 引数に安全でない Web ページにアクセスした WebView の Object、第 2 引数に WebResourceRequest、第 3 引数に threat の種類、第 4 引数に判定に対する挙動を設定するための SafeBrowsingResponse オブジェクトが渡される。

SafeBrowsingResponse オブジェクトを使って選択できる挙動には、以下の 3 つがある。

- backToSafety(boolean report): 警告は表示せず前のページに戻る (前のページがない場合、blank ページを表示)
- proceed(boolean report): 警告を無視して Web ページを表示
- showInterstitial(boolean allowReporting): 警告ページを表示 (デフォルトの挙動)

backToSafety() および proceed() は引数で Google にレポートを送信するか否かを引数で設定でき、showInterstitial() は「Google にレポートを送信するかの選択を行うチェックボックス」を表示させるかを引数で設定できる。

```
public class MyWebViewClient extends WebViewClient {

    // Safe Browsing 機能が有効化された状態で安全でない Web ページにアクセスすると Callback される
    @Override
    public void onSafeBrowsingHit(WebView view, WebResourceRequest request,
                                  int threatType, SafeBrowsingResponse callback) {
        callback.showInterstitial(true); // 「Google にレポートを送信するか」を選択するチェックボックスを持った警
告ページを表示する （推奨）
        callback.backToSafety(true); // 警告ページは表示せず安全なページに戻し、Google にレポートを送信する␣
↪（推奨）
        callback.proceed(false); // 警告を無視しページにアクセスし、Google にレポートを送信する （非推奨）
```

(continues on next page)

(continued from previous page)

```
    }
}
```

これらのクラス・API には対応する Android Support Library が無い。したがってこれらのクラス・API を利用するアプリを API Level 26 または 27 未満の端末で動作させたい場合はバージョンによって処理を分けるなどの措置が必要である。

WebView で Safe Browsing を利用し、安全でない Web ページへのアクセスをハンドリングするためのサンプルコードを以下に示す。

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="org.jssec.android.webview.safebrowsing">

    <uses-permission android:name="android.permission.INTERNET" />

    <application
        android:allowBackup="false"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme"
        android:networkSecurityConfig="@xml/network_security_config">
        <activity
            android:name=".MainActivity"
            android:exported="true"
            android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

        <!-- アプリプロセス内の WebView の Safe Browsing 機能を有効化する -->
        <meta-data
            android:name="android.webkit.WebView.EnableSafeBrowsing"
            android:value="true" />
    </application>
</manifest>
```

MainActivity.java

```
package org.jssec.android.webview.safebrowsing;

import android.support.v7.app.AppCompatActivity;

import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.webkit.ValueCallback;
import android.webkit.WebView;

import java.util.ArrayList;
import java.util.Arrays;

public class MainActivity extends AppCompatActivity {
```

(continues on next page)

(continued from previous page)

```
    private boolean mSafeBrowsingIsInitialized;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        findViewById(R.id.button1).setOnClickListener(setWhiteList);
        findViewById(R.id.button2).setOnClickListener(reload);

        final WebView webView = findViewById(R.id.webView);
        webView.setWebViewClient(new MyWebViewClient());

        mSafeBrowsingIsInitialized = false;
        // Android 8.1 未満の端末ではサポートされないため、実際の実装時には端末の Android OS バージョンで処理を分ける必要がある
        WebView.startSafeBrowsing(this, new ValueCallback<Boolean>() {
            @Override
            public void onReceiveValue(Boolean result) {
                mSafeBrowsingIsInitialized = true;
                if (result) {
                    Log.i("WebView SafeBrowsing", "Initialized SafeBrowsing!");
                    webView.loadUrl("http://testsafebrowsing.appspot.com/s/malware.html");
                } else {
                    Log.w("WebView SafeBrowsing", "SafeBrowsing initialization failed...");
                    // 初期化処理が失敗したため、Safe Browsing がうまく動作しない可能性がある
                    // この場合 URL をロードしないことが望ましい
                }
            }
        });
    }

    View.OnClickListener setWhiteList = new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            // Safe Browsing 機能を適用させない Host 名・IP アドレスのホワイトリストを設定する
            WebView.setSafeBrowsingWhitelist(new ArrayList<>(Arrays.asList("testsafebrowsing.appspot.com
↪")),
                    new ValueCallback<Boolean>() {
                        @Override
                        public void onReceiveValue(Boolean aBoolean) {
                            Log.i("WebView SafeBrowsing", "Whitelisted " + aBoolean.toString());
                        }
                    });
        }
    };

    View.OnClickListener reload = new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            final WebView webView = findViewById(R.id.webView);
            webView.reload();
        }
    };
```

(continues on next page)

(continued from previous page)

```
}
```

MyWebViewClient.java

```
package org.jssec.android.webview.safebrowsing;

import android.app.AlertDialog;
import android.app.Dialog;
import android.content.Context;
import android.content.Intent;
import android.webkit.SafeBrowsingResponse;
import android.webkit.WebResourceRequest;
import android.webkit.WebView;
import android.webkit.WebViewClient;
import android.widget.Toast;

public class MyWebViewClient extends WebViewClient {

    // Safe Browsing 機能が有効化された状態で安全でない Web ページにアクセスすると Callback される
    @Override
    public void onSafeBrowsingHit(WebView view, WebResourceRequest request,
                                  int threatType, SafeBrowsingResponse callback) {
        // 警告ページは表示せず安全なページに戻す
        callback.backToSafety(true);
        Toast.makeText(view.getContext(), "アクセスしようとした Web ページはマルウェアサイトの疑いがあるため、安全
なページに戻ります。", Toast.LENGTH_LONG).show();
    }
}
```

4.10 Notification を使用する

Android にはエンドユーザーへのメッセージを通知する Notification 機能がある。Notification を使うと、画面上部のステータスバーと呼ばれる領域に、アイコンやメッセージを表示することができる。

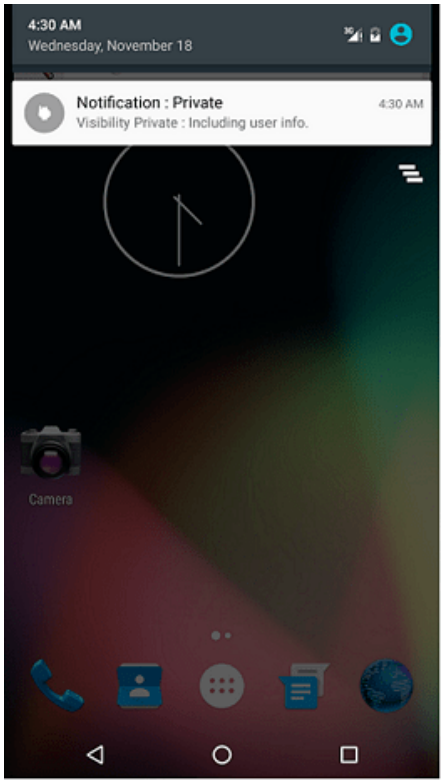

図 4.10.1　Notifcation の表示例

Notification の通知機能は、Android 5.0(API Level 21) で強化され、アプリやユーザー設定によって、画面がロックされている状態であっても Notification による通知を表示することが可能になった。ただし、Notification の使い方を誤ると、端末ユーザー本人にのみ見せるべきプライベートな情報が第三者の目に触れる恐れがある。したがって、プライバシーやセキュリティを考慮して適切に実装を行うことが重要である。

なお、Visibility が取り得る値と Notification の振る舞いは以下の通りである。

表 4.10.1: Visibility が取り得る値と Notification の振る舞い

Visibility の値	**Notification** の振る舞い
Public	すべてのロック画面上で Notification が表示される
Private	すべてのロック画面上で Notifcation が表示されるが、パスワード等で保護されたロック画面（セキュアロック）上では、Notification のタイトルやテキスト等が隠される（プライベート情報が隠された公開可能な文に置き換わる）
Secret	パスワード等で保護されたロック画面（セキュアロック）上では、Notification が表示されなくなる（セキュアロック以外のロック画面では Notification は表示される）

4.10.1 サンプルコード

Notification に端末ユーザーのプライベートな情報を含む場合、プライベート情報を取り除いた通知を画面ロック時の表示用に作成し、加えておくこと。

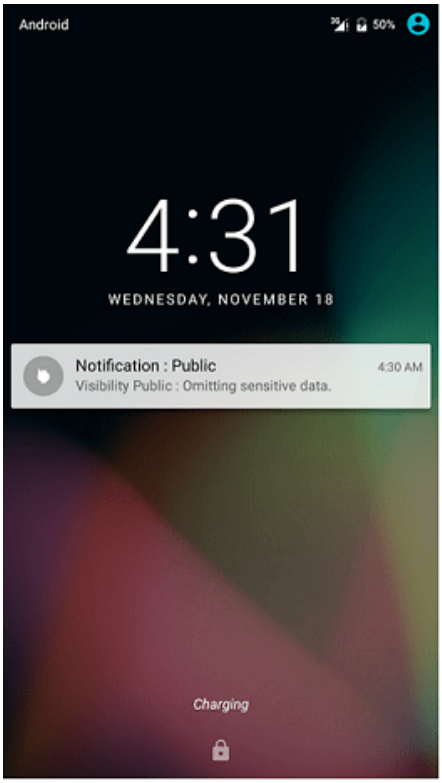

図 4.10.2　ロック画面上の Notification

プライベート情報を含んだ通知を行うサンプルコードを以下に示す。

ポイント：

1. プライベート情報を含んだ通知を行う場合は、公開用（画面ロック時の表示用）の Notification を用意する
2. 公開用（画面ロック時の表示用）の Notification にはプライベート情報を含めない
3. Visibility を明示的に Private に設定して、Notification を作成する
4. Visibility が Private の場合、プライベート情報を含めて通知してもよい

VisibilityPrivateNotificationActivity.java

```
package org.jssec.notification.visibilityPrivate;

import android.app.Activity;
import android.app.Notification;
import android.app.NotificationManager;
import android.content.Context;
import android.os.Build;
import android.os.Bundle;
import android.view.View;

public class VisibilityPrivateNotificationActivity extends Activity {
    /**
     * Private な Notification を表示する
     */
    private final int mNotificationId = 0;

    @Override
    public void onCreate(Bundle savedInstanceState) {
```

(continues on next page)

(continued from previous page)

```
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    public void onSendNotificationClick(View view) {
        // ★ポイント 1★  プライベート情報を含んだ通知を行う場合は、公開用（画面ロック時の表示用）の Notification
を用意する
        Notification.Builder publicNotificationBuilder = new Notification.Builder(this).setContentTitle(
↪"Notification : Public");

        if (Build.VERSION.SDK_INT >= 21)
            publicNotificationBuilder.setVisibility(Notification.VISIBILITY_PUBLIC);
        // ★ポイント 2★ 公開用（画面ロック時の表示用）の Notification にはプライベート情報を含めない
        publicNotificationBuilder.setContentText("Visibility Public : Omitting sensitive data.");
        publicNotificationBuilder.setSmallIcon(R.drawable.ic_launcher);
        Notification publicNotification = publicNotificationBuilder.build();

        // プライベート情報を含む Notification を作成する
        Notification.Builder privateNotificationBuilder = new Notification.Builder(this).setContentTitle(
↪"Notification : Private");

        // ★ポイント 3★ 明示的に Visibility を Private に設定して、Notification を作成する
        if (Build.VERSION.SDK_INT >= 21)
            privateNotificationBuilder.setVisibility(Notification.VISIBILITY_PRIVATE);
        // ★ポイント 4★ Visibility が Private の場合、プライベート情報を含めて通知してもよい
        privateNotificationBuilder.setContentText("Visibility Private : Including user info.");
        privateNotificationBuilder.setSmallIcon(R.drawable.ic_launcher);
        // Visibility が Private の Notification を利用する場合、Visibility を Public にした公開用の Notification
を合わせて設定する
        if (Build.VERSION.SDK_INT >= 21)
            privateNotificationBuilder.setPublicVersion(publicNotification);

        Notification privateNotification = privateNotificationBuilder.build();

        // 本サンプルでは実装していないが、Notification では setContentIntent(PendingIntent intent) を使い
        // Notification をクリックした際に Intent が送信されるように実装することが多い。
        // このときに設定する Intent は、呼び出すコンポーネントの種類に合わせて、
        // 安全な方法で呼び出すことが必要である（例えば、明示的 Intent を使うなど）
        // 各コンポーネントの安全な呼び出し方法は以下の項目を参照のこと
        // 4.1. Activity を作る・利用する
        // 4.2. Broadcast を受信する・送信する
        // 4.4. Service を作る・利用する

        NotificationManager notificationManager = (NotificationManager) this.getSystemService(Context.
↪NOTIFICATION_SERVICE);
        notificationManager.notify(mNotificationId, privateNotification);
    }
}
```

4.10.2 ルールブック

Notification を利用する際には以下のルールを守ること。

1. *Visibility* の設定に依らず、*Notification* にはセンシティブな情報を含めない（プライベート情報は例外） （必須）

2. *Visibility Public の Notification* には プライベート情報を含めない （必須）
3. （特に *Visibility Private* にする場合）*Visibility* は明示的に設定する （必須）
4. *Visibility* が *Private の Notification* を利用する場合、*Visibility* を *Public* にした公開用の *Notification* を併せて設定する （推奨）

4.10.2.1 Visibility の設定に依らず、Notification にはセンシティブな情報を含めない（プライベート情報は例外）（必須）

Android4.3(API Level 18) 以降の端末では、設定画面からユーザーが Notification の読み取り許可をアプリに与えることができる。許可されたアプリは、全ての Notification の情報を読み取ることが可能になるため、センシティブな情報を Notification に含めてはならない（ただし、プライベート情報は Visibility の設定によっては Notification に含めて良い）。

Notification に含まれた情報は、通常は Notification を送信したアプリを除き、他のアプリから読み取ることはできない。しかし、ユーザーが明示的に許可を与えることで、ユーザーが指定したアプリは全ての Notification の情報を読み取ることが可能になる。ユーザーが許可を与えたアプリのみが Notification の情報を読み取れることから、ユーザー自身のプライベート情報を Notification に含めることは問題ない。一方で、ユーザーのプライベート情報以外のセンシティブな情報（例えば、アプリ開発者のみが知り得る機密情報）を Notification に含めると、ユーザー自身が Notification に含まれた情報を読みとろうとして Notification への閲覧をアプリに許可する可能性があるため、利用者のプライベート情報以外のセンシティブな情報を含めることは問題となる。

具体的な方法と条件は、「4.10.3.1. ユーザー許可による *Notification* の閲覧について」を参照の事。

4.10.2.2 Visibility Public の Notification には プライベート情報を含めない （必須）

Visibility が Public に設定された Notification によって通知を行う場合、ユーザーのプライベート情報を Notification に含めてはならない。Visibility が Public に設定された Notification は、画面ロック中にも Notification の情報が表示され、端末に物理的に接近できる第三者がプライベート情報を盗み見るリスクにつながるためである。

```
VisibilityPrivateNotificationActivity.java
    // 公開用（画面ロック時の表示用）の センシティブな情報を持たない Notification を用意する
    Notification.Builder publicNotificationBuilder = new Notification.Builder(this).setContentTitle(
↪"Notification : Public");

    publicNotificationBuilder.setVisibility(Notification.VISIBILITY_PUBLIC);
    // 公開用（画面ロック時の表示用）の Notification にはプライベート情報を含めない
    publicNotificationBuilder.setContentText("Visibility Public : センシティブな情報は含めずに通知");
    publicNotificationBuilder.setSmallIcon(R.drawable.ic_launcher);
```

プライベート情報の典型例としては、ユーザー宛てに送信されたメールやユーザーの位置情報など、「5.5. プライバシー情報を扱う」で言及されている情報が挙げられる。

4.10.2.3 （特に Visibility Private にする場合）Visibility は明示的に設定する （必須）

「4.10.2.2. *Visibility Public の Notification* には プライベート情報を含めない （必須）」の通り、Android 5.0(API Level 21) 以降の端末では、画面ロック中にも Notification が表示されるため、Visibility の設定が重要であり、デフォルト値に頼らず明示的に設定すること。

現状では、Notification の Visibility のデフォルト値は Private に設定されており、明示的に Public を指定しない限りプライベート情報が盗み見られるリスクは発生しない。しかし、Visibility のデフォルト値が将来変更になる可能性もあ

り、含める情報の取り扱いを常に意識するためにも、たとえ Visiblity を Private にする場合であっても、Notification の Visibility は明示的に設定することを必須としている。

```
VisibilityPrivateNotificationActivity.java
    // プライベート情報を含む Notification を作成する
    Notification.Builder priavteNotificationBuilder = new Notification.Builder(this).setContentTitle(
↪"Notification : Private");

    // ★ポイント★ 明示的に Visibility を Private に設定して、Notification を作成する
    priavteNotificationBuilder.setVisibility(Notification.VISIBILITY_PRIVATE);
```

4.10.2.4 Visibility が Private の Notification を利用する場合、Visibility を Public にした公開用の Notification を併せて設定する （推奨）

Visibility が Private に設定された Notification を使って通知する場合、画面ロック中に表示される情報を制御するため、Visibility を Public にした公開用の Notification を併せて設定することが望ましい。

Visibility が Private に設定された Notification に公開用の Notification を設定しない場合、画面ロック中にはシステムで用意されたデフォルトの文言が表示されるためセキュリティ上の問題はない。しかし、Notification に含める情報の取り扱いを常に意識するためにも、公開用の Notification を明示的に用意し設定することを推奨する。

```
VisibilityPrivateNotificationActivity.java
    // プライベート情報を含む Notification を作成する
    Notification.Builder privateNotificationBuilder = new Notification.Builder(this).setContentTitle(
↪"Notification : Private");

    // ★ポイント★ 明示的に Visibility を Private に設定して、Notification を作成する
    if (Build.VERSION.SDK_INT >= 21)
        privateNotificationBuilder.setVisibility(Notification.VISIBILITY_PRIVATE);
    // ★ポイント★ Visibility が Private の場合、プライベート情報を含めて通知してもよい
    privateNotificationBuilder.setContentText("Visibility Private : Including user info.");
    privateNotificationBuilder.setSmallIcon(R.drawable.ic_launcher);
    // Visibility が Private の Notification を利用する場合、Visibility を Public にした公開用の Notification を合
わせて設定する
    if (Build.VERSION.SDK_INT >= 21)
        privateNotificationBuilder.setPublicVersion(publicNotification);
```

4.10.3 アドバンスト

4.10.3.1 ユーザー許可による Notification の閲覧について

「4.10.2.1. *Visibility* の設定に依らず、*Notification* にはセンシティブな情報を含めない（プライベート情報は例外） （必須）」で述べたように、Android4.3(API Level 18) 以降の端末では、ユーザーが許可を与えた場合、指定されたアプリは全ての Notification の情報を読み取ることが可能になる。ただし、ユーザー許可の対象となるためには、アプリが NotificationListenerService を継承した Service を実装しておく必要がある。

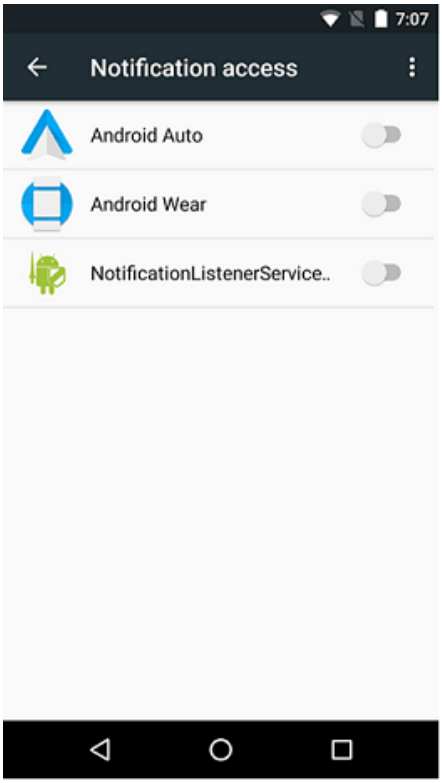

図 4.10.3　Notification の読み取りを設定する「通知へのアクセス」画面

NotificationListenerService を使ったサンプルコードを以下に示す。

AndroidManifest.xml

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="org.jssec.notification.notificationListenerService">

     <application
         android:allowBackup="false"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name" >
         <service android:name=".MyNotificationListenerService"
             android:label="@string/app_name"
             android:permission="android.permission.BIND_NOTIFICATION_LISTENER_SERVICE">
             <intent-filter>
                 <action android:name=
                     "android.service.notification.NotificationListenerService" />
             </intent-filter>
         </service>
     </application>
</manifest>
```

MyNotificationListenerService.java

```
package org.jssec.notification.notificationListenerService;

import android.app.Notification;
import android.service.notification.NotificationListenerService;
import android.service.notification.StatusBarNotification;
import android.util.Log;

public class MyNotificationListenerService extends NotificationListenerService {
```

(continues on next page)

(continued from previous page)

```
    @Override
    public void onNotificationPosted(StatusBarNotification sbn) {
        // Notification is posted.
        outputNotificationData(sbn, "Notification Posted : ");
    }

    @Override
    public void onNotificationRemoved(StatusBarNotification sbn) {
        // Notification is deleted.
        outputNotificationData(sbn, "Notification Deleted : ");
    }

    private void outputNotificationData(StatusBarNotification sbn, String prefix) {
        Notification notification = sbn.getNotification();
        int notificationID = sbn.getId();
        String packageName = sbn.getPackageName();
        long PostTime = sbn.getPostTime();

        String message = prefix + "Visibility :" + notification.visibility + " ID : " + notificationID;
        message += " Package : " + packageName + " PostTime : " + PostTime;

        Log.d("NotificationListen", message);
    }
}
```

上記の通り、NotificationListenerService を使い、ユーザーの許可を得ることで、Notification を読み取ることが可能になるが、Notification に含まれる情報には端末のプライベート情報が含まれることが多いため、取り扱いには十分な注意が必要である。

4.11 共有メモリを使用する

Android OS には以前から共有メモリ機構が存在しており android.os.MemoryFile として提供されていたが、複数のアプリで共有するための API やアクセス制御が直接的には提供されておらず、一般のアプリには使いにくいものであった。Android 8.1 (API Level 27) で android.os.SharedMemory パッケージが導入され、一般のアプリからも共有メモリ機構を比較的簡単に利用できるようになった。Android 8.1 の時点で MemoryFile は SharedMemory のラッパーとなり、SharedMemory の方を利用することが推奨されている。ここでは、この SharedMemory API を使用する際にセキュリティ上で注意すべき点を解説する。

後述するようにこの API は、あるアプリの Service が共有メモリを作成し、それらを他のアプリに提供することで提供されたアプリとメモリを共有する構成を前提とした作りになっている。したがって「4.4. *Service* を作る・利用する」で解説されているすべての内容が、共有メモリを提供するアプリおよびそれを利用するアプリについても当てはまる。未読の方は先に「4.4. *Service* を作る・利用する」を読んでから、以下の説明に進むことを推奨する。

なお、SharedMemory API には対応する Android Support Library が無い。したがって SharedMemory を利用するアプリを API Level 27 未満の端末で動作させたい場合は、C 言語レベルの API を JNI でラップするなどして同等の仮想メモリ機構を実装し、バージョンによって使い分けるなどの措置が必要である。

4.11.1 Android 共有メモリの概要

共有メモリは複数のアプリの間で同一の物理メモリ領域を共有する機構である。

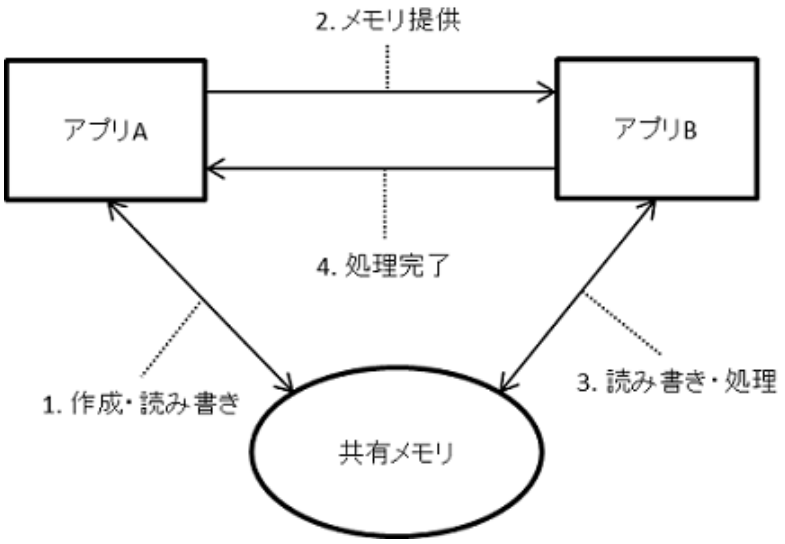

図 4.11.1　共有メモリの概要

上の図ではアプリ A とアプリ B で共有メモリを使用している様子を示している。アプリ A が共有メモリオブジェクトを作成し、それをアプリ B に提供している。アプリ A で共有メモリを提供する役割はアプリ A の Service が担う。アプリ B はこの Service に接続して共有メモリを要求・取得し、共有メモリで必要な処理が済んだら、利用が完了したことをアプリ A に知らせる。

共有メモリを利用することによって、例えば大きな画像のビットマップデータなど、通常のプロセス間通信で許されている最大サイズ (1MB[*30]) を超えるデータを複数のプロセスで共有することができる。また、デバイス全体で消費するメモリ量を削減でき、通常のメモリアクセスとなるため、非常に高速なプロセス間通信が実現できる。しかし、複数のアプリが同時並行的にアクセスすることになるため、データの整合性を維持するための考慮が必要となる場合がある。これを避けるためにはアプリ間での排他制御を行ったり、メモリ領域を適切に分割して互いのアクセスに干渉しないように考慮するなどの注意深い設計が必要である。

上述のように Android SDK の共有メモリ API は、Service が共有メモリオブジェクトを作成して他のプロセスへ供給する形態をとる。共有メモリのクラス (android.os.SharedMemory) は Parcelable として定義されているため、共有メモリのインスタンスを Binder 経由で他のプロセスに容易に渡すことができる。後で示すサンプルコードでの Service とクライアントとの間のやりとりの概要は（Service の作りによって大きく異なることもあるが)、次の図に示すような構造となる：

[*30] https://developer.android.com/guide/components/activities/parcelables-and-bundles

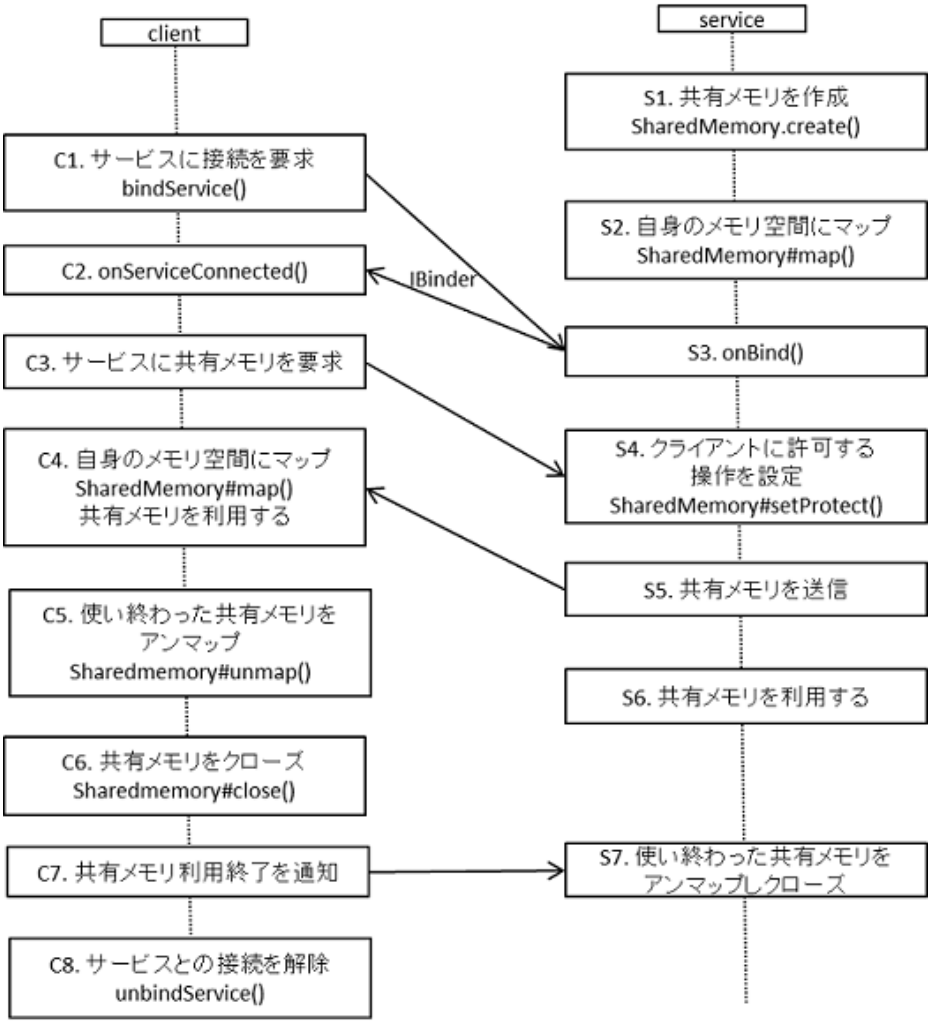

図 4.11.2　共有メモリ Service とクライアントのやりとり

- S1. Service は SharedMemory.create() により共有メモリを作成する
- S2. Service 自身が共有メモリを利用する場合 SharedMemory#map() によって自身のメモリ空間に共有メモリをマップする
- C1. クラアントは明示的 Intent を使用し Context#bindService() によってサービスに接続する
- S3. クライアントからの接続要求が届くと、Service の onBind() call back が呼び出される。Service はここで (あれば) 必要な前処理を行い、クライアントへ接続用の IBinder を返す。
- C2. Service が onBind() を実行した際の戻り値 (IBinder のインスタンス) は、クライアント側の onServiceConnected() callback の引数として返される。以降はこの IBinder を利用して Service との通信を行う。
- C3. クライアントは Service に対して共有メモリを要求する。
- S4. Service はクライアントからの共有メモリ要求を受け、クラアントが共有メモリをアクセスする際に許可する操作 (読み・書き) を設定する。
- S5. Service はクライアントへ共有メモリオブジェクトを渡す。
- C4. クライアントは受け取った共有メモリにアクセスするため、自身のアドレス空間へ共有メモリをマップし、利用する。
- C5, C6. クライアントが共有メモリ利用が終了したとき、自身のメモリ空間からアンマップし (C5)、共有メモリを

クローズ (C6) する。

- C7. クライアントはその後、共有メモリの利用が終了したことをサーバーへ通知する。
- C8. クライアントは Service との接続を解除する。
- S7. Service はクライアントから利用終了のメッセージを受け取ったのちに、自身も共有メモリをアンマップしクローズする。

上の C2. 項の onServiceConnected() は、android.content.ServiceConnection クラスを実装したクラスで定義する。具体例は後で示すサンプルコードを参照してほしい。IBinder を利用した通信方法にはいくつかのやり方があるが、サンプルコードでは Messenger を利用している。

4.11.2 サンプルコード

先に述べた通り、共有メモリを作成して他のアプリに提供する側は Service として実装する。そのため、機能や情報共有のセキュリティという観点からは「4.4. *Service* を作る・利用する」で述べられている事と本質的な違いはない。下図に 4.4. の分類に倣い、どのような相手とメモリを共有するかという視点で判定フローを示す。

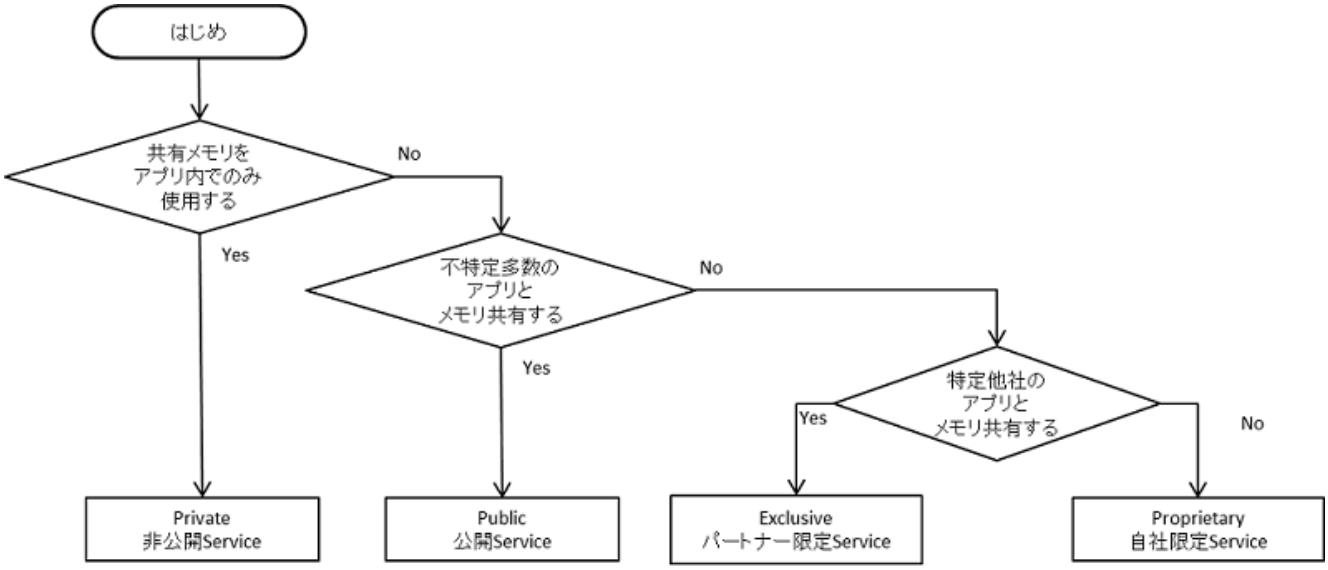

図 4.11.3　共有メモリ Service タイプの選択フロー

「4.4.1. サンプルコード」の 表 4.4.1 で、Service の実装方法について述べられているが、共有メモリの場合は、他のアプリとの共有は binder 経由で行う必要がある。そのため startService 型や IntentService 型の Service として実装することはできない。したがって下の表のようになる。

表 4.11.1: Service の実装方法（共有メモリの場合）

分類	非公開 **Service**	公開 **Service**	パートナー限定 **Service**	自社限定 **Service**
startService 型	-	-	-	-
IntentService 型	-	-	-	-
local bind 型	o	-	-	-
Messenger bind 型	o	o	-	**o***
AIDL bind 型	o	o	o	o

全体の作りについては、「4.4.1. サンプルコード」に掲載したものと本質的に同じである。また共有メモリに特有の事項については全てのケースで共通のため、具体的なサンプルコードは上の表で*印のついた、自社限定 Service に限定して示す。したがって他のケースで共有メモリを利用する場合は、「4.4.1.1. 非公開 *Service* を作る・利用する」から「4.4.1.3. パートナー限定 *Service*」にかけて記載されている内容を参照されたい。

4.11.2.1 非公開 Service を作る・利用する

この場合アプリに含まれる複数のプロセス間で非公開 Service が作成した共有メモリを共有する構造となる。またこの非公開 Service はアプリのメインプロセスとは独立したプロセスとして起動することとなる。

ポイント:

1. 共有メモリを作成する Service は exported=”false” により、明示的に非公開設定する
2. あるアプリ内プロセスが他のプロセスが書き込んだデータを参照する場合、同一アプリ内のプロセスであっても安全性を確認する
3. メモリを共有するのは同一アプリ内のプロセスであるから、センシティブな情報を共有しても良い

「4.4.1.1. 非公開 *Service* を作る・利用する」のサンプルコードでは Intent により Service を利用していたが、共有メモリの場合は Intent 経由ではメモリリソースを共有することができないため、local bind 型、Message bind 型、AIDL bind 型のいずれかの方法を用いる必要がある。

4.11.2.2 公開 Service を作る・利用する

「4.4.1.2. 公開 *Service* を作る・利用する」で述べた通り、公開 Service は不特定多数のアプリに利用されることを想定した Service である。そのためマルウェアが利用することも想定しなければならない。基本的に 4.4.1.2. のポイントで述べた通りの注意が必要であるが、以下ではこれを共有メモリの立場から言い換えてポイントを示す。

ポイント (Service を作る) :

1. exported=”true” により、明示的に公開設定する
2. サービスの起動やメモリ共有のためのリクエストなどに含まれるパラメータやデータの安全性を確認する
3. 共有メモリによってセンシティブな情報を共有しない

ポイント (Service を利用する):

1. センシティブな情報を共有メモリに書き込んではならない
2. 他のアプリが書き込んだデータを参照する場合は、その安全性を確認する

4.11.2.3 パートナー限定 Service

「4.4.1.3. パートナー限定 *Service*」で示されていることと本質的に同様であるが共有メモリの観点から言い換えて、以下にポイントを示す (4.4.1.3. のサンプルコードと同じく、AIDL bind 型の Service を想定する)。

ポイント (Service を作る) :

1. Intent Filter を定義せず、exported=”true” を明示的に宣言する
2. 利用元アプリの証明書がホワイトリストに登録されていることを確認する

3. onBind(onStartCommand, onHandleIntent) で呼び出し元がパートナーかどうか判別できない
4. パートナーアプリからの Intent であっても、受信 Intent の安全性を確認する
5. パートナーアプリに開示してよい情報に限り共有メモリに書き込んでも良い

ポイント (Service を使う):

1. 利用先のパートナー限定 Service アプリの証明書がホワイトリストに登録されていることを確認する
2. 利用先パートナー限定アプリに開示してよい情報のみ共有メモリに書き込んでも良い
3. 明示的 Intent によりパートナー限定 Service を呼び出す
4. パートナー限定アプリが書き込んだ情報であっても、データの安全性を確認する

4.11.2.4 自社限定 Service

ここでは、外部に公開された Service で共有メモリを提供するが、共有メモリを提供する先は自社製アプリに限定する例を示す。「4.4.1.4. 自社限定 *Service*」の例と同じく Messenger bind 型の Service となっている。背景にある考え方や設定については 4.4.1.4. に解説があるので、未読の方は 4.4.1.4. を先に参照されたい。

Service 側アプリのサンプルコード (Messenger bind 型)

以下にポイントを示すが、1 項から 5 項および 7 項は「4.4.1.4. 自社限定 *Service*」で提示されているものであり、共有メモリ特有のものは 6 項のみである。

ポイント:

1. 独自定義 Signature Permission を定義する
2. 独自定義 Signature Permission を要求宣言する
3. Intent Filter を定義せず、exported=”true” を明示的に宣言する
4. 独自定義 Signature Permission が自社アプリによって定義されていることを確認する
5. 自社アプリからの Intent であっても、受信 Intent の安全性を確認する
6. クライアントに共有メモリを渡す前に SharedMemory#setProtect() によってクライアント側が可能な操作を制限する
7. 利用元アプリと同じ開発者鍵で APK を署名する

この例では簡単のため共有メモリをアロケートする Service とそれを利用する Activity を同じアプリケーション内で定義している (Service は同じアプリ内の別プロセスとして起動されるようにしている)。そのため独自定義 Signature Permission の定義と利用宣言が両方ともに Manifest ファイルに記載されている。

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="org.jssec.android.sharedmemory.inhouseservice.messenger">
    <!-- ★ ポイント 1 ★ 独自定義 Signature Permission を定義する -->
    <permission android:name="org.jssec.android.sharedmemory.inhouseservice.messenger.MY_PERMISSION"
        android:protectionLevel="signature" />
```

(continues on next page)

(continued from previous page)

```
    <uses-permission
        android:name="org.jssec.android.service.inhouseservice.messenger.MY_PERMISSION" />
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name="org.jssec.android.sharedmemory.inhouseservice.messenger.MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <!-- Messenger を利用した Service -->
        <!-- ★ ポイント 2 ★ 独自定義 Signature Permission を要求宣言する -->
        <!-- ★ ポイント 3 ★ Intent Filter を定義せず exported="true"を明示的に設定する -->
        <!-- 簡単のため共有メモリを提供するサービスを同じアプリの別プロセスとする -->
        <service android:name="org.jssec.android.sharedmemory.inhouseservice.messenger.SHMService"
            android:exported="true"
            android:permission="org.jssec.android.sharedmemory.inhouseservice.messenger.MY_PERMISSION"
            android:process=".shmService" />
    </application>
</manifest>
```

SHMService.java

```
package org.jssec.android.sharedmemory.inhouseservice.messenger;

import org.jssec.android.shared.SigPerm;
import org.jssec.android.shared.Utils;

import android.app.Service;
import android.content.Context;
import android.content.Intent;
import android.os.Handler;
import android.os.IBinder;
import android.os.Message;
import android.os.Messenger;
import android.os.RemoteException;
import android.os.SharedMemory;
import android.system.ErrnoException;
import android.util.Log;
import android.widget.Toast;

import java.nio.ByteBuffer;

import static android.system.OsConstants.PROT_READ;
import static android.system.OsConstants.PROT_WRITE;


public class SHMService extends Service {

```

(continues on next page)

(continued from previous page)

```
    // 自社の Signature Permission
    private static final String MY_PERMISSION = "org.jssec.android.sharedmemory.inhouseservice.messenger.
↪MY_PERMISSION";

    // 自社の証明書のハッシュ値
    private static String sMyCertHash = null;
    private static String myCertHash(Context context) {
        if (sMyCertHash == null) {
            if (Utils.isDebuggable(context)) {
                // debug.keystore の"androiddebugkey"の証明書ハッシュ値
                sMyCertHash = "0EFB7236 328348A9 89718BAD DF57F544 D5CCB4AE B9DB34BC 1E29DD26 F77C8255";
            } else {
                // keystore の"my company key"の証明書ハッシュ値
                sMyCertHash = "D397D343 A5CBC10F 4EDDEB7C A10062DE 5690984F 1FB9E88B D7B3A7C2 42E142CA";
            }
        }
        return sMyCertHash;
    }

    private final String TAG = "SHM";

    // クライアントへ送るメッセージ文字列
    private final String greeting = "Hi! I send you my memory. Let's Share it!";
    private final String greeting2 = "You can write here!";
    private final String greeting3 = "From this point, I'll also write.";
    // ページサイズは 4K bytes
    public static final int PAGE_SIZE = 1024 * 4;
    // 共有メモリを２つ使用する。クライアントは下の識別子を指定し、どの共有メモリが欲しいかを指定する
    public static final int SHMEM1 = 0;
    public static final int SHMEM2 = 1;
    // 共有メモリのインスタンス
    // mSHMem1: クライアントへ渡すデータ用として使用する
    private SharedMemory mSHMem1 = null;
    // mSHMem をマップするためのバイトバッファ
    private ByteBuffer m1Buffer1;
    // mSHMem2: クライアントからのデータ受け取り用
    private SharedMemory mSHMem2 = null;
    // mSHMem2 をマップするためのバイトバッファ
    private ByteBuffer m2Buffer1;
    private ByteBuffer m2Buffer2;
    // すべてのバイトバッファが正常にマップされたときに true
    private boolean mBufferMapped = false;

    // この例ではクライアントとは Messenger 機構を利用して通信する
    // 以下はそこで使用するクライアントからのメッセージ識別子
    public static final int MSG_INVALID = Integer.MIN_VALUE;
    public static final int MSG_ATTACH  = MSG_INVALID + 1; // クライアントが SHMEM1 アロケート要求
    public static final int MSG_ATTACH2 = MSG_ATTACH + 1;  // クライアントが SHMEM2 アロケート要求
    public static final int MSG_DETACH  = MSG_ATTACH2 + 1; // クライアントが SHMEM1 を使い終わった
    public static final int MSG_DETACH2 = MSG_DETACH + 1;  // クライアントが SHMEM2 を使い終わった
    public static final int MSG_REPLY1  = MSG_DETACH2 + 1; // クライアントからの返答１
    public static final int MSG_REPLY2  = MSG_REPLY1 + 1;  // クライアントからの返答 2
    public static final int MSG_END     = MSG_REPLY2 + 1;  // Service が終了を宣言
```

(continues on next page)

(continued from previous page)

```
    // クライアントから受け取った Message を処理する Handler
    private class CommHandler extends Handler {
        @Override
        public void handleMessage(Message msg) {
            switch (msg.what) {
                case MSG_ATTACH:
                    shareWith1(msg);
                    break;
                case MSG_ATTACH2:
                    shareWith2(msg);
                    break;
                case MSG_DETACH:
                    unShare(msg);
                    break;
                case MSG_REPLY1:
                    gotReply(msg);
                    break;
                case MSG_REPLY2:
                    gotReply2(msg);
                    break;
                default:
                    invalidMsg(msg);
            }
        }
    }

    private final Handler mHandler = new CommHandler();

    // クライアントからのデータを受信するときに利用する Messenger
    private final Messenger mMessenger = new Messenger(mHandler);


    // バインド時に Messenger から Binder を取得しクライアントへ渡す
    @Override
    public IBinder onBind(Intent intent) {
        // ★ポイント 4★ 独自定義 Signature Permission が自社アプリにより定義されていることを確認する
        if (!SigPerm.test(this, MY_PERMISSION, myCertHash(this))) {
            Toast.makeText(this, "独自定義 Signature Permission が自社アプリによって定義されていない。",␣
↪Toast.LENGTH_LONG).show();
            return null;
        }
        // ★ポイント 5★ 自社アプリからの Intent であっても、受信 Intent の安全性を確認する
        // サンプルにつき割愛。「3.2 入力データの安全性を確認する」を参照。
        String param = intent.getStringExtra("PARAM");
        Log.d(TAG, String.format("パラメータ [%s] を受け取った！ ", param));
        return mMessenger.getBinder();
    }

    // マップ構造
    // オフセットはページ境界でなければならない
    public static final int SHMEM1_BUF1_OFFSET = 0;
    public static final int SHMEM1_BUF1_LENGTH = 1024;
    public static final int SHMEM2_BUF1_OFFSET = 0;
    public static final int SHMEM2_BUF1_LENGTH = 1024;
    public static final int SHMEM2_BUF2_OFFSET = PAGE_SIZE;
```

(continues on next page)

(continued from previous page)

```
    public static final int SHMEM2_BUF2_LENGTH = 128;

    // ２つの共有メモリをアロケートする
    private boolean allocateSharedMemory() {
        try {
            // クライアントへ渡すデータ用共有メモリ
            mSHMem1 = SharedMemory.create("SHM", PAGE_SIZE);
            // クライアントからのデータ受け取り用共有メモリ
            mSHMem2 = SharedMemory.create("SHM2", PAGE_SIZE * 2);
        } catch (ErrnoException e) {
            Log.e(TAG, "failed to allocate shared memory" + e.getMessage());
            return false;
        }
        return true;
    }

    // 指定の共有メモリをマップする
    private ByteBuffer mapShared(SharedMemory mem, int prot, int offset, int size) {
        ByteBuffer tBuf ;
            tBuf = mem.map(prot, offset, size);
        } catch (ErrnoException e) {
            Log.e(TAG, "could not map, prot=" + prot + ", offset=" + offset + ", length=" + size + "\n "␣
↪+ e.getMessage() + "err no. = " + e.errno);
            return null;
        } catch (IllegalArgumentException e){
            Log.e(TAG, "map failed: " + e.getMessage());
            return null;
        }
        return tBuf;
    }

    // 自分用に二つの共有メモリをマップ
    private void mapMemory() {
        // mSHMem1: 読み書きする
        m1Buffer1 = mapShared(mSHMem1, PROT_READ | PROT_WRITE, SHMEM1_BUF1_OFFSET, SHMEM1_BUF1_LENGTH);
        // mSHMem2: 二つの領域に分け、それぞれ読み書きする
        m2Buffer1 = mapShared(mSHMem2, PROT_READ | PROT_WRITE, SHMEM2_BUF1_OFFSET, SHMEM2_BUF1_LENGTH);
        m2Buffer2 = mapShared(mSHMem2, PROT_READ | PROT_WRITE, SHMEM2_BUF2_OFFSET, SHMEM2_BUF2_LENGTH);

        if (m1Buffer1 != null && m2Buffer1 != null && m2Buffer2 != null) mBufferMapped = true;
    }

    // 共有メモリを解放する
    private void deAllocateSharedMemory () {
        if (mBufferMapped) {
            if (mSHMem1 != null) {
                if (m1Buffer1 != null) SharedMemory.unmap(m1Buffer1);
                m1Buffer1 = null;
                mSHMem1.close();
                mSHMem1 = null;
            }

            if (mSHMem2 != null) {
                if (m2Buffer1 != null) SharedMemory.unmap(m2Buffer1);
                if (m2Buffer2 != null) SharedMemory.unmap(m2Buffer2);
```

(continues on next page)

(continued from previous page)

```
                m2Buffer1 = null;
                m2Buffer2 = null;
                mSHMem2.close();
                mSHMem2 = null;
            }
            mBufferMapped = false;
        }
    }


    @Override
    public void onCreate() {
        super.onCreate();

        // インスタンス生成のタイミングで共有メモリをアロケート
        // アロケートできたら共有メモリをマップする
        if (allocateSharedMemory()) {
            mapMemory();
        }
    }

    // SHMEM1 をクライアントへ提供
    public void shareWith1(Message msg){
        // 正常にアロケート・マップていなかったらなにもしない
        if (!mBufferMapped) return;

        // ★ポイント 6★ クライアントに共有メモリを渡す前に setProtect() によってクライアント側が可能な操作を制限
する
        // mSHMem1 はクライアントに読み込みのみを許可する
        mSHMem1.setProtect(PROT_READ);

        // 上の setProtect(PROT_READ) 実行以前にマッピングが完了しているため、書き込みが可能
        // mBuffer の先頭にメッセージのサイズを入れ、続いてメッセージ文字列を追加
        m1Buffer1.putInt(greeting.length());
        m1Buffer1.put(greeting.getBytes());

        try {
            // 要求元に共有メモリオブジェクトを渡す
            Message sMsg = Message.obtain(null, SHMEM1, mSHMem1);
            msg.replyTo.send(sMsg);
        } catch (RemoteException e) {
            Log.e(TAG, "Failed to share" + e.getMessage());
        }
    }

    // SHMEM2 をクライアントへ提供
    public void shareWith2(Message msg) {

        if (!mBufferMapped) return;

        // ★ポイント 6★ クライアントに共有メモリを渡す前に setProtect() によってクライアント側が可能な操作を制限
する
        // mSHMem2 はクライアントに書き込みを許可する
        mSHMem2.setProtect(PROT_WRITE);
        // クライアントに送るメッセージをそれぞれのバッファにセットする
```

(continues on next page)

(continued from previous page)

```
        m2Buffer1.putInt(greeting2.length());
        m2Buffer1.put(greeting2.getBytes());
        m2Buffer2.putInt(greeting3.length());
        m2Buffer2.put(greeting3.getBytes());
        try {
            // 要求元に共有メモリオブジェクトを送る
            Message sMsg = Message.obtain(null, SHMEM2, mSHMem2);
            msg.replyTo.send(sMsg);
        } catch (RemoteException e){
            Log.e(TAG, "failed to share mSHMem2" + e.getMessage());
        }
    }

   // メモリの共有を止める
    public void unShare(Message msg){
        deAllocateSharedMemory();
    }

    // 不正なメッセージを受け取った
    public void invalidMsg(Message msg){
        Log.e(TAG, "Got an Invalid message: " + msg.what);
    }

    // クライアントから受け取った情報を取り出す
    // 先頭にサイズ、それに続いて文字バイト列が格納されているものとする
    private String extractReply (ByteBuffer buf){
        int len = buf.getInt();
        byte [] bytes = new byte[len];
        buf.get(bytes);
        return new String(bytes);
    }

    // この例では 2 種類のメッセージをクライアントから受け取る
    // こちらは m1Buffer1 にクライアントからの情報が格納されていることを想定
    public void gotReply(Message msg) {
        m1Buffer1.rewind();
        String message = extractReply(m1Buffer1);
        if (!message.equals(greeting)){
            Log.e(TAG, "my message was overwritten: " + message);
        }
    }


    // こちらは m2Buffer1 にクライアントからの情報が格納されていることを想定
    public void gotReply2(Message msg) {
        m2Buffer1.rewind();
        String message = extractReply(m2Buffer1);
        android.util.Log.d(TAG, "got a message of length " + message.length() + " from client: " +␣
↪message);
        // このリプライメッセージが来た場合は、共有メモリの利用を終了することをクライアントと取り決めている
        Message eMsg = Message.obtain();
        eMsg.what = MSG_END;
        try {
            msg.replyTo.send(eMsg);
        } catch (RemoteException e){
```

(continues on next page)

(continued from previous page)

```
            Log.e(TAG, "error in reply 2: " + e.getMessage());
        }
    }
}
```

SigPerm.java

```
package org.jssec.android.shared;

import android.content.Context;
import android.content.pm.PackageManager;
import android.content.pm.PackageManager.NameNotFoundException;
import android.content.pm.PermissionInfo;
import android.os.Build;

import static android.content.pm.PackageManager.CERT_INPUT_SHA256;

public class SigPerm {

    public static boolean test(Context ctx, String sigPermName, String correctHash) {
        if (correctHash == null) return false;
        correctHash = correctHash.replaceAll(" ", "");
        try{
            // sigPermName を定義したアプリのパッケージ名を取得する
            PackageManager pm = ctx.getPackageManager();
            PermissionInfo pi = pm.getPermissionInfo(sigPermName, PackageManager.GET_META_DATA);
            String pkgname = pi.packageName;
            // 非 Signature Permission の場合は失敗扱い
            if (pi.protectionLevel != PermissionInfo.PROTECTION_SIGNATURE) return false;
            // pkgname の実際のハッシュ値と正解のハッシュ値を比較する
            if (Build.VERSION.SDK_INT >= 28) {
                // ★ API Level >= 28 では Package Manager の API で直接検証が可能
                return pm.hasSigningCertificate(pkgname, Utils.hex2Bytes(correctHash), CERT_INPUT_
↪SHA256);
            } else {
                // API Level < 28 の場合は PkgCert を利用し、ハッシュ値を取得して比較する
                return correctHash.equals(PkgCert.hash(ctx, pkgname));
            }
        } catch (NameNotFoundException e){
            return false;
        }
    }
}
```

PkgCert.java

```
package org.jssec.android.shared;

import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

import android.content.Context;
import android.content.pm.PackageInfo;
import android.content.pm.PackageManager;
import android.content.pm.PackageManager.NameNotFoundException;
import android.content.pm.Signature;
```

(continues on next page)

(continued from previous page)

```
public class PkgCert {

    public static boolean test(Context ctx, String pkgname, String correctHash) {
        if (correctHash == null) return false;
        correctHash = correctHash.replaceAll(" ", "");
        return correctHash.equals(hash(ctx, pkgname));
    }

    public static String hash(Context ctx, String pkgname) {
        if (pkgname == null) return null;
        try {
            PackageManager pm = ctx.getPackageManager();
            PackageInfo pkginfo = pm.getPackageInfo(pkgname, PackageManager.GET_SIGNATURES);
            if (pkginfo.signatures.length != 1) return null;    // 複数署名は扱わない
            Signature sig = pkginfo.signatures[0];
            byte[] cert = sig.toByteArray();
            byte[] sha256 = computeSha256(cert);
            return byte2hex(sha256);
        } catch (NameNotFoundException e) {
            return null;
        }
    }

    private static byte[] computeSha256(byte[] data) {
        try {
            return MessageDigest.getInstance("SHA-256").digest(data);
        } catch (NoSuchAlgorithmException e) {
            return null;
        }
    }

    private static String byte2hex(byte[] data) {
        if (data == null) return null;
        final StringBuilder hexadecimal = new StringBuilder();
        for (final byte b : data) {
            hexadecimal.append(String.format("%02X", b));
        }
        return hexadecimal.toString();
    }
}
```

★ポイント 7 ★ APK を Export するときに、利用元アプリと同じ開発者鍵で APK を署名する。

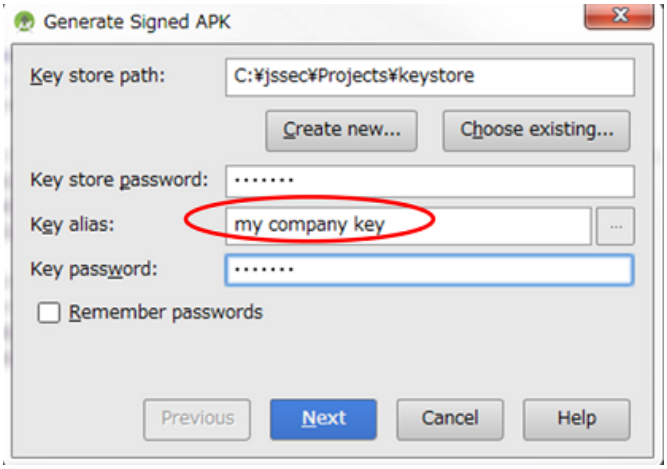

図 4.11.4　利用元アプリと同じ開発者鍵で APK を署名する

クライアント側のサンプルコード

ポイント:

8. 独自定義 Signature Permission を利用宣言する
9. 独自定義 Signature Permission が自社アプリにより定義されていることを確認する
10. 利用先アプリの証明書が自社の証明書であることを確認する
11. 利用先アプリは自社アプリであるから、センシティブな情報を送信してもよい
12. 明示的 Intent により自社限定 Service を呼び出す
13. 利用先アプリと同じ開発者鍵で APK を署名する

ここで示されているポイントは、すべて「4.4.1.4. 自社限定 *Service*」のクライアント側のポイントと同じであり、共有メモリ特有のものはない。共有メモリの利用については、下記のサンプルコードに基本的なものが示されているのでこれを参照してほしい。

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="org.jssec.android.sharedmemory.inhouseservice.messenger">

    <permission android:name="org.jssec.android.sharedmemory.inhouseservice.messenger.MY_PERMISSION"
        android:protectionLevel="signature" />
    <!-- ★ポイント 8★ 独自定義 Signature Permission を利用宣言する -->
    <uses-permission
        android:name="org.jssec.android.service.inhouseservice.messenger.MY_PERMISSION" />
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name="org.jssec.android.sharedmemory.inhouseservice.messenger.MainActivity">
            <intent-filter>
```

(continues on next page)

(continued from previous page)

```
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <!-- Messenger を利用した Service -->
        <!-- 簡単のため共有メモリを提供するサービスを同じアプリの別プロセスとする -->
        <service android:name="org.jssec.android.sharedmemory.inhouseservice.messenger.SHMService"
            android:exported="true"
            android:permission="org.jssec.android.sharedmemory.inhouseservice.messenger.MY_PERMISSION"
            android:process=".shmService" />
    </application>
</manifest>
```

MainActivity.java

```
package org.jssec.android.sharedmemory.inhouseservice.messenger;

import android.app.Activity;
import android.content.ComponentName;
import android.content.Context;
import android.content.Intent;
import android.content.ServiceConnection;
import android.os.Bundle;
import android.os.Handler;
import android.os.IBinder;
import android.os.Message;
import android.os.Messenger;
import android.os.RemoteException;
import android.os.SharedMemory;
import android.system.ErrnoException;
import android.widget.Toast;
import android.util.Log;

import org.jssec.android.shared.PkgCert;
import org.jssec.android.shared.SigPerm;
import org.jssec.android.shared.Utils;

import java.nio.ByteBuffer;
import java.nio.ReadOnlyBufferException;

import static android.system.OsConstants.PROT_EXEC;
import static android.system.OsConstants.PROT_READ;
import static android.system.OsConstants.PROT_WRITE;

public class MainActivity extends Activity {

    private final String TAG = "SHMClient";

    // Service へデータを送信するときに使用する Messenger
    private Messenger mServiceMessenger = null;

    // 共有メモリオブジェクト
    private SharedMemory myShared1;
    private SharedMemory myShared2;
```

(continues on next page)

(continued from previous page)

```
    // 共有メモリをマップする ByteBuffer
    private ByteBuffer mBuf1;
    private ByteBuffer mBuf2;

    // 利用先の Activity 情報
    private static final String SHM_PACKAGE = "org.jssec.android.sharedmemory.inhouseservice.messenger";
    private static final String SHM_CLASS = "org.jssec.android.sharedmemory.inhouseservice.messenger.
↪SHMService";


    // 自社の Signature Permission
    private static final String MY_PERMISSION = "org.jssec.android.sharedmemory.inhouseservice.messenger.
↪MY_PERMISSION";

    // 自社の証明書のハッシュ値
    private static String sMyCertHash = null;
    private static String myCertHash(Context context) {
        if (sMyCertHash == null) {
            if (Utils.isDebuggable(context)) {
                // debug.keystore の"androiddebugkey"の証明書ハッシュ値
                sMyCertHash = "0EFB7236 328348A9 89718BAD DF57F544 D5CCB4AE B9DB34BC 1E29DD26 F77C8255";
            } else {
                // keystore の"my company key"の証明書ハッシュ値
                sMyCertHash = "D397D343 A5CBC10F 4EDDEB7C A10062DE 5690984F 1FB9E88B D7B3A7C2 42E142CA";
            }
        }
        return sMyCertHash;
    }

    // Service と接続された状態で true となる
    private boolean mIsBound = false;

    // Service から受け取った Message を処理する Handler
    private class MyHandler extends Handler {
        @Override
        public void handleMessage(Message msg) {
            switch (msg.what) {
                case SHMService.SHMEM1:
                    // サービスから SHMEM 1 が提供された
                    // Message.obj に共有メモリが格納されている
                    myShared1 = (SharedMemory) msg.obj;
                    useSHMEM1();
                    break;
                case SHMService.SHMEM2:
                    // サービスから SHMEM 2 が提供された
                    myShared2 = (SharedMemory) msg.obj;
                    useSHMEM2();
                    break;
                case SHMService.MSG_END:
                    alloverNow();
                    break;
                default:
                    Log.e(TAG, "invalid message: " + msg.what);
            }
```

(continues on next page)

(continued from previous page)

```
        }
    }

    private Handler mHandler = new MyHandler();

    // Service からデータを受信するときに使用する Messenger
    private Messenger mLocalMessenger = new Messenger(mHandler);

    // Service と接続する時に利用するコネクション。bindService で実装する場合は必要になる
    private class MyServiceConnection implements ServiceConnection {
        // Service に接続された場合に呼ばれる
        public void onServiceConnected(ComponentName className, IBinder service){
            mServiceMessenger = new Messenger(service);

            // 共有メモリ Service にバインドしたタイミングで一つ目の共有メモリを要求する
            sendMessageToService(SHMService.MSG_ATTACH);
        }
        // Service が異常終了して、コネクションが切断された場合に呼ばれる
        public void onServiceDisconnected(ComponentName className){
            mIsBound = false;
            mServiceMessenger = null;
        }
    }
    private MyServiceConnection mServiceConnection;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        doBindService ();
    }

    // 共有メモリ Service に接続する
    private void doBindService () {
        mServiceConnection = new MyServiceConnection();
        if (!mIsBound) {
            // ★ポイント 9★ 独自定義 Signature Permission が自社アプリにより定義されていることを確認する
            if (!SigPerm.test(this, MY_PERMISSION, myCertHash(this))) {
                Toast.makeText(this, "独自定義 Signature Permission が自社アプリにより定義されていない。",␣
↪Toast.LENGTH_LONG).show();
                return;
            }

            // ★ポイント 10★ 利用先アプリの証明書が自社の証明書であることを確認する
            if (!PkgCert.test(this, SHM_PACKAGE, myCertHash(this))) {
                Toast.makeText(this, "利用先サービスは自社アプリではない。", Toast.LENGTH_LONG).show();
                return;
            }
        }
        Intent it = new Intent();
        // ★ポイント 11★ 利用先アプリは自社アプリであるから、センシティブな情報を送信してもよい
        it.putExtra("PARAM", "センシティブな情報");

        // ★ポイント 12★ 明示的インテントを用いて共有メモリサービスに bind する
        it.setClassName(SHM_PACKAGE, SHM_CLASS);
```

(continues on next page)

(continued from previous page)

```
        if (!bindService(it, mServiceConnection, Context.BIND_AUTO_CREATE)) {
            Toast.makeText(this, "Bind Service Failed", Toast.LENGTH_LONG).show();
            return;
        }
        mIsBound = true;
    }

    // サービスとの接続を切断する
    private void releaseService () {
        unbindService(mServiceConnection);
    }

    // SHMEM1 を利用する例
    private void useSHMEM1 () {
        // SHMEM1 は読み込みのみが許可されているので PROT_READ でマップする
        // これ以外でマップすると例外が発生する
        mBuf1 = mapMemory(SHMService.SHMEM1, PROT_READ, SHMService.SHMEM1_BUF1_OFFSET, SHMService.SHMEM1_
↪BUF1_LENGTH);
        // サービス側がセットしたデータを読み込む
        int len = mBuf1.getInt();
        byte[] bytes = new byte[len];
        mBuf1.get(bytes);
        String message = new String(bytes);
        Toast.makeText(MainActivity.this, "Got: " + message, Toast.LENGTH_LONG).show();
        // サービスへリプライ
        sendMessageToService(SHMService.MSG_REPLY1);
        // 続けて書き込み可能な共有メモリをリクエストする
        sendMessageToService(SHMService.MSG_ATTACH2);
    }

    // SHMEM2 を利用する例
    private void useSHMEM2 () {
        // SHMEM2 は書き込みが許可されているため PROT_WRITE でマッピングする
        // サービス側では PROT_WRITE を設定しているため PROT_READ | PROT_WRITE でマップすると例外が発生する
        mBuf2 = mapMemory(SHMService.SHMEM2, PROT_WRITE, SHMService.SHMEM2_BUF1_OFFSET, SHMService.
↪SHMEM2_BUF1_LENGTH);
        if (mBuf2 != null) {
            // PROT_WRITE のみでも読み込みができる場合が多い
            int size = mBuf2.getInt();
            byte [] bytes = new byte[size];
            mBuf2.get(bytes);
            String msg = new String(bytes);
            Log.d(TAG, "Got a message from service: " + msg);
            // サービス側で設定したデータを上書きする
            String replyStr = "OK Thanks!";
            mBuf2.putInt(replyStr.length());
            mBuf2.put(replyStr.getBytes());
            // サービスへリプライ
            sendMessageToService(SHMService.MSG_REPLY2);
        }
    }

    // 指定の共有メモリをマップする
    private ByteBuffer mapMemory(SharedMemory mem, int proto, int offset, int length){
```

(continues on next page)

(continued from previous page)

```
        ByteBuffer tempBuf;
        try {
            tempBuf = mem.map(proto, offset, length);
        } catch (ErrnoException e){
            Log.e(TAG, "could not map, proto: " + proto + ", offset:" + offset + ", length: " + length +
↪"\n " + e.getMessage() + "err no. = " + e.errno);
            return null;
        }
        return tempBuf;
    }

    // サービスへメッセージを送る
    private void sendMessageToService(int what){
        try {
            Message msg = Message.obtain();
            msg.what = what;
            msg.replyTo = mLocalMessenger;
            mServiceMessenger.send(msg);
        } catch (RemoteException e) {
            Log.e(TAG, "Error in sending message: " + e.getMessage());
        }
    }

    // 使い終わった共有メモリを後始末する
    public void alloverNow() {
        // サービスへ使い終わったことを通知
        sendMessageToService(SHMService.MSG_DETACH);
        // マップされた ByteBuffer を unmap
        if (mBuf1 != null) SharedMemory.unmap(mBuf1);
        if (mBuf2 != null) SharedMemory.unmap(mBuf2);
        // 共有メモリをクローズ
        myShared1.close();
        myShared2.close();
        mBuf1 = null;
        mBuf2 = null;
        myShared1 = null;
        myShared2 = null;
        // サービスとのコネクションを切る
        releaseService();
    }
}
```

★ポイント 13 ★ APK を Export するときに、利用先アプリと同じ開発者鍵で APK を署名する。

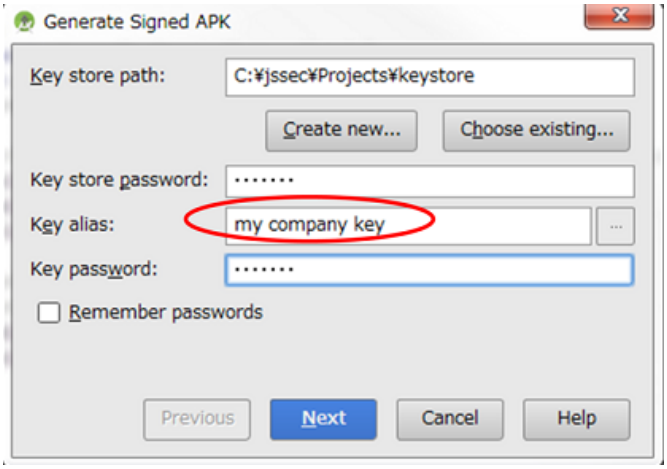

図 4.11.5　利用先アプリと同じ開発者鍵で APK を署名する

4.11.3 ルールブック

SharedMemory を利用する際には Service についてのルールブック (4.4.2. ルールブック) に記載のルールを守ること。それらに加えて、以下のルールを守ること。

1. 共有メモリを提供する側は利用側へ許可するアクセス権を適切に設定する *(必須)*
2. 共有メモリ上のデータはすべて共有するアプリから読まれることを前提に設計する *(必須)*

4.11.3.1 共有メモリを提供する側は利用側へ許可するアクセス権を適切に設定する (必須)

メモリを共有する場合、各アプリがメモリに対して可能な操作を設計上必要最低限の操作に制限し、情報の漏洩や改ざん・破壊を防止すべきである。SharedMemory オブジェクトを作成する Service は他のアプリと共有する前に SharedMemory#setProtect() を使用して共有メモリ全体に対して可能な操作を制限できる。SharedMemory オブジェクトに対して可能な操作の初期値は、読み・書き・実行である。このうち実行可能なメモリ領域は特別な理由がない限り、不正なコードの実行を未然に防ぐため使用すべきではない[*31]。また、他のアプリが共有メモリに書き込む必要のある場合は、別途専用の共有メモリを作成して提供することでより安全にメモリを共有することができる。

SharedMemory#setProtect() の引数は読み・書き・実行のそれぞれに対応したビットフラグ (PROT_READ、PROT_WRITE、PROT_EXEC) の論理的 OR である。下は SharedMemory オブジェクト shMem に対して読み込みと書き込みのみを許可している例である。

```
shMem.setProtect(PROT_READ | PROT_WRITE)
```

クライアント側でこの共有メモリ内の (一部または全体の) 領域をアクセスできるようにするには SharedMemory#map() を事前に実行する必要がある。その際引数の一つとしてメモリに対して行う操作を指定するが、事前に Service 側が SharedMemory#setProtect() で許可した以上の操作を指定することはできない。例えば Service 側が読み込みのみを許可しているにもかかわらず、クライアント側で書き込み操作を指定することはできない。下の例は Service から提供された SharedMemory オブジェクト ashMem を map() している例である。

[*31] デバイス (使用している CPU アーキテクチャ) によっては、あるメモリ領域が読み込み可能であった場合、自動的に実行可能となるものがある。しかしその場合でも、その領域に対して書き込みを禁止することにより、他のアプリがその領域に実行可能なコードを書き込むのを防止することが可能である。

```
ByteBuffer mbuf;
// Service が読み込みのみを許可している場合、下のコードは例外を引き起こす
mbuf = ashMem.map(offset, length, PROT_WRITE);
```

クライアント側でも setProtect() を呼び出して共有メモリ全体に許可される操作を設定し直すことは可能だが、map() と同様に先に Service によって許可された操作以上の操作を許可するように設定することはできない。

4.11.3.2 共有メモリ上のデータはすべて共有するアプリから読まれることを前提に設計する (必須)

上述のように他のアプリとメモリを共有する場合、提供する側で事前に共有メモリに対するアクセス権 (読み・書き・実行) を設定できる。しかしフラグを PROT_WRITE のみにして書き込みだけを許可しても、メモリの読み込みを禁止することができない場合がある。言い換えると、デバイスが使用しているメモリ管理ユニット (MMU) が書き込みのみを許すメモリアクセスをサポートしていない場合は、あるメモリ領域が書き込み可能であると読み込みも可能となるということである。実際にこのようなデバイスは多いと考えられ、そのため共有メモリの内容は他のアプリに知られることを前提とした設計をしなければならない。

```
// Service 側で SharedMemory#setProtect(PROT_WRITE) で書き込みのみを許可しているものとする
// クライアント側で map により PROT_WRITE でマップしても、読み込みが可能である場合が多い
ByteBuffer buf;
buf = ashMem.map(offset, length, PROT_WRITE);
// その場合読み込み操作はエラーとならない
int len = buf.getInt();
byte [] bytes = new byte[len];
buf.get(bytes);
```

フラグに PROT_NONE を指定し、いかなる操作もできないようにすることは可能であるが、その時点で共有の意味は失われる。

4.11.4 アドバンスト

4.11.4.1 共有メモリの実際

これまでの説明では共有メモリを複数のアプリ間でメモリを共有する機構として述べてきた。しかし、実際のところ共有メモリは同一の物理メモリ領域を複数のプロセス間で共有する機構である。各々のプロセスは共有する物理メモリ領域を自身のアドレス空間にマップしてアクセスする (SharedMemory#map() はこれを行なっている)。Android の共有メモリの場合、マップされたメモリ領域は (Java 言語の場合) 一つの ByteBuffer オブジェクトである (ページサイズを超えた共有メモリをアロケートした場合、共有される物理メモリ領域は連続した領域ではなく複数の非連続なページに分割されているのが一般的であるが、プロセスのアドレス空間上にマップした場合は連続したアドレス空間となる。)

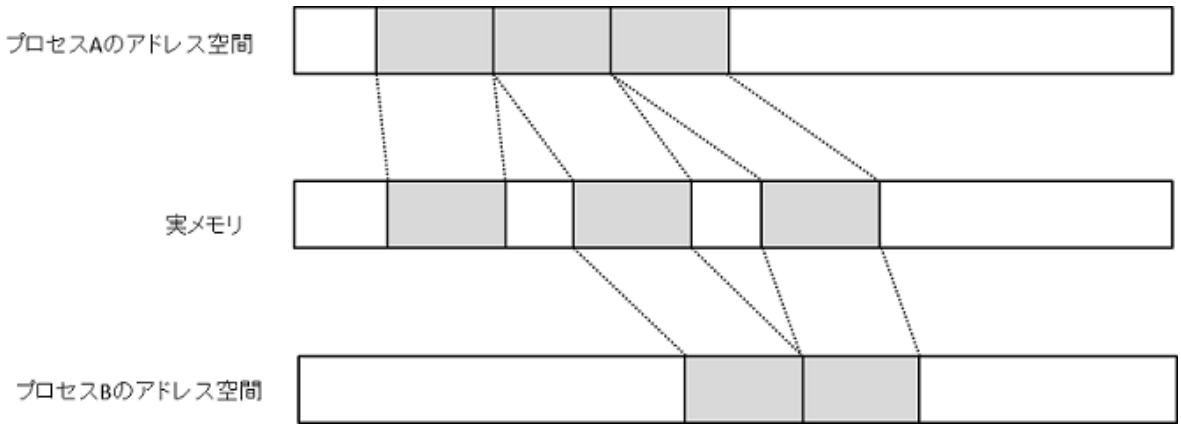

図 4.11.6　実メモリとプロセスのアドレス空間

Android OS を含む Unix 系の OS では、接続されているターミナルや USB デバイスなどの周辺機器をデバイスファイルという概念で抽象化し、仮想的なファイルとして取り扱う。Android OS の共有メモリもその例外ではなく、/dev/ashmem というデバイスファイルがこれに相当する。このデバイスファイルをオープンすると、通常のファイルをオープンした場合と同様にファイルディスクリプタが返され、これを通じて共有メモリへのアクセスが行われる。このファイルディスクリプタは通常のファイルの場合とおなじく mmap() によってプロセスのアドレス空間にマップすることができる。mmap() は Unix 系 OS の標準的なシステムコールであり、様々なデバイスに対応するデバイスファイルのファイルディスクリプタを取り、それを呼び出しプロセスのアドレス空間にマップする機能を提供している。Android OS の共有メモリでもこれを利用している。マップされたアドレス空間はプログラムからはバイト列として見える (Java の場合は上述のように ByteBuffer、C 言語レベルでは char * である)。

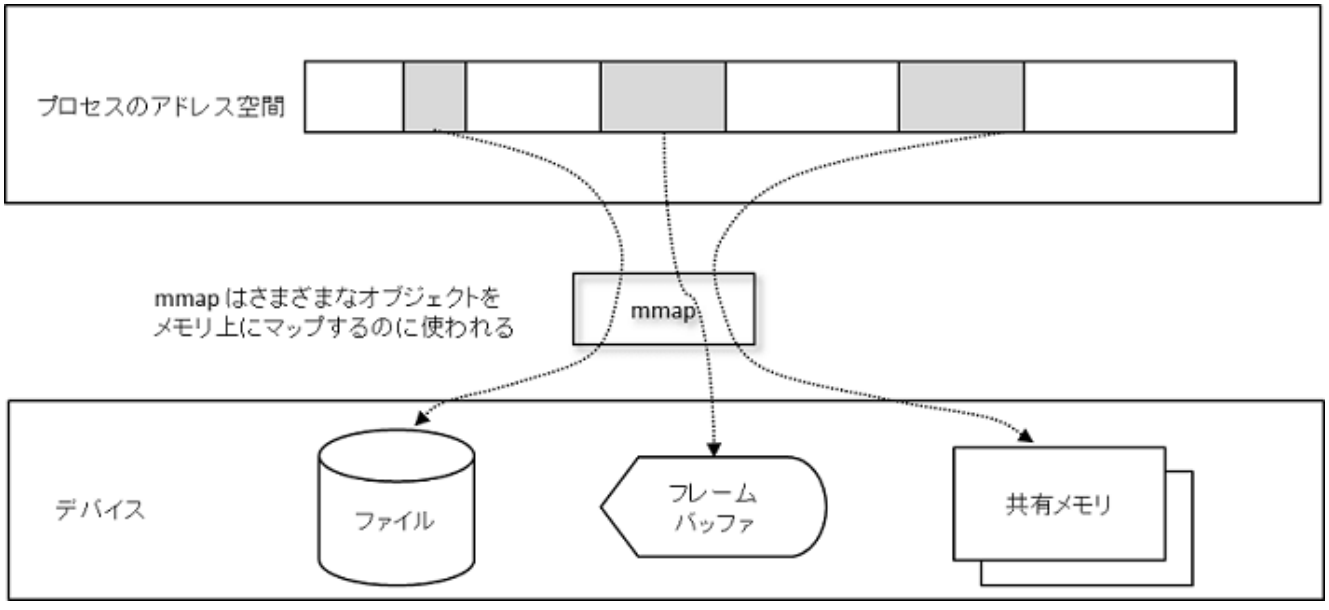

図 4.11.7　仮想ファイルとアドレス空間のマッピング

この枠組みでプロセス間でメモリを共有するとは、そのメモリ領域に対応する /dev/asmem のファイルディスクリプタを共有することに等しい[*32] 。そのため共有にかかるコストは小さくて済み、一旦プロセスのアドレス空間上にマップされた後は通常のメモリアクセスと等価な効率でのアクセスが可能となる。

[*32] ファイルディスクリプタはプロセス内で固有の値であるため、他のプロセスにそれを渡す場合は適切な変換が必要となるが、Android SDK API レベルではこれを気にする必要はない。

5 セキュリティ機能の使い方

暗号や電子署名、Permission など、Android にはさまざまなセキュリティ機能が用意されている。これらのセキュリティ機能は取り扱いを間違えるとセキュリティ機能が十分に発揮されず抜け道ができてしまう。この章では開発者がセキュリティ機能を活用するシーンを想定した記事を扱う。

5.1 パスワード入力画面を作る

5.1.1 サンプルコード

パスワード入力画面を作る際、セキュリティ上考慮すべきポイントについて述べる。ここではパスワードの入力に関する内容のみとする。パスワードの保存方法については今後の版にて別途記事を設ける予定である。

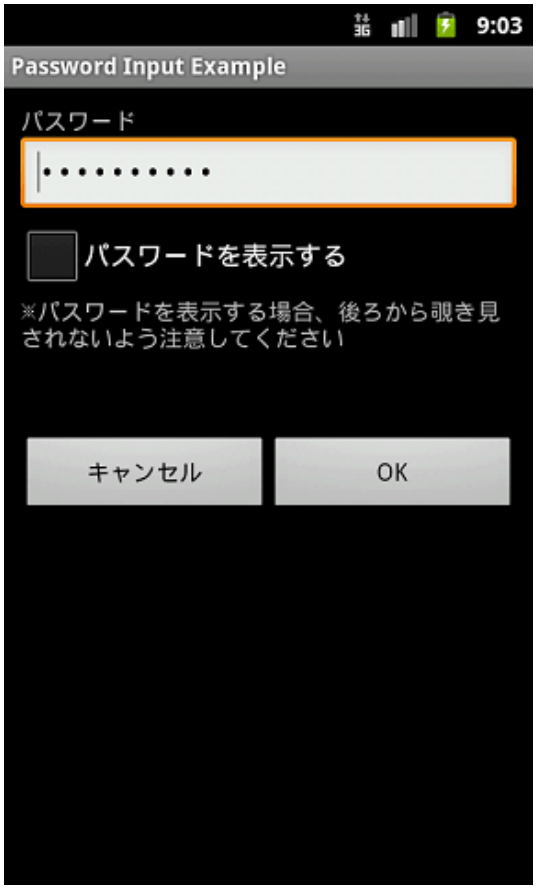

図 5.1.1　パスワード入力画面例

ポイント：

1. 入力したパスワードはマスク表示（●で表示）する
2. パスワードを平文表示するオプションを用意する
3. パスワード平文表示時の危険性を注意喚起する

ポイント：　前回入力したパスワードを扱う場合には上記ポイントに加え、下記ポイントにも気を付けること

4. Activity 初期表示時に前回入力したパスワードがある場合、前回入力パスワードの桁数を推測されないよう固定桁数の●文字でダミー表示する
5. 前回入力パスワードをダミー表示しているとき、「パスワードを表示」した場合、前回入力パスワードをクリアして、新規にパスワードを入力できる状態とする
6. 前回入力パスワードをダミー表示しているとき、ユーザーがパスワードを入力しようとした場合、前回入力パスワードをクリアし、ユーザーの入力を新たなパスワードとして扱う

```
password_activity.xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical"
    android:padding="10dp" >
```

(continues on next page)

(continued from previous page)

```
    <!-- パスワード項目のラベル -->
    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/password" />

    <!-- パスワード入力項目 -->
    <!-- ★ポイント 1★ 入力したパスワードはマスク表示（●で表示）する -->
    <EditText
        android:id="@+id/password_edit"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:hint="@string/hint_password"
        android:inputType="textPassword" />

    <!-- ★ポイント 2★ パスワードを平文表示するオプションを用意する -->
    <CheckBox
        android:id="@+id/password_display_check"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/display_password" />

    <!-- ★ポイント 3★ パスワード平文表示時の危険性を注意喚起する -->
    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/alert_password" />

    <!-- キャンセル、OK ボタン -->
    <LinearLayout
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_marginTop="50dp"
        android:gravity="center"
        android:orientation="horizontal" >

        <Button
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:onClick="onClickCancelButton"
            android:text="@android:string/cancel" />

        <Button
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:onClick="onClickOkButton"
            android:text="@android:string/ok" />
    </LinearLayout>
</LinearLayout>
```

次の PasswordActivity.java の最後に配置した 3 つのメソッドは用途に合わせて実装内容を調整すること。

- private String getPreviousPassword()

- private void onClickCancelButton(View view)
- private void onClickOkButton(View view)

PasswordActivity.java

```
package org.jssec.android.password.passwordinputui;

import android.app.Activity;
import android.os.Bundle;
import android.text.Editable;
import android.text.InputType;
import android.text.TextWatcher;
import android.view.View;
import android.view.WindowManager;
import android.widget.CheckBox;
import android.widget.CompoundButton;
import android.widget.CompoundButton.OnCheckedChangeListener;

import android.widget.EditText;
import android.widget.Toast;

public class PasswordActivity extends Activity {

    // 状態保存用のキー
    private static final String KEY_DUMMY_PASSWORD = "KEY_DUMMY_PASSWORD";

    // Activity 内の View
    private EditText mPasswordEdit;
    private CheckBox mPasswordDisplayCheck;

    // パスワードがダミー表示かを表すフラグ
    private boolean mIsDummyPassword;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.password_activity);

        // スクリーンキャプチャを無効化する
        getWindow().addFlags(WindowManager.LayoutParams.FLAG_SECURE);

        // View の取得
        mPasswordEdit = (EditText) findViewById(R.id.password_edit);
        mPasswordDisplayCheck = (CheckBox) findViewById(R.id.password_display_check);

        // 前回入力パスワードがあるか
        if (getPreviousPassword() != null) {
            // ★ポイント 4★ Activity 初期表示時に前回入力したパスワードがある場合、
            // 前回入力パスワードの桁数を推測されないよう固定桁数の●文字でダミー表示する

            // 表示はダミーパスワードにする
            mPasswordEdit.setText("**********");
            // パスワード入力時にダミーパスワードをクリアするため、テキスト変更リスナーを設定
            mPasswordEdit.addTextChangedListener(new PasswordEditTextWatcher());
            // ダミーパスワードフラグを設定する
            mIsDummyPassword = true;
```

(continues on next page)

(continued from previous page)

```
        }

        // パスワードを表示するオプションのチェック変更リスナーを設定
        mPasswordDisplayCheck
                .setOnCheckedChangeListener(new OnPasswordDisplayCheckedChangeListener());
    }

    @Override
    public void onSaveInstanceState(Bundle outState) {
        super.onSaveInstanceState(outState);

        // 画面の縦横変更で Activity が再生成されないよう指定した場合には不要
        // Activity の状態保存
        outState.putBoolean(KEY_DUMMY_PASSWORD, mIsDummyPassword);
    }

    @Override
    public void onRestoreInstanceState(Bundle savedInstanceState) {
        super.onRestoreInstanceState(savedInstanceState);

        // 画面の縦横変更で Activity が再生成されないよう指定した場合には不要
        // Activity の状態の復元
        mIsDummyPassword = savedInstanceState.getBoolean(KEY_DUMMY_PASSWORD);
    }

    /**
     * パスワードを入力した場合の処理
     */
    private class PasswordEditTextWatcher implements TextWatcher {

        public void beforeTextChanged(CharSequence s, int start, int count,
                int after) {
            // 未使用
        }

        public void onTextChanged(CharSequence s, int start, int before,
                int count) {
            // ★ポイント 6★ 前回入力パスワードをダミー表示しているとき、ユーザーがパスワードを入力しようと
            // した場合、前回入力パスワードをクリアし、ユーザーの入力を新たなパスワードとして扱う
            if (mIsDummyPassword) {
                // ダミーパスワードフラグを設定する
                mIsDummyPassword = false;
                // パスワードを入力した文字だけにする
                CharSequence work = s.subSequence(start, start + count);
                mPasswordEdit.setText(work);
                // カーソル位置が最初に戻るので最後にする
                mPasswordEdit.setSelection(work.length());

            }
        }

        public void afterTextChanged(Editable s) {
            // 未使用
        }
```

(continues on next page)

(continued from previous page)

```
    }

    /**
     * パスワードの表示オプションチェックを変更した場合の処理
     */
    private class OnPasswordDisplayCheckedChangeListener implements
            OnCheckedChangeListener {

        public void onCheckedChanged(CompoundButton buttonView,
                boolean isChecked) {
            // ★ポイント 5★ 前回入力パスワードをダミー表示しているとき、「パスワードを表示」した場合、
            // 前回入力パスワードをクリアして、新規にパスワードを入力できる状態とする
            if (mIsDummyPassword && isChecked) {
                // ダミーパスワードフラグを設定する
                mIsDummyPassword = false;
                // パスワードを空表示にする
                mPasswordEdit.setText(null);
            }

            // カーソル位置が最初に戻るので今のカーソル位置を記憶する
            int pos = mPasswordEdit.getSelectionStart();

            // ★ポイント 2★ パスワードを平文表示するオプションを用意する
            // InputType の作成
            int type = InputType.TYPE_CLASS_TEXT;
            if (isChecked) {
                // チェック ON 時は平文表示
                type |= InputType.TYPE_TEXT_VARIATION_VISIBLE_PASSWORD;
            } else {
                // チェック OFF 時はマスク表示
                type |= InputType.TYPE_TEXT_VARIATION_PASSWORD;
            }

            // パスワード EditText に InputType を設定
            mPasswordEdit.setInputType(type);

            // カーソル位置を設定する
            mPasswordEdit.setSelection(pos);
        }

    }

    // 以下のメソッドはアプリに合わせて実装すること

    /**
     * 前回入力パスワードを取得する
     *
     * @return 前回入力パスワード
     */
    private String getPreviousPassword() {
        // 保存パスワードを復帰させたい場合にパスワード文字列を返す
        // パスワードを保存しない用途では null を返す
        return "hirake5ma";
    }
```

(continues on next page)

(continued from previous page)

```
    /**
     * キャンセルボタンの押下処理
     *
     * @param view
     */
    public void onClickCancelButton(View view) {
        // Activity を閉じる
        finish();
    }

    /**
     * OK ボタンの押下処理
     *
     * @param view
     */
    public void onClickOkButton(View view) {
        // password を保存するとか認証に使うとか必要な処理を行う

        String password = null;

        if (mIsDummyPassword) {
            // 最後までダミーパスワード表示だった場合は前回入力パスワードを確定パスワードとする
            password = getPreviousPassword();

        } else {
            // ダミーパスワード表示じゃない場合はユーザー入力パスワードを確定パスワードとする
            password = mPasswordEdit.getText().toString();
        }

        // パスワードを Toast 表示する
        Toast.makeText(this, "password is \"" + password + "\"",
                Toast.LENGTH_SHORT).show();

        // Activity を閉じる
        finish();
    }
}
```

5.1.2 ルールブック

パスワード入力画面を作る際には以下のルールを守ること。

1. パスワードを入力するときにはマスク表示（●で表示する）機能を用意する （必須）
2. パスワードを平文表示するオプションを用意する （必須）
3. *Activity* 起動時はパスワードをマスク表示にする （必須）
4. 前回入力したパスワードを表示する場合、ダミーパスワードを表示する （必須）

5.1.2.1 パスワードを入力するときにはマスク表示（●で表示する）機能を用意する （必須）

スマートフォンは電車やバス等の人混みで利用されることが多く、第三者にパスワードを盗み見られるリスクが大きい。アプリの仕様として、パスワードをマスク表示する機能が必要である。

パスワードを入力する EditText をマスク表示する方法には、静的にレイアウト XML で指定する方法と、動的にプログラム上で切り替える方法の 2 種類がある。前者は、android:inputType 属性に”textPassword” を指定することで実現でき、また android:password 属性でも実現できる。後者は、EditText クラスの setInputType() メソッドで EditText の入力タイプに InputType.TYPE_TEXT_VARIATION_PASSWORD を追加することで実装できる。

以下、それぞれのサンプルコードを示す。

レイアウト XML で指定する方法

```
password_activity.xml
    <!-- パスワード入力項目 -->
    <!-- android:password を true に設定する -->
    <EditText
        android:id="@+id/password_edit"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:hint="@string/hint_password"
        android:inputType="textPassword" />
```

Activity 内で指定する方法

```
PasswordActivity.java
    // パスワード表示タイプを設定
    // InputType に TYPE_TEXT_VARIATION_PASSWORD を設定する
    EditText passwordEdit = (EditText) findViewById(R.id.password_edit);
    int type = InputType.TYPE_CLASS_TEXT
            | InputType.TYPE_TEXT_VARIATION_PASSWORD;
    passwordEdit.setInputType(type);
```

5.1.2.2 パスワードを平文表示するオプションを用意する （必須）

スマートフォンにおけるパスワード入力はタッチパネルでの入力となるため、PC でキーボード入力する場合と比較すると誤入力が生じやすい。その入力の煩わしさからユーザーは単純なパスワードを利用してしまう可能性があり、かえって危険である。また複数回のパスワード入力失敗によりアカウントをロックするなどのポリシーがある場合、誤入力はできるだけ避けるようにする必要もある。それらの解決策として、パスワードを平文表示できるオプションを用意することで、安全なパスワードを利用してもらえるようになる。

ただし、パスワードを平文表示した際に覗き見される可能性もあるため、そのオプションを使う際に、ユーザーに背後からの覗き見への注意を促す必要がある。また、平文表示するオプションをつけた場合、平文表示の時間を設定するなど平文表示の自動解除を行う仕組みも用意する必要がある。パスワードの平文表示の制限については今後の版にて別途記事を設ける予定である。そのため、この版のサンプルコードにはパスワードの平文表示の制限は含めていない。

図 5.1.2　パスワードの平文表示

EditText の InputType 指定で、マスク表示と平文表示を切り替えることができる

PasswordActivity.java

```
    /**
     * パスワードの表示オプションチェックを変更した場合の処理
     */
    private class OnPasswordDisplayCheckedChangeListener implements
            OnCheckedChangeListener {

        public void onCheckedChanged(CompoundButton buttonView,
                boolean isChecked) {
            // ★ポイント 5★ ダミー表示時は空表示にする
            if (mIsDummyPassword && isChecked) {
                // ダミーパスワードフラグを設定する
                mIsDummyPassword = false;
                // パスワードを空表示にする
                mPasswordEdit.setText(null);
            }

            // カーソル位置が最初に戻るので今のカーソル位置を記憶する
            int pos = mPasswordEdit.getSelectionStart();

            // ★ポイント 2★ チェックに応じてパスワードを平文表示する
            // InputType の作成
            int type = InputType.TYPE_CLASS_TEXT;
            if (isChecked) {
                // チェック ON 時は平文表示
                type |= InputType.TYPE_TEXT_VARIATION_VISIBLE_PASSWORD;
            } else {
                // チェック OFF 時はマスク表示
                type |= InputType.TYPE_TEXT_VARIATION_PASSWORD;
            }

            // パスワード EditText に InputType を設定
            mPasswordEdit.setInputType(type);

            // カーソル位置を設定する
```

(continues on next page)

(continued from previous page)

```
            mPasswordEdit.setSelection(pos);
        }

    }
```

5.1.2.3 Activity 起動時はパスワードをマスク表示にする （必須）

意図せずパスワード表示してしまい、第三者に見られることを防ぐため、Activity 起動時にパスワードを表示するオプションのデフォルト値はオフにするべきである。デフォルト値は安全側に定めるのが基本である。

5.1.2.4 前回入力したパスワードを表示する場合、ダミーパスワードを表示する （必須）

前回入力したパスワードを指定する場合、第三者にパスワードのヒントを与えないように、固定文字数のマスク文字（●など）でダミー表示するべきである。また、ダミー表示時に「パスワードを表示する」とした場合は、パスワードをクリアしてから平文表示モードにする。これにより、スマートフォンが盗難される等によって第三者の手に渡ったとしても前回入力したパスワードが盗み見られる危険性を低く抑えることができる。なお、ダミー表示時にユーザーがパスワードを入力しようとした場合には、ダミー表示を解除して通常の入力状態に戻す必要がある。

前回入力したパスワードを表示する場合、ダミーパスワードを表示する

PasswordActivity.java

```
    @Override
    public void onCreate(Bundle savedInstanceState) {

        // ～省略～

        // 前回入力パスワードがあるか
        if (getPreviousPassword() != null) {
            // ★ポイント 4★ 前回入力パスワードがある場合はダミーパスワードを表示する

            // 表示はダミーパスワードにする
            mPasswordEdit.setText("**********");
            // パスワード入力時にダミーパスワードをクリアするため、テキスト変更リスナーを設定
            mPasswordEdit.addTextChangedListener(new PasswordEditTextWatcher());
            // ダミーパスワードフラグを設定する
            mIsDummyPassword = true;
        }
        // ～省略～
    }

    /**
     * 前回入力パスワードを取得する
     *
     * @return 前回入力パスワード
     */
    private String getPreviousPassword() {
        // 保存パスワードを復帰させたい場合にパスワード文字列を返す
        // パスワードを保存しない用途では null を返す
        return "hirake5ma";
    }
```

ダミー表示時は、パスワードを表示するオプションをオンにすると表示内容をクリアする

PasswordActivity.java

```
    /**
     * パスワードの表示オプションチェックを変更した場合の処理
     */
    private class OnPasswordDisplayCheckedChangeListener implements
            OnCheckedChangeListener {

        public void onCheckedChanged(CompoundButton buttonView,
                boolean isChecked) {
            // ★ポイント 5★ ダミー表示時は空表示にする
            if (mIsDummyPassword && isChecked) {
                // ダミーパスワードフラグを設定する
                mIsDummyPassword = false;
                // パスワードを空表示にする
                mPasswordEdit.setText(null);
            }
            // ～省略～
        }
    }
```

ダミー表示時にユーザーがパスワードを入力した場合には、ダミー表示を解除する

PasswordActivity.java

```
    // 状態保存用のキー
    private static final String KEY_DUMMY_PASSWORD = "KEY_DUMMY_PASSWORD";

    // ～省略～

    // パスワードがダミー表示かを表すフラグ
    private boolean mIsDummyPassword;

    @Override
    public void onCreate(Bundle savedInstanceState) {

        // ～省略～

        // 前回入力パスワードがあるか
        if (getPreviousPassword() != null) {
            // ★ポイント 4★ 前回入力パスワードがある場合はダミーパスワードを表示する

            // 表示はダミーパスワードにする
            mPasswordEdit.setText("**********");
            // パスワード入力時にダミーパスワードをクリアするため、テキスト変更リスナーを設定
            mPasswordEdit.addTextChangedListener(new PasswordEditTextWatcher());
            // ダミーパスワードフラグを設定する
            mIsDummyPassword = true;
        }
        // ～省略～
    }

    @Override
    public void onSaveInstanceState(Bundle outState) {
        super.onSaveInstanceState(outState);

        // 画面の縦横変更で Activity が再生成されないよう指定した場合には不要
```

(continues on next page)

(continued from previous page)

```
        // Activity の状態保存
        outState.putBoolean(KEY_DUMMY_PASSWORD, mIsDummyPassword);
    }

    @Override
    public void onRestoreInstanceState(Bundle savedInstanceState) {
        super.onRestoreInstanceState(savedInstanceState);

        // 画面の縦横変更で Activity が再生成されないよう指定した場合には不要
        // Activity の状態の復元
        mIsDummyPassword = savedInstanceState.getBoolean(KEY_DUMMY_PASSWORD);
    }

    /**
     * パスワードを入力した場合の処理
     */
    private class PasswordEditTextWatcher implements TextWatcher {

        public void beforeTextChanged(CharSequence s, int start, int count,
                int after) {
            // 未使用
        }

        public void onTextChanged(CharSequence s, int start, int before,
                int count) {
            // ★ポイント 6★ ダミー表示時にパスワードを再入力した場合は入力内容に応じた表示にする
            if (mIsDummyPassword) {
                // ダミーパスワードフラグを設定する
                mIsDummyPassword = false;
                // パスワードを入力した文字だけにする
                CharSequence work = s.subSequence(start, start + count);
                mPasswordEdit.setText(work);
                // カーソル位置が最初に戻るので最後にする
                mPasswordEdit.setSelection(work.length());
            }
        }

        public void afterTextChanged(Editable s) {
            // 未使用
        }
    }
```

5.1.3 アドバンスト

5.1.3.1 ログイン処理について

パスワード入力が求められる場面の代表例はログイン処理である。ログイン処理で気を付けるポイントをいくつか紹介する。

ログイン失敗時のエラーメッセージ

ログイン処理では ID（アカウント）とパスワードの 2 つの情報を入力する。ログイン失敗時には ID が存在しない場合と、ID は存在するがパスワードが間違っている場合の 2 つがある。ログイン失敗のメッセージでこの 2 つの場合を区別して表示すると、攻撃者は「指定した ID が存在するか否か」を推測できてしまう。このような推測を許さないためにも、ログイン失敗時のメッセージでは、上記 2 つの場合を区別せずに下記のように表示すべきである。

メッセージ例： ログイン ID または パスワード が間違っています。

自動ログイン機能

一度、ログイン処理が成功すると次回以降はログイン ID とパスワードの入力を省略して、自動的にログインを行う機能がある。自動ログイン機能は煩わしい入力が省略できるので利便性が高まるが、その反面スマートフォンが盗難された場合に第三者に悪用されるリスクが伴う。

第三者に悪用された場合の被害が受け入れられる用途か、十分なセキュリティ対策が可能な場合にのみ、自動ログイン機能は利用することができる。例えば、オンラインバンキングアプリの場合、第三者に端末を操作されると金銭的な被害が出るので自動ログイン機能に合わせてセキュリティ対策が必須となる。対策としては、「決済処理などの金銭的な処理が発生する直前にはパスワードの再入力を求める」、「自動ログイン設定時にユーザーに対して十分に注意を喚起し、確実な端末のロックを促す」などいくつか考えられる。自動ログインの利用にあたっては、これらの対策を前提に利便性とリスクを勘案して、慎重な検討を行うべきである。

5.1.3.2 パスワード変更について

一度設定したパスワードを別のパスワードに変更する場合、以下の入力項目を画面上に用意すべきである。

- 現在のパスワード
- 新しいパスワード
- 新しいパスワード（入力確認用）

自動ログイン機能がついている場合、第三者がアプリを利用できる可能性がある。その場合、勝手にパスワードを変更されないよう、現在のパスワードの入力を求める必要がある。また、新しいパスワードが入力ミスで使用不能に陥る危険を減らすため、新しいパスワードは 2 回、入力を求める必要がある。

5.1.3.3 システムの「パスワードを表示」設定メニューについて

Android の設定メニューの中に「パスワードを表示」という設定がある。

設定 > セキュリティ > パスワードを表示

Android 5.0 より前のバージョンまで、この手順で設定できる。ただし Android 5.0 以降では、「パスワードを表示」はチェックボックスからトグルボタンに変更されている。

図 5.1.3　設定 ー パスワードを表示

「パスワードを表示」設定をオンにすると最後に入力した 1 文字が平文表示となる。一定時間（2 秒程度）経過後、または次の文字が入力されると平文表示されていた文字はマスク表示される。オフにすると、入力直後からマスク表示となる。これはシステム全体に影響する設定であり、EditText のパスワード表示機能を使用しているすべてのアプリに適用される。

図 5.1.4　パスワード入力時の文字の表示

5.1.3.4 スクリーンキャプチャの無効化

パスワード入力画面ではパスワードなどの個人情報が画面上に表示される可能性がある。そのような画面では第三者によってスクリーンキャプチャから個人情報が流出してしまう恐れがある。よってパスワード入力画面などの個人情報が表示されてしまう恐れのある画面ではスクリーンキャプチャを無効にしておく必要がある。スクリーンキャプチャの無効化は WindowManager に addFlag で FLAG_SECURE を設定することで実現できる。

5.2 Permission と Protection Level

Permission の Protection Level には normal, dangerous, signature, signatureOrSystem の 4 種類がある。その他に「development」「system」「appop」も存在するが、一般的なアプリでは使用しないので本章での説明は省略する。Permission はどの Protection Level であるかによってそれぞれ、Normal Permission, Dangerous Permission, Signature Permission, SignatureOrSystem Permission と呼ばれる。以下、このような名称を使う。

5.2.1 サンプルコード

5.2.1.1 Android OS 既定の Permission を利用する方法

Android OS は電話帳や GPS などのユーザー資産をマルウェアから保護するための Permission というセキュリティの仕組みがある。Android OS が保護対象としている、こうした情報や機能にアクセスするアプリは、明示的にそれらにアクセスするための権限（Permission）を利用宣言しなければならない。ユーザー確認が必要な Permission では、その Permission を利用宣言したアプリがインストールされるときに次のようなユーザー確認画面が表示される[*1]。

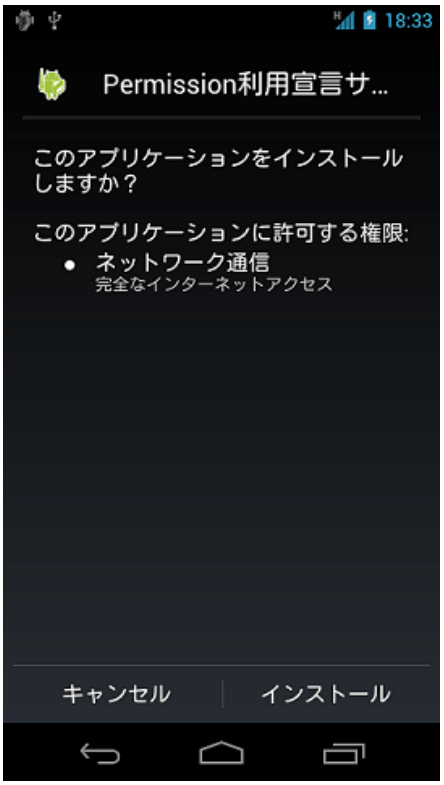

図 5.2.1　Permission 利用宣言確認画面

この確認画面により、ユーザーはそのアプリがどのような機能や情報にアクセスしようとしているのかを知ることができる。もし、アプリの動作に明らかに不必要な機能や情報にアクセスしようとしている場合は、そのアプリは悪意があるアプリの可能性が高い。ゆえに自分のアプリがマルウェアであると疑われないためにも、利用宣言する Permission は最小限にしなければならない。

ポイント：

[*1] Android 6.0(API Level 23) 以降では、ユーザー確認と権限の付与はインストール時に行われず、アプリの実行中に権限の利用を要求する仕様に変更された。詳細は「5.2.1.4. *Android 6.0 以降で Dangerous Permission を利用する方法*」および「5.2.3.6. *Android 6.0 以降の Permission モデルの仕様変更について*」を参照すること。

1. 利用する Permission を AndroidManifest.xml に uses-permission で利用宣言する
2. 不必要な Permission は利用宣言しない

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="org.jssec.android.permission.usespermission">

    <!-- ★ポイント 1★　アプリで利用する Permission を利用宣言する -->
    <!-- インターネットにアクセスする Permission -->
    <uses-permission android:name="android.permission.INTERNET"/>

    <!-- ★ポイント 2★　不必要な Permission は利用宣言しない -->
    <!-- アプリの動作に不必要な Permission を利用宣言していると、ユーザーに不信感を与えてしまう -->

    <application
        android:allowBackup="false"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name" >
        <activity
            android:name=".MainActivity"
            android:label="@string/app_name"
            android:exported="true" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

5.2.1.2 独自定義の Signature Permission で自社アプリ連携する方法

Android OS が定義する既定の Permission の他に、アプリが独自に Permission を定義することができる。独自定義の Permission（正確には独自定義の Signature Permission）を使えば、自社アプリだけが連携できる仕組みを作ることができる。複数の自社製アプリをインストールした場合に、それぞれのアプリの単機能に加え、アプリ間連携による複合機能を提供することで、複数の自社製アプリをシリーズ販売して収益を上げる、といった用途がある。

サンプルプログラム「独自定義 Signature Permission（UserApp）」はサンプルプログラム「独自定義 Signature Permission（ProtectedApp）」に startActivity() する。両アプリは同じ開発者鍵で署名されている必要がある。もし署名した開発者鍵が異なる場合は、UserApp は Intent を送信せず、ProtectedApp は受信した Intent を処理しない。またアドバンストセクションで説明しているインストール順序による Signature Permission 回避の問題にも対処している。

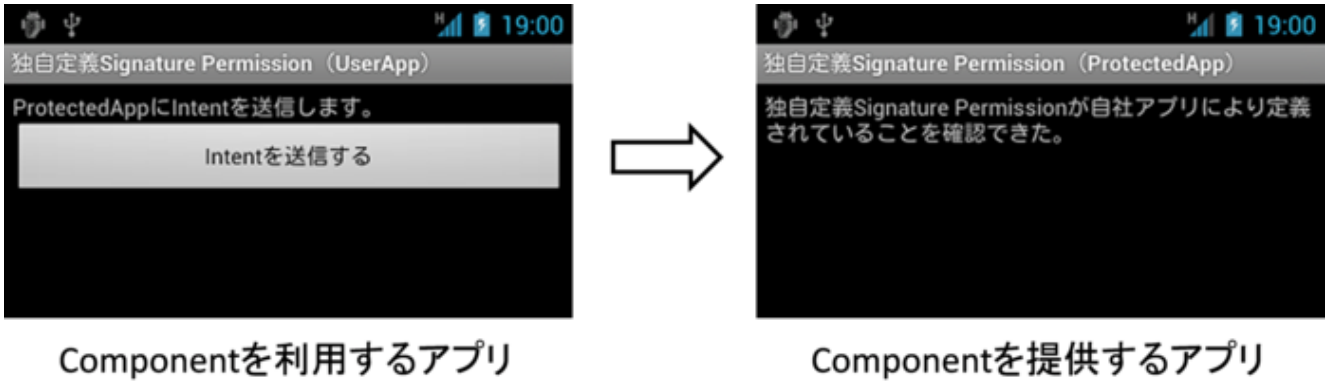

図 5.2.2　独自定義の Signature Permission で自社アプリを連携する

ポイント：Component を提供するアプリ

1. 独自 Permission を protectionLevel=”signature” で定義する
2. Component には permission 属性で独自 Permission 名を指定する
3. Component が Activity の場合には intent-filter を定義しない
4. ソースコード上で、独自定義 Signature Permission が自社アプリにより定義されていることを確認する
5. Component を利用するアプリと同じ開発者鍵で APK を署名する

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="org.jssec.android.permission.protectedapp">

    <!-- ★ポイント 1★ 独自 Permission を protectionLevel="signature"で定義する -->
    <permission
        android:name="org.jssec.android.permission.protectedapp.MY_PERMISSION"
        android:protectionLevel="signature" />

    <application
        android:allowBackup="false"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name" >

        <!-- ★ポイント 2★ Component には permission 属性で独自 Permission 名を指定する -->
        <activity
            android:name=".ProtectedActivity"
            android:exported="true"
            android:label="@string/app_name"
            android:permission="org.jssec.android.permission.protectedapp.MY_PERMISSION" >

            <!-- ★ポイント 3★ Component が Activity の場合には intent-filter を定義しない -->
        </activity>
    </application>
</manifest>
```

ProtectedActivity.java

```
package org.jssec.android.permission.protectedapp;
```

(continues on next page)

(continued from previous page)

```
import org.jssec.android.shared.SigPerm;
import org.jssec.android.shared.Utils;

import android.app.Activity;
import android.content.Context;
import android.os.Bundle;
import android.widget.TextView;

public class ProtectedActivity extends Activity {

    // 自社の Signature Permission
    private static final String MY_PERMISSION = "org.jssec.android.permission.protectedapp.MY_PERMISSION
↪";

    // 自社の証明書のハッシュ値
    private static String sMyCertHash = null;
    private static String myCertHash(Context context) {
        if (sMyCertHash == null) {
            if (Utils.isDebuggable(context)) {
                // debug.keystore の"androiddebugkey"の証明書ハッシュ値
                sMyCertHash = "0EFB7236 328348A9 89718BAD DF57F544 D5CCB4AE B9DB34BC 1E29DD26 F77C8255";
            } else {
                // keystore の"my company key"の証明書ハッシュ値
                sMyCertHash = "D397D343 A5CBC10F 4EDDEB7C A10062DE 5690984F 1FB9E88B D7B3A7C2 42E142CA";
            }
        }
        return sMyCertHash;
    }

    private TextView mMessageView;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        mMessageView = (TextView) findViewById(R.id.messageView);

        // ★ポイント 4★ ソースコード上で、独自定義 Signature Permission が自社アプリにより定義されていることを確認
する
        if (!SigPerm.test(this, MY_PERMISSION, myCertHash(this))) {
            mMessageView.setText("独自定義 Signature Permission が自社アプリにより定義されていない。");
            return;
        }

        // 証明書が一致する場合にのみ、処理を続行する
        mMessageView.setText("独自定義 Signature Permission が自社アプリにより定義されていることを確認できた。");
    }
}
```

SigPerm.java

```
package org.jssec.android.shared;

import android.content.Context;
```

(continues on next page)

(continued from previous page)

```
import android.content.pm.PackageManager;
import android.content.pm.PackageManager.NameNotFoundException;
import android.content.pm.PermissionInfo;
import android.os.Build;

import static android.content.pm.PackageManager.CERT_INPUT_SHA256;

public class SigPerm {

    public static boolean test(Context ctx, String sigPermName, String correctHash) {
        if (correctHash == null) return false;
        correctHash = correctHash.replaceAll(" ", "");
        try{
            // sigPermName を定義したアプリのパッケージ名を取得する
            PackageManager pm = ctx.getPackageManager();
            PermissionInfo pi = pm.getPermissionInfo(sigPermName, PackageManager.GET_META_DATA);
            String pkgname = pi.packageName;
            // 非 Signature Permission の場合は失敗扱い
            if (pi.protectionLevel != PermissionInfo.PROTECTION_SIGNATURE) return false;
            // pkgname の実際のハッシュ値と正解のハッシュ値を比較する
            if (Build.VERSION.SDK_INT >= 28) {
                // ★ API Level >= 28 では Package Manager の API で直接検証が可能
                return pm.hasSigningCertificate(pkgname, Utils.hex2Bytes(correctHash), CERT_INPUT_
↪SHA256);
            } else {
                // API Level < 28 の場合は PkgCert を利用し、ハッシュ値を取得して比較する
                return correctHash.equals(PkgCert.hash(ctx, pkgname));
            }
        } catch (NameNotFoundException e){
            return false;
        }
    }
}
```

PkgCert.java

```
package org.jssec.android.shared;

import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

import android.content.Context;
import android.content.pm.PackageInfo;
import android.content.pm.PackageManager;
import android.content.pm.PackageManager.NameNotFoundException;
import android.content.pm.Signature;

public class PkgCert {

    public static boolean test(Context ctx, String pkgname, String correctHash) {
        if (correctHash == null) return false;
        correctHash = correctHash.replaceAll(" ", "");
        return correctHash.equals(hash(ctx, pkgname));
    }
```

(continues on next page)

(continued from previous page)

```
    public static String hash(Context ctx, String pkgname) {
        if (pkgname == null) return null;
        try {
            PackageManager pm = ctx.getPackageManager();
            PackageInfo pkginfo = pm.getPackageInfo(pkgname, PackageManager.GET_SIGNATURES);
            if (pkginfo.signatures.length != 1) return null;    // 複数署名は扱わない
            Signature sig = pkginfo.signatures[0];
            byte[] cert = sig.toByteArray();
            byte[] sha256 = computeSha256(cert);
            return byte2hex(sha256);
        } catch (NameNotFoundException e) {
            return null;
        }
    }

    private static byte[] computeSha256(byte[] data) {
        try {
            return MessageDigest.getInstance("SHA-256").digest(data);
        } catch (NoSuchAlgorithmException e) {
            return null;
        }
    }

    private static String byte2hex(byte[] data) {
        if (data == null) return null;
        final StringBuilder hexadecimal = new StringBuilder();
        for (final byte b : data) {
            hexadecimal.append(String.format("%02X", b));
        }
        return hexadecimal.toString();
    }
}
```

★ポイント 5 ★ Android Studio からメニュー：Build -> Generated Signed APK と選択し、Component を提供するアプリと同じ開発者鍵で署名する。

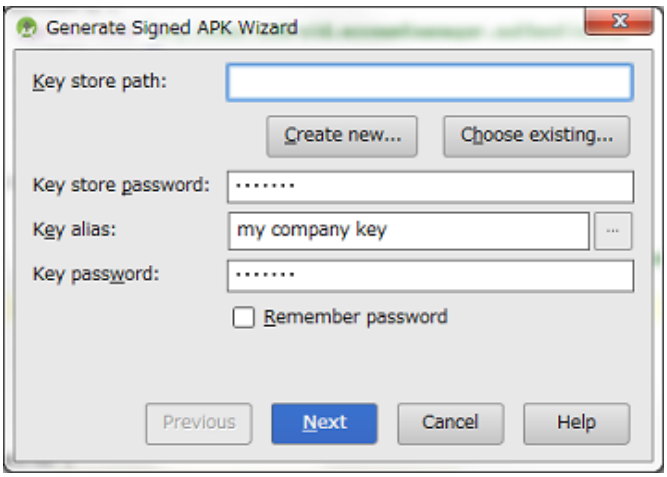

図 5.2.3　Component を提供するアプリと同じ開発者鍵で APK を署名する

ポイント：Component を利用するアプリ

6. 独自定義 Signature Permission は定義しない
7. uses-permission により独自 Permission を利用宣言する
8. ソースコード上で、独自定義 Signature Permission が自社アプリにより定義されていることを確認する
9. 利用先アプリが自社アプリであることを確認する
10. 利用先 Component が Activity の場合、明示的 Intent を使う
11. Component を提供するアプリと同じ開発者鍵で APK を署名する

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="org.jssec.android.permission.userapp">

    <!-- ★ポイント 6★ 独自定義 Signature Permission は定義しない -->

    <!-- ★ポイント 7★ uses-permission により独自 Permission を利用宣言する -->
    <uses-permission
        android:name="org.jssec.android.permission.protectedapp.MY_PERMISSION" />

    <application
        android:allowBackup="false"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name" >
        <activity
            android:name=".UserActivity"
            android:label="@string/app_name"
            android:exported="true" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

UserActivity.java

```
package org.jssec.android.permission.userapp;

import org.jssec.android.shared.PkgCert;
import org.jssec.android.shared.SigPerm;
import org.jssec.android.shared.Utils;

import android.app.Activity;
import android.content.Context;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.widget.Toast;

public class UserActivity extends Activity {

    // 利用先の Activity 情報
```

(continues on next page)

(continued from previous page)

```
    private static final String TARGET_PACKAGE  = "org.jssec.android.permission.protectedapp";
    private static final String TARGET_ACTIVITY = "org.jssec.android.permission.protectedapp.
↪ProtectedActivity";

    // 自社の Signature Permission
    private static final String MY_PERMISSION = "org.jssec.android.permission.protectedapp.MY_PERMISSION
↪";

    // 自社の証明書のハッシュ値
    private static String sMyCertHash = null;
    private static String myCertHash(Context context) {
        if (sMyCertHash == null) {
            if (Utils.isDebuggable(context)) {
                // debug.keystore の"androiddebugkey"の証明書ハッシュ値
                sMyCertHash = "0EFB7236 328348A9 89718BAD DF57F544 D5CCB4AE B9DB34BC 1E29DD26 F77C8255";
            } else {
                // keystore の"my company key"の証明書ハッシュ値
                sMyCertHash = "D397D343 A5CBC10F 4EDDEB7C A10062DE 5690984F 1FB9E88B D7B3A7C2 42E142CA";
            }
        }
        return sMyCertHash;
    }

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }

    public void onSendButtonClicked(View view) {

        // ★ポイント 8★ ソースコード上で、独自定義 Signature Permission が自社アプリにより定義されていることを確認
する
        if (!SigPerm.test(this, MY_PERMISSION, myCertHash(this))) {
            Toast.makeText(this, "独自定義 Signature Permission が自社アプリにより定義されていない。", Toast.
↪LENGTH_LONG).show();
            return;
        }

        // ★ポイント 9★ 利用先アプリが自社アプリであることを確認する
        if (!PkgCert.test(this, TARGET_PACKAGE, myCertHash(this))) {
            Toast.makeText(this, "利用先アプリは自社アプリではない。", Toast.LENGTH_LONG).show();
            return;
        }

        // ★ポイント 10★ 利用先 Component が Activity の場合、明示的 Intent を使う
        try {
            Intent intent = new Intent();
            intent.setClassName(TARGET_PACKAGE, TARGET_ACTIVITY);
            startActivity(intent);
        } catch(Exception e) {
            Toast.makeText(this,
                    String.format("例外発生:%s", e.getMessage()),
                    Toast.LENGTH_LONG).show();
        }
```

(continues on next page)

(continued from previous page)

```
    }
}
```

PkgCertWhitelists.java

```
package org.jssec.android.shared;

import android.content.pm.PackageManager;
import java.util.HashMap;
import java.util.Map;
import android.content.Context;
import android.os.Build;

import static android.content.pm.PackageManager.CERT_INPUT_SHA256;

public class PkgCertWhitelists {
    private Map<String, String> mWhitelists = new HashMap<String, String>();

    public boolean add(String pkgname, String sha256) {
        if (pkgname == null) return false;
        if (sha256 == null) return false;

        sha256 = sha256.replaceAll(" ", "");
        if (sha256.length() != 64) return false;    // SHA-256 は 32 バイト
        sha256 = sha256.toUpperCase();
        if (sha256.replaceAll("[0-9A-F]+", "").length() != 0) return false; // 0-9A-F 以外の文字がある

        mWhitelists.put(pkgname, sha256);
        return true;
    }

    public boolean test(Context ctx, String pkgname) {
        // pkgname に対応する正解のハッシュ値を取得する
        String correctHash = mWhitelists.get(pkgname);
        android.util.Log.d("Partner", "hash=" + correctHash);
        // pkgname の実際のハッシュ値と正解のハッシュ値を比較する
        if (Build.VERSION.SDK_INT >= 28) {
            // ★ API Level >= 28 では Package Manager の API で直接検証が可能
            PackageManager pm = ctx.getPackageManager();
            return pm.hasSigningCertificate(pkgname, hex2Bytes(correctHash), CERT_INPUT_SHA256);
        } else {
            // API Level < 28 の場合は PkgCert の機能を利用する
            return PkgCert.test(ctx, pkgname, correctHash);
        }
    }

    private byte[] hex2Bytes(String s) {
        int len = s.length();
        byte[] data = new byte[len / 2];
        for (int i = 0; i < len; i += 2) {
            data[i / 2] = (byte) ((Character.digit(s.charAt(i), 16) << 4)
                    + Character.digit(s.charAt(i+1), 16));
        }
        return data;
    }
}
```

PkgCert.java

```
package org.jssec.android.shared;

import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

import android.content.Context;
import android.content.pm.PackageInfo;
import android.content.pm.PackageManager;
import android.content.pm.PackageManager.NameNotFoundException;
import android.content.pm.Signature;

public class PkgCert {

    public static boolean test(Context ctx, String pkgname, String correctHash) {
        if (correctHash == null) return false;
        correctHash = correctHash.replaceAll(" ", "");
        return correctHash.equals(hash(ctx, pkgname));
    }

    public static String hash(Context ctx, String pkgname) {
        if (pkgname == null) return null;
        try {
            PackageManager pm = ctx.getPackageManager();
            PackageInfo pkginfo = pm.getPackageInfo(pkgname, PackageManager.GET_SIGNATURES);
            if (pkginfo.signatures.length != 1) return null;    // 複数署名は扱わない
            Signature sig = pkginfo.signatures[0];
            byte[] cert = sig.toByteArray();
            byte[] sha256 = computeSha256(cert);
            return byte2hex(sha256);
        } catch (NameNotFoundException e) {
            return null;
        }
    }

    private static byte[] computeSha256(byte[] data) {
        try {
            return MessageDigest.getInstance("SHA-256").digest(data);
        } catch (NoSuchAlgorithmException e) {
            return null;
        }
    }

    private static String byte2hex(byte[] data) {
        if (data == null) return null;
        final StringBuilder hexadecimal = new StringBuilder();
        for (final byte b : data) {
            hexadecimal.append(String.format("%02X", b));
        }
        return hexadecimal.toString();
    }
}
```

★ポイント 11 ★ Android Studio からメニュー：Build -> Generated Signed APK と選択し、Component を提供するアプリと同じ開発者鍵で署名する。

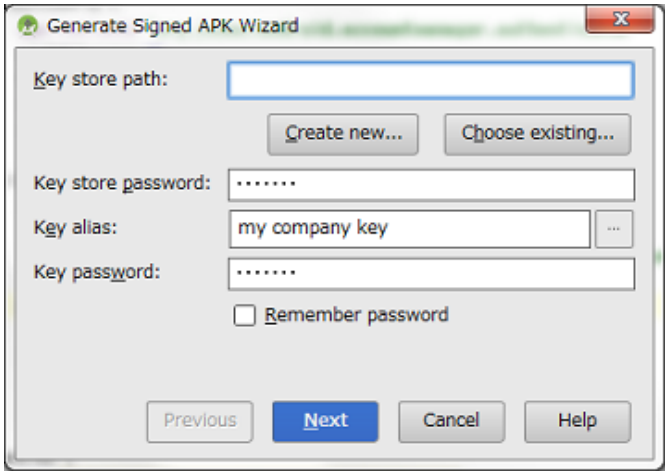

図 5.2.4　Component を提供するアプリと同じ開発者鍵で APK を署名する

Android 9.0 (API Level 28) 以降における署名の検証

Android 9.0 (API Level 28) で APK の署名スキーム V3 が導入され、署名用キーのローテーションができるようになった。これと同時にパッケージの署名関連の API もアップデートされている[*2]。アプリの署名検証の観点から変更点を見ると、PackageManager クラスの新規メソッド hasSigningCertificate() メソッドを使用して検証できるようになった。具体的には、ガイドのサンプルコード PkgCert クラスが行っていた、検証対象のパッケージから署名に使用した証明書を取得しハッシュ値を計算するなどの処理を代替することができる。上に掲載したサンプルコードの SigPerm や PkgCertWhiteLists はこれを反映して API Level が 28 以上の場合に、この新規メソッド hasSigningCertificate() を用いている。hasSigningCertificate() は署名スキーム違いや複数署名に伴う検査の仕方の違いも吸収してくれるため、API Level 28 以降をターゲットとしている場合はこれを用いることをお薦めする[*3]。

5.2.1.3 アプリの証明書のハッシュ値を確認する方法

このガイド文書の各所で出てくるアプリの証明書のハッシュ値を確認する方法を紹介する。厳密には「APK を署名するときに使った開発者鍵の公開鍵証明書の SHA256 ハッシュ値」を確認する方法である。

Keytool により確認する方法

JDK に付属する keytool というプログラムを利用すると開発者鍵の公開鍵証明書のハッシュ値（証明書のフィンガープリントとも言う）を求めることができる。ハッシュ値にはハッシュアルゴリズムの違いにより MD5 や SHA1、SHA256 など様々なものがあるが、このガイド文書では暗号ビット長の安全性を考慮して SHA256 の利用を推奨している。残念なことに Android SDK で利用されている JDK6 に付属する keytool は SHA256 でのハッシュ値出力に対応しておらず、JDK7 以降に付属する keytool を使う必要がある。

Android のデバッグ証明書の内容を keytool で出力する例

```
> keytool -list -v -keystore <キーストアファイル> -storepass <パスワード>

キーストアのタイプ: JKS
```

(continues on next page)

[*2] 具体的な変更内容については Android Developers サイト (https://developer.android.com/reference/android/content/pm/PackageManager) を参照のこと。

[*3] Android 9.0 (API Level 28) の android.content.pm.PackageManager に互換な Android Support Library は本記事を執筆時点でまだ提供されていない。

(continued from previous page)

```
キーストア・プロバイダ: SUN

キーストアには 1 エントリが含まれます

別名: androiddebugkey
作成日: 2012/01/11
エントリ・タイプ: PrivateKeyEntry
証明書チェーンの長さ: 1
証明書 [1]:
所有者: CN=Android Debug, O=Android, C=US
発行者: CN=Android Debug, O=Android, C=US
シリアル番号: 4f0cef98
有効期間の開始日: Wed Jan 11 11:10:32 JST 2012 終了日: Fri Jan 03 11:10:32 JST 2042
証明書のフィンガプリント:
         MD5:  9E:89:53:18:06:B2:E3:AC:B4:24:CD:6A:56:BF:1E:A1
         SHA1: A8:1E:5D:E5:68:24:FD:F6:F1:ED:2F:C3:6E:0F:09:A3:07:F8:5C:0C
         SHA256:␣
↪FB:75:E9:B9:2E:9E:6B:4D:AB:3F:94:B2:EC:A1:F0:33:09:74:D8:7A:CF:42:58:22:A2:56:85:1B:0F:85:C6:35
         署名アルゴリズム名: SHA1withRSA
         バージョン: 3


*******************************************
*******************************************
```

JSSEC 証明書ハッシュ値チェッカーにより確認する方法

JDK7 以降をインストールしなくても、JSSEC 証明書ハッシュ値チェッカーを使えば簡単に証明書ハッシュ値を確認できる。

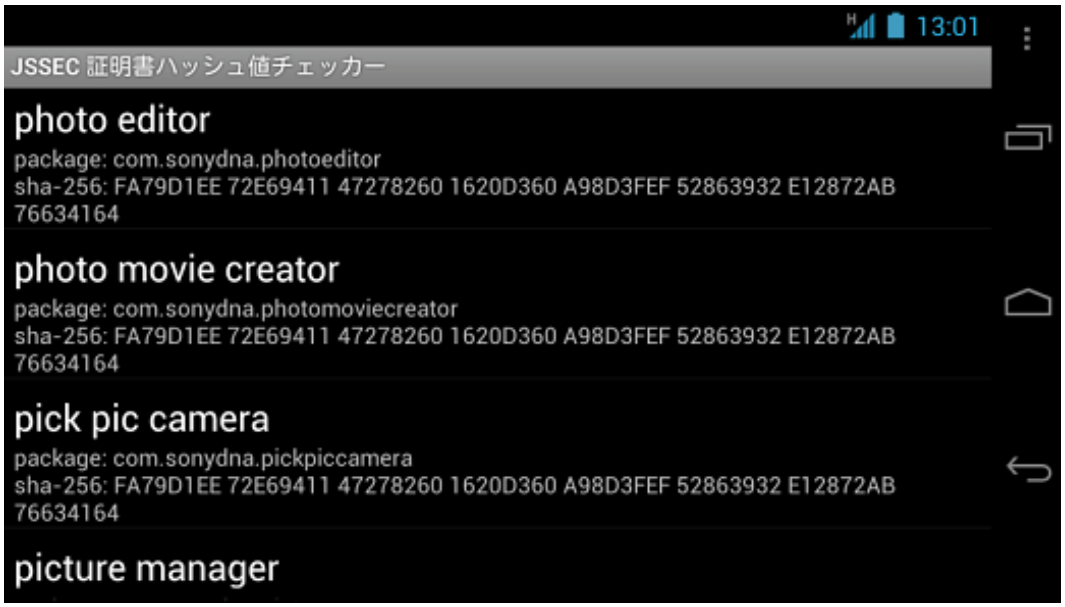

図 5.2.5　JSSEC 証明書ハッシュ値チェッカーによる確認

これは端末にインストールされているアプリの証明書ハッシュ値を一覧表示する Android アプリである。上図中、「sha-256」の右に表示されている 16 進数文字列 64 文字が証明書ハッシュ値である。このガイド文書と一緒に配布しているサンプルコードの「JSSEC CertHash Checker」フォルダがそのソースコード一式である。ビルドして活用していた

だきたい。

5.2.1.4 Android 6.0 以降で Dangerous Permission を利用する方法

アプリに対する Permission 付与のタイミングについて、Android 6.0(API Level 23) でアプリの実装にかかわる仕様変更が行われた。

Android 5.1(API Level 22) 以前の Permission モデル（「5.2.3.6. *Android 6.0 以降の Permission モデルの仕様変更について*」参照）では、アプリが利用宣言している Permission は全てアプリのインストール時に付与される。しかし、Android 6.0 以降では、Dangerous Permission についてはアプリが適切なタイミングで Permission を要求するよう、アプリ開発者が明示的に実装しなければならない。アプリが Permission を要求すると、Android OS はユーザーに対して下記のような確認画面を表示し、その Permission の利用を許可するかどうかの判断を求めることになる。ユーザーが Permission の利用を許可すれば、アプリはその Permission を必要とする処理を実行することができる。

図 5.2.6　Dangerous Permission 利用確認画面

Permission を付与する単位にも変更が加えられている。従来はすべての Permission が一括して付与されていたが、Android 6.0（API Level 23）以降では Permission Group 毎に、Android 8.0(API Level 26) 以降では Permission 個別に付与される。これに伴いユーザー確認画面も個別に表示され、ユーザーは Permission 利用の可・不可について従来よりも柔軟に判断できるようになった。アプリ開発者は、Permission の付与が拒否された場合も考慮して、アプリの仕様や設計を見直す必要がある。

Android 6.0 以降の Permission モデルについての詳細は「5.2.3.6. *Android 6.0 以降の Permission モデルの仕様変更について*」を参照すること。

ポイント：

1. アプリで利用する Permission を利用宣言する
2. 不必要な Permission は利用宣言しない

3. Permission がアプリに付与されているか確認する
4. Permission を要求する（ユーザーに許可を求めるダイアログを表示する）
5. Permission の利用が許可されていない場合の処理を実装する

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="org.jssec.android.permission.permissionrequestingpermissionatruntime" >

    <!-- ★ポイント 1★ アプリで利用する Permission を利用宣言する -->
    <!-- 連絡先情報を読み取る Permission (Protection Level: dangerous) -->
    <uses-permission android:name="android.permission.READ_CONTACTS" />

    <!-- ★ポイント 2★  不必要な Permission は利用宣言しない -->

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme" >
        <activity
            android:name=".MainActivity"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity
            android:name=".ContactListActivity"
            android:exported="false">
        </activity>
    </application>

</manifest>
```

MainActivity.java

```
package org.jssec.android.permission.permissionrequestingpermissionatruntime;

import android.Manifest;
import android.content.Intent;
import android.content.pm.PackageManager;
import android.os.Bundle;
import android.support.v4.app.ActivityCompat;
import android.support.v4.content.ContextCompat;
import android.support.v7.app.AppCompatActivity;
import android.view.View;
import android.widget.Button;
import android.widget.Toast;

public class MainActivity extends AppCompatActivity implements View.OnClickListener {
    private static final int REQUEST_CODE_READ_CONTACTS = 0;
```

(continues on next page)

(continued from previous page)

```
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        Button button = (Button)findViewById(R.id.button);
        button.setOnClickListener(this);
    }

    @Override
    public void onClick(View v) {
        readContacts();
    }

    private void readContacts() {
        // ★ポイント 3★ Permission がアプリに付与されているか確認する
        if (ContextCompat.checkSelfPermission(getApplicationContext(), Manifest.permission.READ_
↪CONTACTS) != PackageManager.PERMISSION_GRANTED) {
            // Permission が付与されていない
            // ★ポイント 4★ Permission を要求する（ユーザーに許可を求めるダイアログを表示する）
            ActivityCompat.requestPermissions(this, new String[]{Manifest.permission.READ_CONTACTS},␣
↪REQUEST_CODE_READ_CONTACTS);
        } else {
            // Permission がすでに付与されている
            showContactList();
        }
    }

    // ユーザー選択の結果を受けるコールバックメソッド
    @Override
    public void onRequestPermissionsResult(int requestCode, String[] permissions, int[] grantResults) {
        switch (requestCode) {
            case REQUEST_CODE_READ_CONTACTS:
                if (grantResults.length > 0 && grantResults[0] == PackageManager.PERMISSION_GRANTED) {
                    // Permission の利用が許可されているので、連絡先情報を利用する処理を実行できる
                    showContactList();
                } else {
                    // Permission の利用が許可されていないため、連絡先情報を利用する処理は実行できない
                    // ★ポイント 5★ Permission の利用が許可されていない場合の処理を実装する
                    Toast.makeText(this, String.format("連絡先の利用が許可されていません"), Toast.LENGTH_
↪LONG).show();
                }
                return;
        }
    }

    // 連絡先一覧を表示
    private void showContactList() {
        // ContactListActivity を起動
        Intent intent = new Intent();
        intent.setClass(getApplicationContext(), ContactListActivity.class);
        startActivity(intent);
    }
}
```

5.2.2 ルールブック

独自 Permission 利用時には以下のルールを守ること。

1. *Android OS* 規定の *Dangerous Permission* はユーザーの資産を保護するためにだけ利用する （必須）
2. 独自定義の *Dangerous Permission* は利用してはならない （必須）
3. 独自定義 *Signature Permission* は *Component* の提供側アプリでのみ定義する （必須）
4. 独自定義 *Signature Permission* は自社アプリにより定義されていることを確認する （必須）
5. 独自定義の *Normal Permission* は利用してはならない （推奨）
6. 独自定義の *Permission* 名はアプリのパッケージ名を拡張した文字列にする （推奨）

5.2.2.1 Android OS 規定の Dangerous Permission はユーザーの資産を保護するためにだけ利用する （必須）

独自定義の Dangerous Permission の利用は非推奨（「5.2.2.2. 独自定義の *Dangerous Permission* は利用してはならない（必須）」参照）のため、ここでは Android OS 規定の Dangerous Permission を前提に話をする。

Dangerous Permission は他の 3 つの Permission と異なり、アプリにその権限を付与するかどうかをユーザーに判断を求める機能がある。Dangerous Permission を利用宣言しているアプリを端末にインストールするとき、次のような画面が表示される。これにより、そのアプリがどのような権限（Dangerous Permission および Normal Permission）を利用しようとしているのかをユーザーが知ることができる。ユーザーが「インストール」をタップすることで、そのアプリは利用宣言した権限が付与され、インストールされるようになっている。

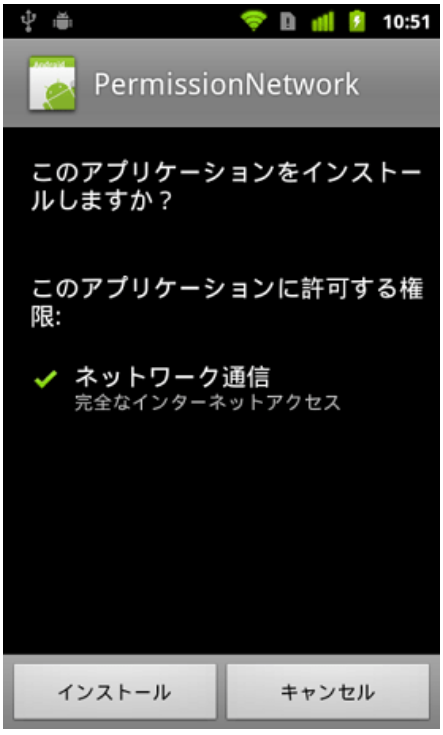

図 5.2.7　Android OS 規定の Dangerous Permission 利用確認画面

アプリの中には、ユーザーの資産とアプリ開発者が保護したい資産がある。それらのうち、Dangerous Permission で保護できるのはユーザーの資産だけであることに注意が必要である。なぜなら、権限の付与がユーザーの判断に委ねられているためである。一方、アプリ開発者が保護したい資産については、この方法では保護できない。

例えば、自社アプリだけと連携する Component は他社アプリからのアクセスを禁止したい場合を考える。このような Component を Dangerous Permission により保護するように実装したとする。他社アプリがインストールされるときに、ユーザーの判断により他社アプリに対して権限の付与を許可してしまうと、保護すべき自社の資産が他社アプリに悪用される危険が生じる。このような場合に自社の資産を保護するためには、独自定義の Signature Permission を使うとよい。

5.2.2.2 独自定義の **Dangerous Permission** は利用してはならない （必須）

独自定義の Dangerous Permission を使用しても、インストール時に「ユーザーに権限の許可を求める」画面が表示されない場合がある。つまり Dangerous Permission の特徴であるユーザーに判断を求める機能が働かないことがあるのだ。よって本ガイドでは「独自定義の Dangerous Permission を利用しない」ことをルールとする。

まず説明のために 2 つのアプリを想定する。1 つは独自の Dangerous Permission を定義し、この Permission により保護した Component を公開するアプリである。これを ProtectedApp とする。もう 1 つは ProtectedApp の Component を悪用しようとする別のアプリでこれを AttackerApp とする。ここで AttackerApp は ProtectedApp が定義した Permission の利用宣言とともに同じ Permission の定義も行っているものとする。

AttackerApp がユーザーの許可なしに ProtectedApp の Component を利用できてしまうケースは以下のような場合に起きる。

1. ユーザーがまず AttackerApp をインストールすると、Dangerous Permission の利用許可を求める画面は表示されずに、そのままインストールが完了してしまう
2. 次に ProtectedApp をインストールすると、ここでも特に警告もなくインストールできてしまう
3. その後、ユーザーが AttackerApp を起動すると、AttackerApp はユーザーの気づかぬうちに ProtectedApp の Component にアクセスできてしまい、場合によっては被害に繋がる

このケースの原因は次の通りである。先に AttackerApp をインストールしようとすると uses-permission により利用宣言された Permission はまだその端末上では定義されていない。このとき Android OS はエラーとすることもなくインストールを続行してしまう。Dangerous Permission のユーザー確認はインストール時だけしか実施されないため、一度インストールされたアプリは権限を許可されたものとして扱われる。したがって後からインストールされるアプリの Component を同名の Dangerous Permission で保護していた場合、ユーザーの許可なく先にインストールされたアプリからその Component が利用できてしまうのである。

なお、Android OS 既定の Dangerous Permission はアプリがインストールされるときにはその存在が保証されているので、uses-permission しているアプリがインストールされるときには必ずユーザー確認画面が表示される。独自定義の Dangerous Permission の場合にだけこの問題は生じる。

現在、このケースで Component へのアクセスを防止するよい方法は見つかっていない。したがって、独自定義の Dangerous Permission は利用してはならない。

5.2.2.3 独自定義 **Signature Permission** は **Component** の提供側アプリでのみ定義する （必須）

自社アプリ間で連携する場合、実行時に Signature Permission をチェックすることでセキュリティを担保できることを「5.2.1.2. 独自定義の *Signature Permission* で自社アプリ連携する方法」で例示した。この仕組みを利用する際には、Protection Level が Signature の独自 Permission の定義は、Component 提供側アプリの AndroidManifest.xml でのみ行い、利用側アプリでは独自の Signature Permission を定義してはならない。

なお、signatureOrSystem Permission についても同様である。

以下がその理由となる。

提供側アプリより先にインストールされた利用側アプリが複数あり、どの利用側アプリも独自定義 Permission の利用宣言とともに Permission の定義もしている場合を考える。この状況で提供側アプリをインストールすると、すべての利用側アプリから提供側アプリにアクセスすることが可能になる。次に、最初にインストールした利用側アプリをアンインストールすると、Permission の定義が削除され、Permission が未定義となる。そのため、残った利用側アプリからの提供側アプリの利用が不可能となってしまう。

このように、利用側アプリで Permission の定義を行うと思わぬ Permission の未定義状態が発生するので、Permission の定義は保護する Component の提供側アプリのみ行い、利用側アプリで Permission を定義するのは避けなければならない。

こうすることで、提供側アプリのインストール時に権限付与が行われ、かつ、アンインストール時に Permission が未定義となり、提供側アプリと Permission の定義の存在期間が必ず一致するので、適正な Component の提供と保護が可能である。なお、独自定義 Signature Permission に関しては、連携するアプリのインストール順によらず、利用側アプリに Permission 利用権限が付与されるため、この議論が成り立つことに注意[*4]。

5.2.2.4 独自定義 **Signature Permission** は自社アプリにより定義されていることを確認する （必須）

AnroidManifest.xml で Signature Permission を宣言し、Component をその Permission で保護しただけでは、実は保護が十分ではない。この詳細はアドバンストセクションの「5.2.3.1. 独自定義 *Signature Permission* を回避できる *Android OS* の特性とその対策」を参照すること。

以下、独自定義 Signature Permission を安全に正しく使う手順である。

まず、AndroidManifest.xml にて次を行う。

1. 保護したい Component のあるアプリの AndroidManifest.xml にて、独自 Signature Permission を定義する（Permission の定義）例： <permission android:name=”xxx” android:protectionLevel=”signature” />
2. 保護したい Component のある AndroidManifest.xml にて、その Component の定義タグの permission 属性で、独自定義 Signature Permission を指定する（Permission の要求宣言）例： <activity android:permission=”xxx” ... >...</activity>
3. 保護したい Component にアクセスする連携アプリの AndroidManifest.xml にて、uses-permission タグに独自定義 Signature Permission を指定する（Permission の利用宣言）例： <uses-permission android:name=”xxx” />

続いて、ソースコード上にて次を実装する。

1. 保護したい Component でリクエストを処理する前に、独自定義した Signature Permission が自社アプリにより定義されたものかどうかを確認し、そうでなければリクエストを無視する（Component 提供側による保護）
2. 保護したい Component にアクセスする前に、独自定義した Signature Permission が自社アプリにより定義されたものかどうかを確認し、そうでなければ Component にアクセスしない（Component 利用側による保護）

最後に Android Studio の署名機能にて次を行う。

1. 連携するすべてのアプリの APK を同じ開発者鍵で署名する

ここで「独自定義した Signature Permission が、自社アプリにより定義されたものかどうかを確認」する必要があるが、具体的な実装方法についてはサンプルコードセクションの「5.2.1.2. 独自定義の *Signature Permission* で自社アプリ連携する方法」を参照すること。

[*4] Normal/Dangerous Permission を利用する場合には、Permission が未定義のまま利用側アプリが先にインストールされると、利用側アプリへの権限の付与が行われず、提供側アプリがインストールされた後もアクセスができない

なお、signatureOrSystem Permission についても同様である。

5.2.2.5 独自定義の **Normal Permission** は利用してはならない （推奨）

Normal Permission を利用するアプリは Android Manifest.xml に uses-permission で利用宣言するだけでその権限を得ることができる。そのため、一度インストールされてしまったマルウェアから Component を保護するような目的に Normal Permission は利用できない。

さらに独自定義 Normal Permission を用いてアプリ間連携を行う場合、連携する各アプリへの Permission の付与はインストール順に依存する。例えば、Permission を定義したコンポーネントを持つアプリよりも先にその Permission を利用宣言したアプリをインストールすると、Permission を定義したアプリをインストールした後も利用側アプリは Permission で保護されたコンポーネントにアクセスすることができない。

インストール順によりアプリ間連携ができなくなる問題を回避する方法として、連携する全てのアプリに Permission を定義することも考えられる。そうすることにより最初に利用側アプリがインストールされた場合でも、全ての利用側アプリが提供側アプリにアクセスすることが可能となる。しかし、最初にインストールした利用側アプリがアンインストールされた際に Permission が未定義な状態となり、他に利用側アプリが存在していても、それらのアプリから提供側アプリにアクセスすることができなくなってしまうのである。

以上のようにアプリの可用性が損なわれる恐れがあることから、独自定義 Normal Permission の利用は控えるべきである。

5.2.2.6 独自定義の **Permission** 名はアプリのパッケージ名を拡張した文字列にする （推奨）

複数のアプリが同じ名前で Permission を定義する場合、先にインストールされたアプリが定義する Protection Level が適用される。先にインストールされたアプリが Normal Permission を定義し、後にインストールされたアプリが同じ名前で Signature Permission を定義した場合、Signature Permission による保護がまったく効かない。悪意がない場合でも、複数のアプリにおいて Permission 名が衝突して意図しない Protection Level で動作する可能性がある。このような事故を防ぐため、Permission 名にはアプリのパッケージ名を入れた方が良い。

```
(パッケージ名).permission.(識別する文字列)
```

例えば、org.jssec.android.sample というパッケージに READ アクセスの Permission を定義するならば、次の様な命名が好ましい。

```
org.jssec.android.sample.permission.READ
```

5.2.3 アドバンスト

5.2.3.1 独自定義 **Signature Permission** を回避できる **Android OS** の特性とその対策

独自定義 Signature Permission は、同じ開発者鍵で署名されたアプリ間だけでアプリ間連携を実現する Permission である。開発者鍵はプライベート鍵であり絶対に公開してはならないものであるため、Signature Permission による保護は自社アプリだけで連携する場合に使われることが多い。

まずは、Android の Dev Guide（https://developer.android.com/guide/topics/security/security.html）で説明されている独自定義 Signature Permission の基本的な使い方を紹介する。ただし、後述するように、この使い方には Permission 回避の問題があることが分かっており、本ガイドに掲載した対策が必要となる。

以下、独自定義 Signature Permission の基本的な使い方である。

1. 保護したい Component のあるアプリの AndroidManifest.xml にて、独自 Signature Permission を定義する例：<permission android:name=”xxx” android:protectionLevel=”signature” />
2. 保護したい Component を持つアプリの AndroidManifest.xml で、保護したい Component に android:permission 属性を指定し、1. で定義した Signature Permission を要求する例: <activity android:permission=”xxx” ... >...</activity>
3. 保護したい Component にアクセスしたい連携アプリの AndroidManifest.xml にて、独自定義 Signature Permission を利用宣言する例： <uses-permission android:name=”xxx” />
4. 連携するすべてのアプリの APK を同じ開発者鍵で署名する

実は、この使い方だけでは、次の条件が成立すると Signature Permission 回避の抜け道ができてしまう。

説明のために独自定義の Signature Permission で保護したアプリを ProtectedApp とし、ProtectedApp とは異なる開発者鍵で署名したアプリを AttackerApp とする。ここで Signature Permission 回避の抜け道とは、AttackerApp は署名が一致していないにもかかわらず、ProtectedApp の Component にアクセス可能になることである。

条件 1. AttackerApp も ProtectedApp が独自定義した Signature Permission と同じ名前で Normal Permission を定義する（厳密には Signature Permission でも構わない）例: <permission android:name=” xxx ” android:protectionLevel=”normal” />

条件 2. AttackerApp は独自定義した Normal Permission を uses-permission で利用宣言する例: <uses-permission android:name=”xxx” />

条件 3. AttackerApp を ProtectedApp より先に端末にインストールする

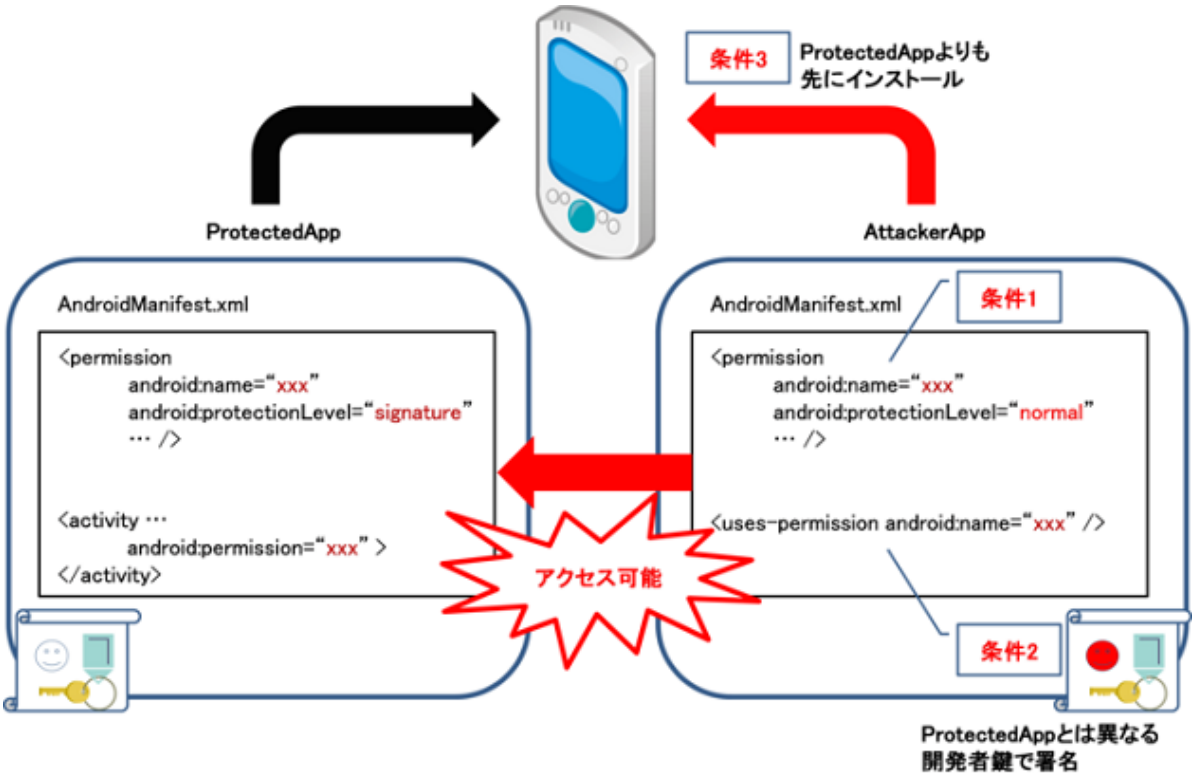

図 5.2.8　Signature Permission 回避の抜け道

条件 1 および条件 2 の成立に必要な ProtectedApp 独自定義の Permission 名は、APK ファイルから AndroidManifest.xml を取り出せば攻撃者にとって容易に知ることができる。条件 3 もユーザーを騙すなどの方法により攻撃者にある程度制

御の余地がある。

このように独自定義の Signature Permission には基本的な使い方だけでは保護を回避されてしまう危険性があり、抜け道をふさぐような対策が必要である。具体的にはルールセクションの「5.2.2.4. 独自定義 *Signature Permission* は自社アプリにより定義されていることを確認する （必須）」に掲載している方法で対処できるので、そちらを参照のこと。

5.2.3.2 ユーザーが AndroidManifest.xml を改ざんする

独自 Permission の Protection Level が意図しないものになるケースは既に説明した。そのことによる不具合を防ぐために、Java のソースコード側で何らかの対応を実施する必要があった。ここでは、AndroidManifest.xml が改ざんされるという視点から、ソースコード側の対応について述べる。改ざんを検知する簡易な実装例を提示するが、犯罪意識をもって改ざんを行うプロのハッカーに対してはほとんど効果がない方法であることに注意すること。

この節はアプリの改ざんに関するものであり、ユーザー自身が悪意を持っているケースである。本来はガイドラインの範囲外であるが、Permission に関する事、これを行うツールがアプリとして公開されている事、から「プロでないハッカーに対する簡易な対策」として述べておくことにした。

Android アプリは、root 権限無しに改ざんできることを頭に置いておく必要がある。なぜなら、AndroidManifest.xml を変更して APK ファイルを再生成、署名するツールが配布されているためである。このツールを使用する事で、誰でも任意のアプリから Permission を削除することが可能になっている。

事例としては INTERNET Permission を取り除いた AndroidManifest.xml から別署名の APK を生成し、アプリに組み込まれた広告モジュールが動作しないようにするケースが多いようである。個人情報がどこかに送信されているかもしれない等の不安が払拭されるということで、この種のツールの存在を評価しているユーザーも存在する。このような行為は、アプリに組み込まれた広告が機能しなくなるため、広告収入を期待している開発者に対して金銭的被害を与える行動であるとも言える。ユーザーのほとんどは罪の意識無くこれらの行為を行っていると思われる。

インターネット Permission を uses-permission で宣言しているアプリが、実行時に自身の AndroidManifest.xml に記載されている Permission を確認する実装例を次に示す。

```
public class CheckPermissionActivity extends Activity {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        // AndroidManifest.xml に定義した Permission を取得
        List<String> list = getDefinedPermissionList();

        // 改ざんを検知する
        if( checkPermissions(list) ){
            // OK
            Log.d("dbg", "OK.");
        }else{
            Log.d("dbg", "manifest file is stale.");
            finish();
        }
    }

    /**
     * AndroidManifest.xml に定義した Permission をリストで取得する
     * @return
```

(continues on next page)

(continued from previous page)

```
     */
    private List<String> getDefinedPermissionList(){
        List<String> list = new ArrayList<String>();
        list.add("android.permission.INTERNET");
        return list;
    }

    /**
     * Permission が変更されていないことを確認する。
     * @param permissionList
     * @return
     */
    private boolean checkPermissions(List<String> permissionList){
        try {
            PackageInfo packageInfo = getPackageManager().getPackageInfo(
                    getPackageName(), PackageManager.GET_PERMISSIONS);
            String[] permissionArray = packageInfo.requestedPermissions;
            if (permissionArray != null) {
                for (String permission : permissionArray) {
                    if(! permissionList.remove(permission)){
                        // 意図しない Permission が付加されている
                        return false;
                    }
                }
            }

            if(permissionList.size() == 0){
                // OK
                return true;
            }

        } catch (NameNotFoundException e) {
        }

        return false;
    }
}
```

5.2.3.3 APK の改ざんを検出する

「5.2.3.2. ユーザーが *AndroidManifest.xml* を改ざんする」ではユーザーによる Permission 改ざんの検出について説明した。しかし、アプリの改ざんは Permission に限らず、リソースを差し替えて別のアプリとしてマーケットで配布するなど、ソースコードを変更することなく改ざんし流用する事例が多様に存在する。ここでは APK ファイルが改ざんされたことを検出するためのより汎用的な方法を紹介する。

APK の改ざんを行うには、APK ファイルを一度展開し、内容を改変した後に再び APK ファイルとして再構成する必要がある。その際に改ざん者は元の開発者の鍵を持ち得ないので、改ざん者自身の鍵で APK を署名することになる。このように APK の改ざんには署名 (証明書) の変更を伴うため、アプリ起動時に APK の証明書と予めソースコードに埋め込んだ開発者の証明書を比較することで改ざんの有無を検出することができる。

以下にサンプルコードを示す。なお、実装例のままではプロのハッカーであれば改ざん検出の無効化が容易である。あくまで簡易な実装例であることを念頭においてアプリへの適用を検討するべきである。

ポイント：

1. 主要な処理を行うまでの間に、アプリの証明書が開発者の証明書であることを確認する

SignatureCheckActivity.java

```
package org.jssec.android.permission.signcheckactivity;

import org.jssec.android.shared.PkgCert;
import org.jssec.android.shared.Utils;

import android.app.Activity;
import android.content.Context;
import android.os.Bundle;
import android.widget.Toast;

public class SignatureCheckActivity extends Activity {
    // 自己証明書のハッシュ値
    private static String sMyCertHash = null;
    private static String myCertHash(Context context) {
        if (sMyCertHash == null) {
            if (Utils.isDebuggable(context)) {
                // debug.keystore の"androiddebugkey"の証明書ハッシュ値
                sMyCertHash = "0EFB7236 328348A9 89718BAD DF57F544 D5CCB4AE B9DB34BC 1E29DD26 F77C8255";
            } else {
                // keystore の"my company key"の証明書ハッシュ値
                sMyCertHash = "D397D343 A5CBC10F 4EDDEB7C A10062DE 5690984F 1FB9E88B D7B3A7C2 42E142CA";
            }
        }
        return sMyCertHash;
    }

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        // ★ポイント 1★ 主要な処理を行うまでの間に、アプリの証明書が開発者の証明書であることを確認する
        if (!PkgCert.test(this, this.getPackageName(), myCertHash(this))) {
            Toast.makeText(this, "自己署名の照合　NG", Toast.LENGTH_LONG).show();
            finish();
            return;
        }
        Toast.makeText(this, "自己署名の照合　OK", Toast.LENGTH_LONG).show();
    }
}
```

PkgCert.java

```
package org.jssec.android.shared;

import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

import android.content.Context;
import android.content.pm.PackageInfo;
import android.content.pm.PackageManager;
import android.content.pm.PackageManager.NameNotFoundException;
```

(continues on next page)

(continued from previous page)

```
import android.content.pm.Signature;

public class PkgCert {

    public static boolean test(Context ctx, String pkgname, String correctHash) {
        if (correctHash == null) return false;
        correctHash = correctHash.replaceAll(" ", "");
        return correctHash.equals(hash(ctx, pkgname));
    }

    public static String hash(Context ctx, String pkgname) {
        if (pkgname == null) return null;
        try {
            PackageManager pm = ctx.getPackageManager();
            PackageInfo pkginfo = pm.getPackageInfo(pkgname, PackageManager.GET_SIGNATURES);
            if (pkginfo.signatures.length != 1) return null;    // 複数署名は扱わない
            Signature sig = pkginfo.signatures[0];
            byte[] cert = sig.toByteArray();
            byte[] sha256 = computeSha256(cert);
            return byte2hex(sha256);
        } catch (NameNotFoundException e) {
            return null;
        }
    }

    private static byte[] computeSha256(byte[] data) {
        try {
            return MessageDigest.getInstance("SHA-256").digest(data);
        } catch (NoSuchAlgorithmException e) {
            return null;
        }
    }

    private static String byte2hex(byte[] data) {
        if (data == null) return null;
        final StringBuilder hexadecimal = new StringBuilder();
        for (final byte b : data) {
            hexadecimal.append(String.format("%02X", b));
        }
        return hexadecimal.toString();
    }
}
```

5.2.3.4 Permission の再委譲問題

アプリが Android OS に保護されている電話帳や GPS といった情報や機能にアクセスするためには Permission を利用宣言しなければならない。Permission を利用宣言し許可されると、そのアプリにはその Permission が委譲されたことになり、その Permission により保護された情報や機能にアクセスできるようになる。

プログラムの組み方によっては、Permission を委譲された（許可された）アプリは Permission で保護されたデータを取得し、そのデータを別のアプリに何の Permission も要求せずに提供することもできてしまう。これは Permission を持たないアプリが Permission で保護されたデータにアクセスできることに他ならない。実質的に Permission を再委譲していることと等価になるので、これを Permission の再委譲問題と呼ぶ。このように Android の Permission セキュリティモデ

ルでは、保護されたデータへのアプリからの直接アクセスだけしか権限管理ができないという仕様上の性質がある。

具体例を 図 5.2.9 に示す。中央のアプリは android.permission.READ_CONTACTS を利用宣言したアプリが連絡先情報を読み取って自分の DB に蓄積している。何の制限もなく Content Provider 経由で蓄積した情報を他のアプリに提供した場合に、Permission の再委譲問題が生じる。

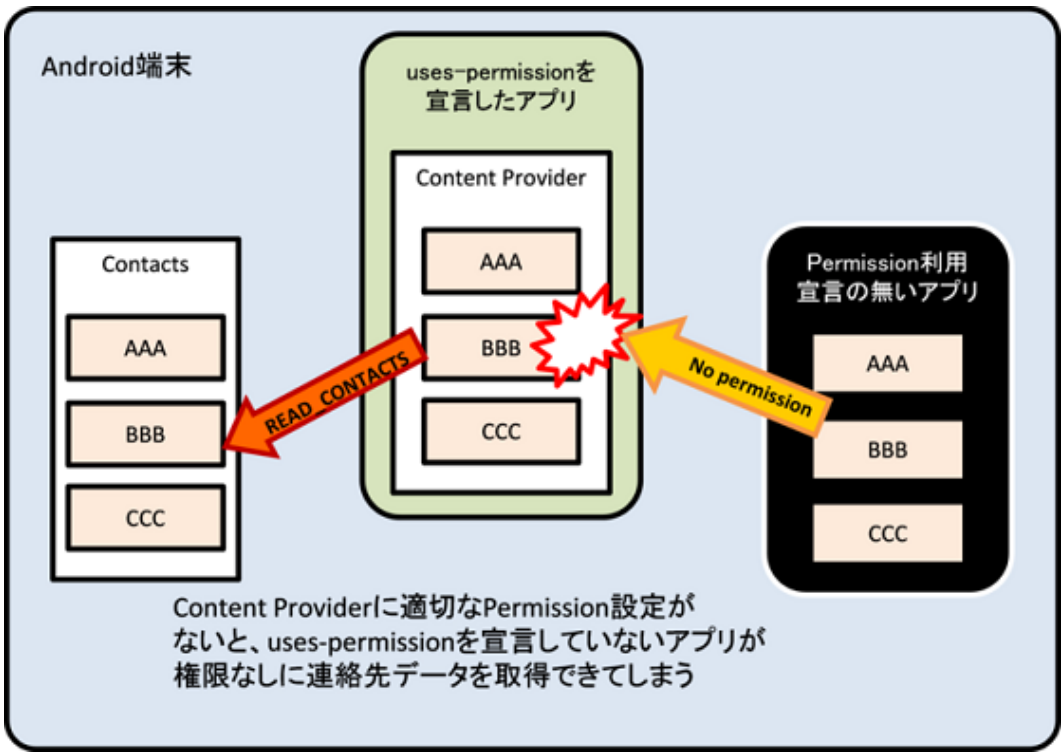

図 5.2.9　Permission を持たないアプリが連絡先情報を取得する

同様の例として、android.permission.CALL_PHONE を利用宣言したアプリが、同 Permission を利用宣言していない他のアプリからの任意の電話番号を受け付け、ユーザーの確認もなくその番号に電話を掛けることができるならば、Permission の再委譲問題がある。

Permission の利用宣言をして得た情報資産・機能資産をほぼそのままの形で他のアプリに二次提供する場合には、提供先アプリに対し同じ Permission を要求するなどして、元の保護水準を維持しなければならない。また情報資産・機能資産の一部分のみを他のアプリに二次提供する場合には、その情報資産・機能資産の一部分が悪用されたときの被害度合に応じた適切な保護が必要である。たとえば前述と同様に同じ Permission を要求したり、ユーザーへ利用許諾を確認したり、「4.1.1.1. 非公開 *Activity* を作る・利用する」「4.1.1.4. 自社限定 *Activity* を作る・利用する」などを利用して対象アプリの制限を設けるなどの保護施策がある。

このような再委譲問題は Permission に限ったことではない。Android アプリでは、アプリに必要な情報･機能を他のアプリやネットワーク・記憶媒体から調達することが一般に行われている。提供元が Android アプリであれば Permission、ネットワークであればログイン、記憶媒体であればアクセス制限といったように、それぞれ調達する際に必要な権限や制限が存在することも多い。こうして調達した情報や機能をその所有者であるユーザーから二次的に他のアプリに提供したり、ネットワークや記憶媒体に転送する際には、ユーザーの意図に反した利用がないように慎重に検討してアプリに対策を施すべきである。必要に応じて、Permission の例と同様に提供先に対して権限の要求や使用の制限を行わなければならない。ユーザーへの利用許諾もその一環である。

以下では、READ_CONTACTS Permission を利用して連絡先 DB から一覧を取得したアプリが、情報提供先のアプリに対して同じ READ_CONTACTS Permission を要求する例を示す。

ポイント：

1. Manifest で提供元と同じ Permission を要求する

```
AndroidManifest.xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="org.jssec.android.permission.transferpermission" >

    <uses-permission android:name="android.permission.READ_CONTACTS"/>

    <application
        android:allowBackup="false"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name=".TransferPermissionActivity"
            android:label="@string/title_activity_transfer_permission" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

        <!-- *** Point1 *** Manifest で提供元と同じ Permission を要求する -->
        <provider
            android:name=".TransferPermissionContentProvider"
            android:authorities="org.jssec.android.permission.transferpermission"
            android:enabled="true"
            android:exported="true"
            android:readPermission="android.permission.READ_CONTACTS" >
        </provider>
    </application>

</manifest>
```

アプリが複数の Permission を要求する必要がある場合は、上記の方法では解決することができない。ソースコード上で Context#checkCallingPermission() や PackageManager#checkPermission() を使用して、呼び出し元のアプリが Manifest ですべての Permission の利用宣言を行っているかどうかを確認することになる。

Activity の場合

```
public void onCreate(Bundle savedInstanceState) {
// ～省略～
    if (checkCallingPermission("android.permission.READ_CONTACTS") == PackageManager.PERMISSION_GRANTED
     && checkCallingPermission("android.permission.WRITE_CONTACTS") == PackageManager.PERMISSION_
↪GRANTED) {
        // 呼び出し元が正しく Permission を利用宣言していた時の処理
        return;
    }
    finish();
}
```

5.2.3.5 独自定義 **Permission** の署名チェック機構について **(Android 5.0** 以降**)**

Android 5.0(API Level 21) 以降の端末では、独自の Permission を定義したアプリにおいて以下のような条件に合致するとインストールに失敗するように仕様変更された。

1. 既に同名の Permission を定義したアプリがインストールされている
2. そのインストール済みのアプリと署名が一致しない

この仕様変更により、保護対象の機能 (Component) の提供側アプリと利用側アプリの双方で Permission を定義した場合には、同じ Permission を定義した署名の異なる他社アプリが両アプリと同時にインストールされるのを防ぐことができる。しかしながら、「5.2.2.3. 独自定義 *Signature Permission* は *Component* の提供側アプリでのみ定義する （必須）」で言及した通り、アプリのアンインストール操作などによって、双方のアプリに Permission を定義するとその Permission が意図せず未定義状態になる場合があるため、この仕様を自社の定義した Signature Permission が他アプリに定義されていないことのチェックに活用することはできないことが分かっている。

結果として、自社限定アプリで独自定義 Signature Permission を利用する場合は、引き続き「5.2.2.3. 独自定義 *Signature Permission* は *Component* の提供側アプリでのみ定義する （必須）」、「5.2.2.4. 独自定義 *Signature Permission* は自社アプリにより定義されていることを確認する （必須）」のルールを順守する必要がある。

5.2.3.6 **Android 6.0** 以降の **Permission** モデルの仕様変更について

Android 6.0(API Level 23) においてアプリの仕様や設計にも影響を及ぼす Permission モデルの仕様変更が行われた。本節では Android 6.0 以降の Permission モデルの概要を解説する。また Android 8.0 以降での変更点についても記載する。

権限の付与・取り消しのタイミング

ユーザー確認が必要な権限（Dangerous Permission）をアプリが利用宣言している場合（「5.2.2.1. *Android OS* 規定の *Dangerous Permission* はユーザーの資産を保護するためにだけ利用する （必須）」参照）、Android 5.1（API Level 22）以前の仕様では、アプリのインストール時にその権限一覧が表示され、ユーザーがすべての権限を許可することでインストールが行われる。この時点で、アプリが利用宣言している (Dangerous Permission 以外の権限を含め) 全ての権限はアプリに付与され、一度付与された権限はアプリが端末からアンインストールされるまで有効である。しかし Android 6.0 以降の仕様では、権限の付与はアプリの実行時に行う仕様となり、アプリのインストール時には権限の付与もユーザーへの確認も行われない。アプリは、Dangerous Permission を必要とする処理を実行する際、事前にその権限がアプリに付与されているかどうかを確認し、権限が付与されていない場合には Android OS に確認画面を表示させ、ユーザーに権限利用の許可を求める必要がある[*5]。ユーザーが確認画面で許可することでその権限はアプリに付与される。ただし、ユーザーは一度アプリに許可した権限 (Dangerous Permission) を、設定メニューを通じて任意のタイミングで取り消すことができる（図 5.2.10）ため、権限がアプリに付与されておらず必要な情報や機能にアクセスすることができない状況においても、アプリが異常な動作を起こすことがないよう適切な処理を実装する必要がある。

[*5] Normal Permission および Signature Permission は Android OS により自動的に付与されるため、ユーザー確認を行う必要はない。

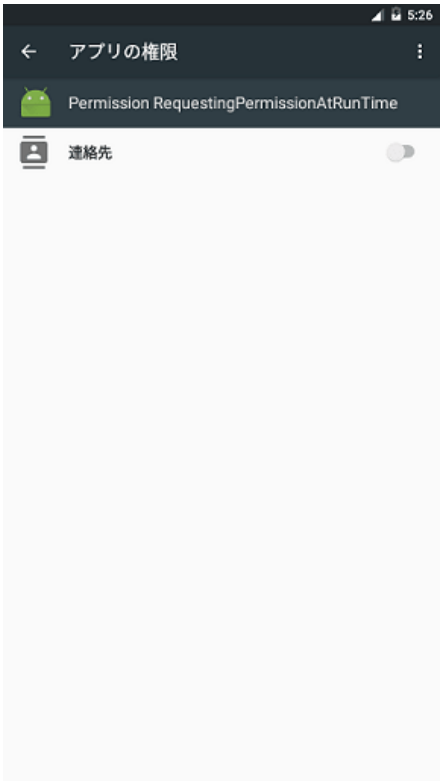

図 5.2.10　アプリの権限設定画面

権限の付与・取り消しの単位

いくつかの Permission はその機能や関連する情報の種類に応じて、Permission Group と呼ばれる単位でグループ化されている。例えば、カレンダー情報の読み取りに必要な Permission である android.permission.READ_CALENDAR と、カレンダー情報の書き込みに必要な Permission である android.permission.WRITE_CALENDAR は、どちらも android.permission-group.CALENDAR という Permission Group に属している。

Android 6.0（API Level 23）以降の Permission モデルにおいて、権限の付与や取り消しはこの Permission Group を単位として行われる。ただし、OS と SDK のバージョンの組み合わせによってこの単位が変わるので注意が必要となる（下記参照）。

- 端末：Android 6.0(API Level 23) 以降、アプリの targetSdkVersion：23~25 の場合

Manifest に android.permission.READ_CALENDAR と android.permission.WRITE_CALENDAR が記載されている状態で、アプリの実行時に android.permission.READ_CALENDAR の要求が行われ、ユーザーがこれを許可すると、Android OS は android.permission.READ_CALENDAR と android.permission.WRITE_CALENDAR の利用が両方とも許可されたとみなし権限が付与される。

- 端末：Android 8.0（API Level 26）以降、アプリの targetSdkVersion：26 以上の場合

要求した Permission の権限のみが付与される。つまり Manifest に android.permission.READ_CALENDAR と android.permission.WRITE_CALENDAR が記載されていても、android.permission.READ_CALENDAR のみを要求しユーザーに許可されたのなら android.permission.READ_CALENDAR の権限のみが付与される。ただし、その後 android.permission.WRITE_CALENDAR が要求された場合は、ユーザーに確認ダイアログが表示されることなく即時に権限が付与される[*6]。

[*6] この場合も、アプリによる android.permission.READ_CALENDAR と android.permission.WRITE_CALENDAR の利用宣言はともに必要である。

また、権限の付与とは異なり、設定メニューからの権限の取り消しは Android 8.0 以降でも Permission Group 単位で行われる。

Permission Group の分類については Developer Reference（https://developer.android.com/intl/ja/guide/topics/security/permissions.html#perm-groups）を参照すること。

仕様変更の影響範囲

アプリの実行時に Permission 要求が必要なのは、端末が Android 6.0 以降で動作していることに加え、アプリの targetSdkVersion が 23 以上に設定されている場合に限られる。端末が Android 5.1 以前で動作している場合や、アプリの targetSdkVersion が 23 未満である場合、権限は従来通りアプリのインストール時にまとめて付与される。ただし、アプリの targetSDKVersion が 23 未満であっても、端末が Android 6.0(API Level 23) 以降であれば、インストールされたアプリの Permission をユーザーが任意のタイミングで取り消すことができるため、意図しないアプリの異常終了が起きる可能性がある。早急に仕様変更に対応するか、アプリの maxSdkVersion を 22 以前に設定して、Android 6.0(API Level 23) 以降の端末にイントールされないようにするなどの対応が必要である。

表 5.2.1: アプリへの権限付与のタイミング

端末の **Android OS** バージョン	アプリの **targetSDKVersion**	アプリへの権限付与のタイミング	ユーザーによる権限制御
≧ 8.0	≧ 26	アプリ実行時 (付与は Permission 単位)	あり
	＜ 26	アプリ実行時 (付与は Group 単位)	あり
	＜ 23	インストール時	あり (早急な対応が必要)
≧ 6.0	≧ 23	アプリ実行時 (付与は Group 単位)	あり
	＜ 23	インストール時	あり (早急な対応が必要)
≦ 5.1	≧ 23	インストール時	なし
	＜ 23	インストール時	なし

ただし、maxSdkVersion の効果は限定的であることに注意が必要である。maxSdkVersion を 22 以前に設定した場合、アプリを Google Play 経由で配布したときには、Android 6.0(API Level 23) 以降の端末が対象アプリのインストール可能端末としてリスト表示されなくなる。一方、Google Play 以外のマーケットプレイスでは maxSdkVersion の値がチェックされないことがあるため、Android 6.0(API Level 23) 以降の端末に対象アプリをインストールできる場合がある。

このように maxSdkVersion の効果は限定的であること、さらに Google が maxSdkVersion の使用を推奨していないことを踏まえ、早急に仕様変更に対応することをお勧めする。

なお、以下のネットワーク通信に関する Permission は、Android 6.0(API Level 23) 以降 Protection Level が dangerous から normal に変更されている。つまり、これらの Permission は利用宣言していても、ユーザーの明示的な許可を必要としないため、今回の仕様変更の影響を受けない。

- android.permission.BLUETOOTH
- android.permission.BLUETOOTH_ADMIN
- android.permission.CHANGE_WIFI_MULTICAST_STATE
- android.permission.CHANGE_WIFI_STATE

- android.permission.CHANGE_WIMAX_STATE
- android.permission.DISABLE_KEYGUARD
- android.permission.INTERNET
- android.permission.NFC

5.3 Account Manager に独自アカウントを追加する

Account Manager はアプリがオンラインサービスへアクセスするために必要となるアカウント情報（アカウント名、パスワード）および認証トークンを一元管理する Android OS の仕組みである[*7]。ユーザーは事前にアカウント情報を Account Manager に登録しておき、アプリがオンラインサービスにアクセスしようとしたときにユーザーの許可を得て、Account Manager がアプリに認証トークンを自動提供する仕組みである。パスワードという極めてセンシティブな情報をアプリが扱わなくて済むことが Account Manager の利点である。

Account Manager を使用したアカウント管理機能は 図 5.3.1 のような構成となる。「利用アプリ」は認証トークンの提供を受けてオンラインサービスにアクセスするアプリであり、前述のアプリのことである。一方、「Authenticator アプリ」は Account Manager の機能拡張であり、Authenticator と呼ばれるオブジェクトを Account Manager に提供することにより、Account Manager がそのオンラインサービスのアカウント情報および認証トークンを一元管理できるようになる。利用アプリと Authenticator アプリは別のアプリである必要はなく、一つのアプリとして実装することもできる。

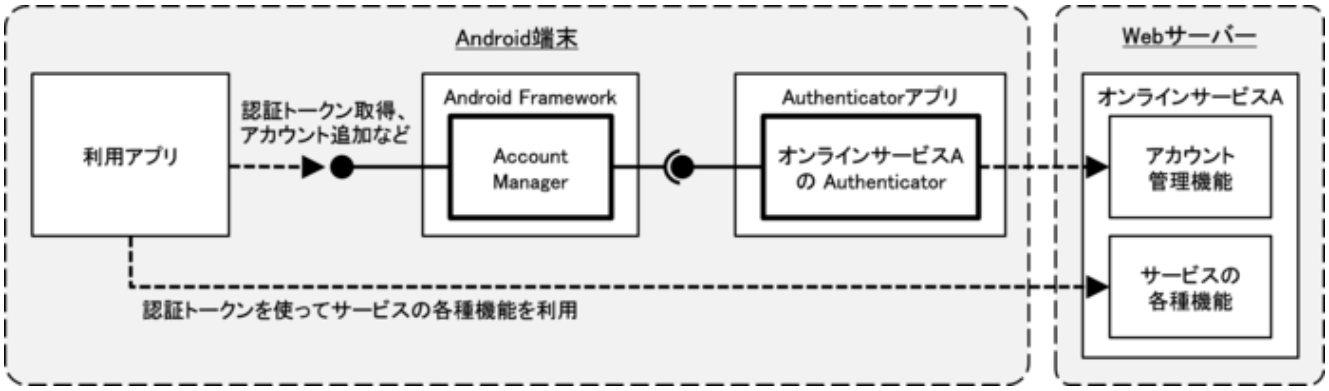

図 5.3.1　Manager を使用したアカウント管理機能の構成

本来、利用アプリと Authenticator アプリは開発者の署名鍵が異なっていてもよい。しかし Android 4.0.x の端末に限り Android Framework のバグがあり、利用アプリと Authenticator アプリの署名鍵が異なっていると利用アプリで例外が発生してしまい、独自アカウントが利用できない。ここで紹介するサンプルコードはこの不具合には対応できていない。詳しくは「5.3.3.2. *Android 4.0.x では利用アプリと Authenticator アプリの署名鍵が異なると例外が発生する*」を参照すること。

5.3.1 サンプルコード

Authenticator アプリのサンプルとして「5.3.1.1. 独自アカウントを作る」を、利用アプリのサンプルとして「5.3.1.2. 独自アカウントを利用する」を用意した。JSSEC の Web サイトで配布しているサンプルコード一式ではそれぞれ AccountManager Authenticator および AccountManager User に対応している。

[*7] Account Manager はオンラインサービスとの同期の仕組みも提供するが、本節では扱っていない。

5.3.1.1 独自アカウントを作る

ここでは Account Manager が独自アカウントを扱えるようにする Authenticator アプリのサンプルコードを紹介する。このアプリはホーム画面から起動できる Activity は存在しない。もう一つのサンプルアプリ「5.3.1.2. 独自アカウントを利用する」から Account Manager 経由で間接的に呼び出されることに注意してほしい。

ポイント：

1. Authenticator を提供する Service は非公開 Service とする
2. ログイン画面 Activity は Authenticator アプリで実装する
3. ログイン画面 Acitivity は公開 Activity とする
4. KEY_INTENT には、ログイン画面 Activity のクラス名を指定した明示的 Intent を与える
5. アカウント情報や認証トークンなどのセンシティブな情報はログ出力しない
6. Account Manager にパスワードを保存しない
7. Authenticator とオンラインサービスとの通信は HTTPS で行う

AndroidManifest.xml にて Authenticator の IBinder を Account Manager に提供するサービスを定義。meta-data にて Authenticator を記述したリソース XML ファイルを指定。

```
AccountManager Authenticator/AndroidManifest.xml
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="org.jssec.android.accountmanager.authenticator"
    xmlns:tools="http://schemas.android.com/tools">

    <!-- Authenticator を実装するのに必要な Permission -->
    <uses-permission android:name="android.permission.GET_ACCOUNTS" />
    <uses-permission android:name="android.permission.AUTHENTICATE_ACCOUNTS" />

    <application
        android:allowBackup="false"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name" >

        <!-- Authenticator の IBinder を AccountManager に提供するサービス -->
        <!-- ★ポイント 1★ Authenticator を提供する Service は非公開 Service とする -->
        <service
            android:name=".AuthenticationService"
            android:exported="false" >
            <!-- intent-filter と meta-data はお決まりのパターン -->
            <intent-filter>
                <action android:name="android.accounts.AccountAuthenticator" />
            </intent-filter>
            <meta-data
                android:name="android.accounts.AccountAuthenticator"
                android:resource="@xml/authenticator" />
        </service>

        <!-- アカウントを追加するときなどに表示されるログイン画面用の Activity -->
        <!-- ★ポイント 2★ ログイン画面 Activity は Authenticator アプリで実装する -->
        <!-- ★ポイント 3★ ログイン画面 Activity は公開 Activity とする -->
        <activity
```

(continues on next page)

(continued from previous page)

```
            android:name=".LoginActivity"
            android:exported="true"
            android:label="@string/login_activity_title"
            android:theme="@android:style/Theme.Dialog"
            tools:ignore="ExportedActivity" />
    </application>

</manifest>
```

XML ファイルで Authenticator を定義。独自アカウントのアカウントタイプ等を指定する。

```
res/xml/authenticator.xml
<account-authenticator xmlns:android="http://schemas.android.com/apk/res/android"
    android:accountType="org.jssec.android.accountmanager"
    android:icon="@drawable/ic_launcher"
    android:label="@string/label"
    android:smallIcon="@drawable/ic_launcher"
    android:customTokens="true" />
```

Authenticator のインスタンスを Account Manager に提供するサービス。このサンプルで実装する Authenticator である JssecAuthenticator クラスのインスタンスを onBind() で return するだけの簡単な実装でよい。

```
AuthenticationService.java
package org.jssec.android.accountmanager.authenticator;

import android.app.Service;
import android.content.Intent;
import android.os.IBinder;

public class AuthenticationService extends Service {

    private JssecAuthenticator mAuthenticator;

    @Override
    public void onCreate() {
        mAuthenticator = new JssecAuthenticator(this);
    }

    @Override
    public IBinder onBind(Intent intent) {
        return mAuthenticator.getIBinder();
    }
}
```

このサンプルで実装する Authenticator である JssecAuthenticator。AbstractAccountAuthenticator を継承して abstract メソッドをすべて実装する。これらのメソッドは Account Manager から呼ばれる。addAccount() および getAuthToken() では、オンラインサービスから認証トークンを取得するための LoginActivity を起動する intent を Account Manager に返している。

```
JssecAuthenticator.java
package org.jssec.android.accountmanager.authenticator;

import android.accounts.AbstractAccountAuthenticator;
```

(continues on next page)

(continued from previous page)

```
import android.accounts.Account;
import android.accounts.AccountAuthenticatorResponse;
import android.accounts.AccountManager;
import android.accounts.NetworkErrorException;
import android.content.Context;
import android.content.Intent;
import android.os.Bundle;

public class JssecAuthenticator extends AbstractAccountAuthenticator {

    public static final String JSSEC_ACCOUNT_TYPE = "org.jssec.android.accountmanager";
    public static final String JSSEC_AUTHTOKEN_TYPE = "webservice";
    public static final String JSSEC_AUTHTOKEN_LABEL = "JSSEC Web Service";
    public static final String RE_AUTH_NAME = "reauth_name";

    protected final Context mContext;

    public JssecAuthenticator(Context context) {
        super(context);
        mContext = context;
    }

    @Override
    public Bundle addAccount(AccountAuthenticatorResponse response, String accountType,
            String authTokenType, String[] requiredFeatures, Bundle options)
            throws NetworkErrorException {

        AccountManager am = AccountManager.get(mContext);
        Account[] accounts = am.getAccountsByType(JSSEC_ACCOUNT_TYPE);
        Bundle bundle = new Bundle();
        if (accounts.length > 0) {
            // 本サンプルコードではアカウントが既に存在する場合はエラーとする
            bundle.putString(AccountManager.KEY_ERROR_CODE, String.valueOf(-1));
            bundle.putString(AccountManager.KEY_ERROR_MESSAGE,
                    mContext.getString(R.string.error_account_exists));

        } else {
            // ★ポイント 2★ ログイン画面 Activity は Authenticator アプリで実装する
            // ★ポイント 4★ KEY_INTENT には、ログイン画面 Activity のクラス名を指定した明示的 Intent を与える
            Intent intent = new Intent(mContext, LoginActivity.class);
            intent.putExtra(AccountManager.KEY_ACCOUNT_AUTHENTICATOR_RESPONSE, response);

            bundle.putParcelable(AccountManager.KEY_INTENT, intent);
        }
        return bundle;
    }

    @Override
    public Bundle getAuthToken(AccountAuthenticatorResponse response, Account account,
            String authTokenType, Bundle options) throws NetworkErrorException {

        Bundle bundle = new Bundle();
        if (accountExist(account)) {
            // ★ポイント 4★ KEY_INTENT には、ログイン画面 Activity のクラス名を指定した明示的 Intent を与える
            Intent intent = new Intent(mContext, LoginActivity.class);
```

(continues on next page)

(continued from previous page)

```
            intent.putExtra(RE_AUTH_NAME, account.name);
            intent.putExtra(AccountManager.KEY_ACCOUNT_AUTHENTICATOR_RESPONSE, response);
            bundle.putParcelable(AccountManager.KEY_INTENT, intent);
        } else {
            // 指定されたアカウントが存在しない場合はエラーとする
            bundle.putString(AccountManager.KEY_ERROR_CODE, String.valueOf(-2));
            bundle.putString(AccountManager.KEY_ERROR_MESSAGE,
                    mContext.getString(R.string.error_account_not_exists));
        }
        return bundle;
    }

    @Override
    public String getAuthTokenLabel(String authTokenType) {
        return JSSEC_AUTHTOKEN_LABEL;
    }

    @Override
    public Bundle confirmCredentials(AccountAuthenticatorResponse response, Account account,
            Bundle options) throws NetworkErrorException {
        return null;
    }

    @Override
    public Bundle editProperties(AccountAuthenticatorResponse response, String accountType) {
        return null;
    }

    @Override
    public Bundle updateCredentials(AccountAuthenticatorResponse response, Account account,
            String authTokenType, Bundle options) throws NetworkErrorException {
        return null;
    }

    @Override
    public Bundle hasFeatures(AccountAuthenticatorResponse response, Account account,
            String[] features) throws NetworkErrorException {
        Bundle result = new Bundle();
        result.putBoolean(AccountManager.KEY_BOOLEAN_RESULT, false);
        return result;
    }

    private boolean accountExist(Account account) {
        AccountManager am = AccountManager.get(mContext);
        Account[] accounts = am.getAccountsByType(JSSEC_ACCOUNT_TYPE);
        for (Account ac : accounts) {
            if (ac.equals(account)) {
                return true;
            }
        }
        return false;
    }
}
```

オンラインサービスにアカウント名、パスワードを送信してログイン認証を行い、その結果として認証トークンを取得

する LoginActivity。新規アカウント追加および認証トークン再取得の場合に表示される。オンラインサービスへの実際のアクセスは WebService クラス内で実装されるものとしている。

LoginActivity.java

```
package org.jssec.android.accountmanager.authenticator;


import org.jssec.android.accountmanager.webservice.WebService;

import android.accounts.Account;
import android.accounts.AccountAuthenticatorActivity;
import android.accounts.AccountManager;
import android.content.Intent;
import android.os.Bundle;
import android.text.InputType;
import android.text.TextUtils;
import android.util.Log;
import android.view.View;
import android.view.Window;
import android.widget.EditText;

public class LoginActivity extends AccountAuthenticatorActivity {
    private static final String TAG = AccountAuthenticatorActivity.class.getSimpleName();
    private String mReAuthName = null;
    private EditText mNameEdit = null;
    private EditText mPassEdit = null;

    @Override
    public void onCreate(Bundle icicle) {
        super.onCreate(icicle);

        // アラートアイコン表示
        requestWindowFeature(Window.FEATURE_LEFT_ICON);
        setContentView(R.layout.login_activity);
        getWindow().setFeatureDrawableResource(Window.FEATURE_LEFT_ICON,
                android.R.drawable.ic_dialog_alert);

        // widget を見つけておく
        mNameEdit = (EditText) findViewById(R.id.username_edit);
        mPassEdit = (EditText) findViewById(R.id.password_edit);

        // ★ポイント 3★ ログイン画面 Activity は公開 Activity として他のアプリからの攻撃アクセスを想定する
        // 外部入力は Intent#extras の String 型の RE_AUTH_NAME だけしか扱わない
        // この外部入力 String は TextEdit#setText()、WebService#login()、new Account() に
        // 引数として渡されるが、どんな文字列が与えられても問題が起きないことを確認している
        mReAuthName = getIntent().getStringExtra(JssecAuthenticator.RE_AUTH_NAME);
        if (mReAuthName != null) {
            // ユーザー名指定で LoginActivity が呼び出されたので、ユーザー名を編集不可とする
            mNameEdit.setText(mReAuthName);
            mNameEdit.setInputType(InputType.TYPE_NULL);
            mNameEdit.setFocusable(false);
            mNameEdit.setEnabled(false);
        }
    }

    // ログインボタン押下時に実行される
```

(continues on next page)

(continued from previous page)

```
    public void handleLogin(View view) {
        String name = mNameEdit.getText().toString();
        String pass = mPassEdit.getText().toString();

        if (TextUtils.isEmpty(name) || TextUtils.isEmpty(pass)) {
            // 入力値が不正である場合の処理
            setResult(RESULT_CANCELED);
            finish();
        }

        // 入力されたアカウント情報によりオンラインサービスにログインする
        WebService web = new WebService();
        String authToken = web.login(name, pass);
        if (TextUtils.isEmpty(authToken)) {
            // 認証が失敗した場合の処理
            setResult(RESULT_CANCELED);
            finish();
        }

        //  以下、ログイン成功時の処理

        // ★ポイント 5★ アカウント情報や認証トークンなどのセンシティブな情報はログ出力しない
        Log.i(TAG, "WebService login succeeded");


        if (mReAuthName == null) {
            // ログイン成功したアカウントを AccountManager に登録する
            // ★ポイント 6★ Account Manager にパスワードを保存しない
            AccountManager am = AccountManager.get(this);
            Account account = new Account(name, JssecAuthenticator.JSSEC_ACCOUNT_TYPE);
            am.addAccountExplicitly(account, null, null);
            am.setAuthToken(account, JssecAuthenticator.JSSEC_AUTHTOKEN_TYPE, authToken);
            Intent intent = new Intent();
            intent.putExtra(AccountManager.KEY_ACCOUNT_NAME, name);
            intent.putExtra(AccountManager.KEY_ACCOUNT_TYPE,
                    JssecAuthenticator.JSSEC_ACCOUNT_TYPE);

            setAccountAuthenticatorResult(intent.getExtras());
            setResult(RESULT_OK, intent);
        } else {
            // 認証トークンを返却する
            Bundle bundle = new Bundle();
            bundle.putString(AccountManager.KEY_ACCOUNT_NAME, name);
            bundle.putString(AccountManager.KEY_ACCOUNT_TYPE,
                    JssecAuthenticator.JSSEC_ACCOUNT_TYPE);
            bundle.putString(AccountManager.KEY_AUTHTOKEN, authToken);
            setAccountAuthenticatorResult(bundle);
            setResult(RESULT_OK);
        }
        finish();
    }
}
```

実際には WebService クラスはダミー実装となっており、常に認証が成功し固定文字列を認証トークンとして返すサンプル実装になっている。

```
WebService.java
package org.jssec.android.accountmanager.webservice;

public class WebService {

    /**
     * オンラインサービスのアカウント管理機能にアクセスする想定
     *
     * @param username アカウント名文字列
     * @param password パスワード文字列
     * @return 認証トークンを返す
     */
    public String login(String username, String password) {
        // ★ポイント 7★ Authenticator とオンラインサービスとの通信は HTTPS で行う
        // 実際には、サーバーとの通信処理を実装するが、 サンプルにつき割愛
        return getAuthToken(username, password);
    }

    private String getAuthToken(String username, String password) {
        // 実際にはサーバーから、ユニーク性と推測不可能性を保証された値を取得するが
        // サンプルにつき、通信は行わずに固定値を返す
        return "c2f981bda5f34f90c0419e171f60f45c";
    }
}
```

5.3.1.2 独自アカウントを利用する

独自アカウントの追加と認証トークンの取得を行うアプリのサンプルコードを以下に示す。もう一つのサンプルアプリ「5.3.1.1. 独自アカウントを作る」が端末にインストールされているときに、独自アカウントの追加や認証トークンの取得ができる。「アクセスリクエスト」画面は両アプリの署名鍵が異なる場合にだけ表示される。

図 5.3.2　サンプルアプリ AccountManager User の動作画面

ポイント：

1. Authenticator が正規のものであることを確認してからアカウント処理を実施する

利用アプリの AndroidManifest.xml。必要な Permission を利用宣言。必要な Permission については「5.3.3.1. *Account Manager の利用と Permission*」を参照。

AccountManager User/AndroidManifest.xml

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="org.jssec.android.accountmanager.user" >

    <uses-permission android:name="android.permission.GET_ACCOUNTS" />
    <uses-permission android:name="android.permission.MANAGE_ACCOUNTS" />
    <uses-permission android:name="android.permission.USE_CREDENTIALS" />

    <application
        android:allowBackup="false"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name=".UserActivity"
            android:label="@string/app_name"
            android:exported="true" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

利用アプリの Activity。画面上のボタンをタップすると addAcount() または getAuthToken() が実行される。指定のアカウントタイプに対応した Authenticator が偽物であるケースがあるので、正規の Authenticator であることを確認してからアカウント処理を始めていることに注意。

UserActivity.java

```
package org.jssec.android.accountmanager.user;

import java.io.IOException;

import org.jssec.android.shared.PkgCert;
import org.jssec.android.shared.Utils;

import android.accounts.Account;
import android.accounts.AccountManager;
import android.accounts.AccountManagerCallback;
import android.accounts.AccountManagerFuture;
import android.accounts.AuthenticatorDescription;
import android.accounts.AuthenticatorException;
import android.accounts.OperationCanceledException;
import android.app.Activity;
import android.content.Context;
import android.os.Bundle;
import android.view.View;
import android.widget.TextView;

public class UserActivity extends Activity {

    // 利用する Authenticator の情報
    private static final String JSSEC_ACCOUNT_TYPE = "org.jssec.android.accountmanager";
```

(continues on next page)

(continued from previous page)

```
    private static final String JSSEC_TOKEN_TYPE = "webservice";
    private TextView mLogView;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.user_activity);

        mLogView = (TextView)findViewById(R.id.logview);
    }

    public void addAccount(View view) {
        logLine();
        logLine("新しいアカウントを追加します");

        // ★ポイント 1★ Authenticator が正規のものであることを確認してからアカウント処理を実施する
        if (!checkAuthenticator()) return;

        AccountManager am = AccountManager.get(this);

        am.addAccount(JSSEC_ACCOUNT_TYPE, JSSEC_TOKEN_TYPE, null, null, this,
                new AccountManagerCallback<Bundle>() {
                    @Override
                    public void run(AccountManagerFuture<Bundle> future) {
                        try {
                            Bundle result = future.getResult();
                            String type = result.getString(AccountManager.KEY_ACCOUNT_TYPE);
                            String name = result.getString(AccountManager.KEY_ACCOUNT_NAME);
                            if (type != null && name != null) {
                                logLine("以下のアカウントを追加しました：");
                                logLine("    アカウント種別: %s", type);
                                logLine("    アカウント名: %s", name);
                            } else {
                                String code = result.getString(AccountManager.KEY_ERROR_CODE);
                                String msg = result.getString(AccountManager.KEY_ERROR_MESSAGE);
                                logLine("アカウントが追加できませんでした");
                                logLine("  エラーコード %s: %s", code, msg);
                            }
                        } catch (OperationCanceledException e) {
                        } catch (AuthenticatorException e) {
                        } catch (IOException e) {
                        }
                    }
                },
                null);
    }

    public void getAuthToken(View view) {
        logLine();
        logLine("トークンを取得します");

        // ★ポイント 1★ Authenticator が正規のものであることを確認してからアカウント処理を実施する
        if (!checkAuthenticator()) return;

        AccountManager am = AccountManager.get(this);
```

(continues on next page)

(continued from previous page)

```
        Account[] accounts = am.getAccountsByType(JSSEC_ACCOUNT_TYPE);
        if (accounts.length > 0) {
            Account account = accounts[0];
            am.getAuthToken(account, JSSEC_TOKEN_TYPE, null, this,
                    new AccountManagerCallback<Bundle>() {
                        @Override
                        public void run(AccountManagerFuture<Bundle> future) {
                            try {
                                Bundle result = future.getResult();
                                String name = result.getString(AccountManager.KEY_ACCOUNT_NAME);
                                String authtoken = result.getString(AccountManager.KEY_AUTHTOKEN);
                                logLine("  %s さんのトークン:", name);
                                if (authtoken != null) {
                                    logLine("    %s", authtoken);
                                } else {
                                    logLine("    取得できませんでした");
                                }
                            } catch (OperationCanceledException e) {
                                logLine("  例外: %s",e.getClass().getName());
                            } catch (AuthenticatorException e) {
                                logLine("  例外: %s",e.getClass().getName());
                            } catch (IOException e) {
                                logLine("  例外: %s",e.getClass().getName());
                            }
                        }
                    }, null);
        } else {
            logLine("アカウントが登録されていません");
        }
    }

    // ★ポイント 1★ Authenticator が正規のものであることを確認する
    private boolean checkAuthenticator() {
        AccountManager am = AccountManager.get(this);
        String pkgname = null;
        for (AuthenticatorDescription ad : am.getAuthenticatorTypes()) {
            if (JSSEC_ACCOUNT_TYPE.equals(ad.type)) {
                pkgname = ad.packageName;
                break;
            }
        }

        if (pkgname == null) {
            logLine("Authenticator が見つかりません");
            return false;
        }

        logLine("  アカウントタイプ： %s", JSSEC_ACCOUNT_TYPE);
        logLine("  Authenticator のパッケージ名：");
        logLine("    %s", pkgname);


        if (!PkgCert.test(this, pkgname, getTrustedCertificateHash(this))) {
            logLine("  正規の Authenticator ではありません（証明書不一致）");
            return false;
```

(continues on next page)

(continued from previous page)

```
        }

        logLine("  正規の Authenticator です");
        return true;
    }

    // 正規の Authenticator アプリの証明書ハッシュ値
    // サンプルアプリ JSSEC CertHash Checker で証明書ハッシュ値は確認できる
    private String getTrustedCertificateHash(Context context) {
        if (Utils.isDebuggable(context)) {
            // debug.keystore の "androiddebugkey"の証明書ハッシュ値
            return "0EFB7236 328348A9 89718BAD DF57F544 D5CCB4AE B9DB34BC 1E29DD26 F77C8255";
        } else {
            // keystore の "my company key"の証明書ハッシュ値
            return "D397D343 A5CBC10F 4EDDEB7C A10062DE 5690984F 1FB9E88B D7B3A7C2 42E142CA";
        }
    }

    private void log(String str) {
        mLogView.append(str);
    }

    private void logLine(String line) {
        log(line + "\n");
    }

    private void logLine(String fmt, Object... args) {
        logLine(String.format(fmt, args));
    }

    private void logLine() {
        log("\n");
    }
}
```

PkgCert.java

```
package org.jssec.android.shared;

import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

import android.content.Context;
import android.content.pm.PackageInfo;
import android.content.pm.PackageManager;
import android.content.pm.PackageManager.NameNotFoundException;
import android.content.pm.Signature;

public class PkgCert {

    public static boolean test(Context ctx, String pkgname, String correctHash) {
        if (correctHash == null) return false;
        correctHash = correctHash.replaceAll(" ", "");
        return correctHash.equals(hash(ctx, pkgname));
    }
```

(continues on next page)

(continued from previous page)

```
    public static String hash(Context ctx, String pkgname) {
        if (pkgname == null) return null;
        try {
            PackageManager pm = ctx.getPackageManager();
            PackageInfo pkginfo = pm.getPackageInfo(pkgname, PackageManager.GET_SIGNATURES);
            if (pkginfo.signatures.length != 1) return null;    // 複数署名は扱わない
            Signature sig = pkginfo.signatures[0];
            byte[] cert = sig.toByteArray();
            byte[] sha256 = computeSha256(cert);
            return byte2hex(sha256);
        } catch (NameNotFoundException e) {
            return null;
        }
    }

    private static byte[] computeSha256(byte[] data) {
        try {
            return MessageDigest.getInstance("SHA-256").digest(data);
        } catch (NoSuchAlgorithmException e) {
            return null;
        }
    }

    private static String byte2hex(byte[] data) {
        if (data == null) return null;
        final StringBuilder hexadecimal = new StringBuilder();
        for (final byte b : data) {
            hexadecimal.append(String.format("%02X", b));
        }
        return hexadecimal.toString();
    }
}
```

5.3.2 ルールブック

Authenticator アプリを実装する際には以下のルールを守ること。

1. *Authenticator* を提供する *Service* は非公開 *Service* とする （必須）
2. ログイン画面 *Activity* は *Authenticator* アプリで実装する （必須）
3. ログイン画面 *Activity* は公開 *Activity* として他のアプリからの攻撃アクセスを想定する （必須）
4. *KEY_INTENT* には、ログイン画面 *Activity* のクラス名を指定した明示的 *Intent* を与える （必須）
5. アカウント情報や認証トークンなどのセンシティブな情報はログ出力しない （必須）
6. *Account Manager* にパスワードを保存しない （推奨）
7. *Authenticator* とオンラインサービスとの通信は *HTTPS* で行う （必須）

利用アプリを実装する際には以下のルールを守ること。

8. *Authenticator* が正規のものであることを確認してからアカウント処理を実施する （必須）

5.3.2.1 **Authenticator** を提供する **Service** は非公開 **Service** とする （必須）

Authenticator を提供する Service は Account Manager から利用されることを前提としており、他のアプリがアクセスできてはならない。非公開 Service とすることにより、他のアプリからのアクセスを排除することができる。また Account Manager は system 権限で動作しているので非公開 Service であってもアクセスできる。

5.3.2.2 ログイン画面 **Activity** は **Authenticator** アプリで実装する （必須）

新規アカウント追加および認証トークン再取得の場合に表示されるログイン画面は Authenticator アプリで実装すべきである。利用アプリ側で独自にログイン画面を用意してはならない。この記事の冒頭で「パスワードという極めてセンシティブな情報をアプリが扱わなくて済むことが Account Manager の利点である。」と述べた。もし利用アプリ側でログイン画面を用意してしまうと、利用アプリがパスワードを扱ってしまうことになり、Account Manager の思想から逸脱した設計となってしまう。

Authenticator アプリがログイン画面を用意することにより、ログイン画面を操作できるのは端末のユーザーだけに限定される。これは悪意あるアプリが直接ログインを試みたり、アカウントを作成したりといったアカウント攻撃をする手段がないということである。

5.3.2.3 ログイン画面 **Activity** は公開 **Activity** として他のアプリからの攻撃アクセスを想定する （必須）

ログイン画面 Activity は利用アプリの権限で起動する仕組みとなっている。利用アプリと Authenticator アプリの署名鍵が異なる場合にもログイン画面 Activity が表示されるためには、ログイン画面 Activity は公開 Activity として実装しなければならない。

ログイン画面 Activity が公開 Activity であるということは、悪意あるアプリからも起動される可能性があるということである。入力データは一切信用してはならない。したがって「3.2. 入力データの安全性を確認する」で述べたような対策が必要となる。

5.3.2.4 **KEY_INTENT** には、ログイン画面 **Activity** のクラス名を指定した明示的 **Intent** を与える （必須）

Authenticator がログイン画面 Activity を開きたいときには、Account Manager に返す Bundle の中にログイン画面 Activity を起動する Intent を KEY_INTENT で与えることになっている。ここで与える Intent はログイン画面 Activity をクラス名で指定する明示的 Intent でなければならない。暗黙的 Intent を与えた場合は、フレームワークが Authenticator アプリがログイン画面のために用意した Activity 以外の Activity の起動を試みる可能性がある。これによって、Andorid 4.4（API Level 19）以降のバージョンではアプリがクラッシュしたり、Android 4.4（API Level 19）より前のバージョンでは他のアプリの用意した意図しない Activity が起動したりする場合がある。

Android 4.4（API Level 19）以降のバージョンでは、フレームワークが KEY_INTENT で与える Intent で起動されるアプリの署名と Authenticator アプリの署名が一致しない場合には SecurityException を発生させるため、偽のログイン画面を起動される恐れはないが、正規のログイン画面を起動できずユーザーの正常なアプリ利用を妨げられるおそれがある。Android 4.4（API Level 19）より前のバージョンでは、悪意のあるアプリの用意した偽のログイン画面を起動され、ユーザーが悪意のあるアプリにパスワード等認証情報を入力してしまう危険がある。よって、いずれにバージョンであっても、KEY_INTENT で与える Intent は明示的 Intent でなければならない。

5.3.2.5 アカウント情報や認証トークンなどのセンシティブな情報はログ出力しない （必須）

オンラインサービスに接続するアプリは、その開発時だけでなく運用時においても、オンラインサービスにうまく接続できないトラブルに悩まされることがある。接続できない原因は多岐に渡り、ネットワーク環境の整備不足、通信プロト

コルの実装ミス、Permission 不足、認証エラーなど様々である。こうした原因の切り分けを目的として、プログラム内部で得られた情報をログ出力する実装もよくみられる。

パスワードや認証トークンなどのセンシティブな情報は決してログ出力してはならない。ログ情報は他のアプリからも読み取ることができるため情報漏洩の原因となりかねないからだ。アカウント名も漏洩も被害につながる場合にはログ出力してはならない。

5.3.2.6 Account Manager にパスワードを保存しない （推奨）

Account Manager に登録するアカウントには、パスワードと認証トークンの 2 つの認証情報を保存することができる。これらの情報は次のディレクトリの accounts.db の中に平文で（つまり暗号化されず）保存される。

- Android 4.1 以前/data/system/accounts.db
- Android 4.2 以降 Android 6.0 以前/data/system/users/0/accounts.db
- Android 7.0 以降/data/system_ce/0/accounts_ce.db

※ Android 4.2 以降はマルチユーザー機能がサポートされているため、ユーザーに合わせたディレクトリへ保存されるように変更されている。また、Android 7.0 以降では Direct Boot 対応のため、ロック時にデータを扱う際のデータベース /data/system_de/0/accounts_de_db とアンロック時にデータを扱う /data/system_ce/0/accounts_ce.db にデータベースファイルが分割された。認証情報は平文の状態で後者のデータベースファイルに保存される。

この accounts.db の内容を読み取るためには root 権限または system 権限が必要であり、市販の Android 端末では読み取ることができない。もし、攻撃者に root 権限や system 権限が奪われてしまう脆弱性が Android OS にある場合には、accounts.db の中に保存された認証情報が危険にさらされることになる。

この記事で紹介している Authenticator アプリは、Account Manager に認証トークンは保存するが、ユーザーのパスワードは保存しない設計としている。一定の期間以内にオンラインサービスに継続的に接続していれば、認証トークンの有効期間が延長されるのが一般的であるため、パスワードを保存しない設計で十分であることが多い。

認証トークンは一般にパスワードよりも有効期限が短く、いつでも無効化できる特徴がある、いわば使い捨ての認証情報である。万一、認証トークンが漏洩したとしても、認証トークンを無効化することができるため、認証トークンはパスワードに比べ安全性が高いとされている。認証トークンが無効化された場合には、ユーザーはもう一度パスワードを入力して新しい認証トークンを取得すればよい。

パスワードが漏洩した場合、パスワードを無効化してしまうと、そのユーザーはオンラインサービスを利用できなくなってしまう。このような場合、コールセンター対応等が必要となってしまうため大きなコストが発生する。ゆえに Account Manager にパスワードを保存する設計はできるだけ避けるべきである。どうしてもパスワードを保存する設計をしなければならない場合は、パスワードを暗号化して、暗号化の鍵を難読化するなど、高度なリバースエンジニアリング対策を実施することになる。

5.3.2.7 Authenticator とオンラインサービスとの通信は HTTPS で行う （必須）

パスワードや認証トークンはいわゆる認証情報といい、これを第三者に奪われてしまうと、第三者がユーザーになりすましできることになる。Authenticator はオンラインサービスとこうした認証情報を送受信することになるので、HTTPS 等の安全性の確立した暗号化通信方式で通信しなければならない。

5.3.2.8 **Authenticator** が正規のものであることを確認してからアカウント処理を実施する （必須）

端末に同一のアカウントタイプを定義した Authenticator が複数存在する場合、先にインストールされた Authenticator が有効になる。自分の Authenticator が後にインストールされた場合には利用されないということである。

もし先にインストールされた Authenticator がマルウェアによる偽装であった場合には、ユーザーが入力したアカウント情報がマルウェアに奪われてしまう恐れがある。利用アプリはアカウント操作を行うアカウントタイプについて、正規の Authenticator がそのアカウントタイプに割り当てられていることを確認してから、アカウント操作を実施しなければならない。

あるアカウントタイプに割り当てられている Authenticator が正規のものであるかは、その Authenticator を含むパッケージの証明書ハッシュ値を、事前に確認している正規の証明書ハッシュ値と一致するかどうかで確認できる。もし証明書ハッシュ値が一致しないことが判明した場合、そのアカウントタイプに割り当てられている意図しない Authenticator を含むパッケージをアンインストールするようユーザーを促すといった対処を施すことが望ましい。

5.3.3 アドバンスト

5.3.3.1 **Account Manager** の利用と **Permission**

AccountManager クラスの各メソッドを利用するためには、アプリの AndroidManifest.xml にそれぞれ適正な Permission の利用宣言をする必要がある。Android 5.1(API Level 22) 以前のバージョンでは AUTHENTICATE_ACCOUNTS や GET_ACCOUNTS, MANAGE_ACCOUNTS といった権限が必要であり、メソッドとの対応を 表 5.3.1 に示す。

表 5.3.1: Account Manager の機能と Permission

	Account Manager が提供する機能	
Permission	メソッド	説明
AUTHENTICATE_ACCOUNTS (Authenticator と同じ鍵で署名された Package のみ利用可能)	getPassword()	パスワードの取得
	getUserData()	利用者情報の取得
	addAccountExplicitly()	アカウントの DB への追加
	peekAuthToken()	キャッシュされたトークンの取得
	setAuthToken()	認証トークンの登録
	setPassword()	パスワードの変更
	setUserData()	利用者情報の設定
	renameAccount()	アカウント名の変更
GET_ACCOUNTS	getAccounts()	すべてのアカウントの一覧取得
	getAccountsByType()	アカウントタイプが同じアカウントの一覧取得
	getAccountsByTypeAndFeatures()	指定した機能を持ったアカウントの一覧取得
	addOnAccountsUpdatedListener()	イベントリスナーの登録
	hasFeatures()	指定した機能の有無
MANAGE_ACCOUNTS	getAuthTokenByFeatures()	指定した機能を持つアカウントの認証トークンの取得
	addAccount()	ユーザーへのアカウント追加要請
	removeAccount()	アカウントの削除
	clearPassword()	パスワードの初期化
	updateCredentials()	ユーザーへのパスワード変更要請

次のページに続く

表 5.3.1 – 前のページからの続き

	editProperties()	Authenticator の設定変更
	confirmCredentials()	ユーザーへのパスワード再入力要請
USE_CREDENTIALS	getAuthToken()	認証トークンの取得
	blockingGetAuthToken()	認証トークンの取得
MANAGE_ACCOUNTS または USE_CREDENTIALS	invalidateAuthToken()	キャッシュされたトークンの削除

ここで、AUTHENTICATE_ACCOUNTS Permission が必要なメソッド群を使う場合には Permission に加えてパッケージの署名鍵に関する制限が設けられている。具体的には、Authenticator を提供するパッケージの署名に使う鍵とメソッドを使うアプリのパッケージの署名に使う鍵が同じでなければならない。そのため、Authenticator 以外に AUTHENTICATE_ACCOUNTS Permission が必要なメソッド群を使うアプリを配布する際には、Authenticator と同じ鍵で署名を施すことになる。

Android 6.0(API Level 23) 以降のバージョンでは GET_ACCOUNTS 以外の Permission は使用されておらず、宣言してもしなくてもできることに差はない。Android 5.1(API Level 22) 以前のバージョンにおいて AUTHENTICATE_ACCOUNTS を要求していたメソッドについては、Permission を要求しないものの、同様に署名が一致する場合のみしか呼び出せない (署名が一致しない場合には SecurityException が発生する) ことに注意する。

さらに、Android 8.0（API Level 26）で GET_ACCOUNTS を必要としていた API のアクセス制御も変更された。Android 8.0（API Level 26）以降のバージョンで、アカウント情報の利用側のアプリの targetSdkVersion が 26 以上の場合には、GET_ACCOUNTS が付与されていたとしても原則として Authenticator アプリと署名が一致する場合にしかアカウントの情報は取得できない。ただし、Authenticator アプリは setAccountVisibility メソッドを呼び出してパッケージ名を指定することで、署名の一致しないアプリに対してもアカウント情報を提供することができる。

Android Studio での開発の際には設定した署名鍵が固定で使われるため、鍵のことを意識せずに Permission だけで実装や動作確認が出来てしまう。特にアプリによって署名鍵を使い分けている開発者は、この制限を考慮してアプリに使う鍵を選定する必要があるので注意をすること。また、Account Manager から取得するデータにはセンシティブな情報が含まれるため、漏洩や不正利用などのリスクを減らすように扱いには十分注意すること。

5.3.3.2 Android 4.0.x では利用アプリと Authenticator アプリの署名鍵が異なると例外が発生する

Authenticator を含む Authenticator アプリと異なる開発者鍵で署名された利用アプリから認証トークンの取得機能が要求された場合、Account Manager は認証トークン使用許諾画面（GrantCredentialsPermissionActivity）を表示してユーザーに認証トークンの使用可否を確認する。しかし、Android 4.0.x の Android Framework には不具合があり、Account Manager によってこの画面が開かれた途端、例外が発生し、アプリが強制終了してしまう（図 5.3.3）。不具合の詳細は https://code.google.com/p/android/issues/detail?id=23421 に記載されている。Android 4.1.x 以降ではこの不具合はない。

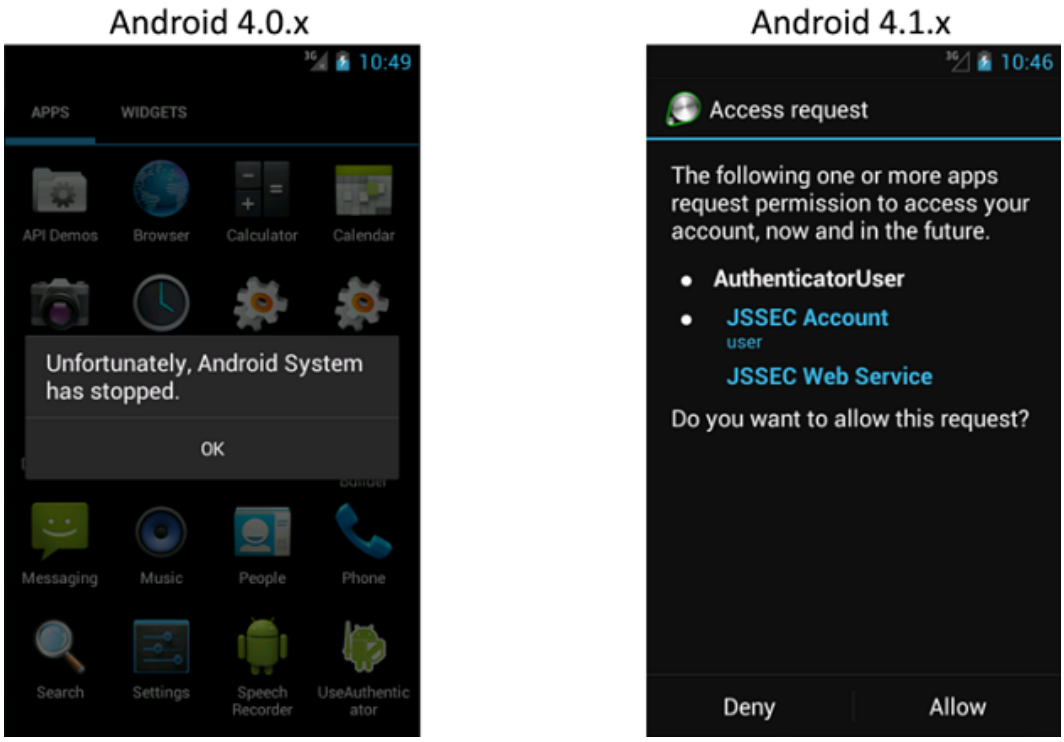

図 5.3.3　Android 標準の認証トークン使用許諾画面を表示した場合

5.3.3.3 Android 8.0（API Level 26）以降で署名の一致しない Authenticator のアカウントを読めるケース

Android 8.0（API Level 26）から、アカウント情報の取得など Android 7.1（API Level 25）以前では GET_ACCOUNTS Permission が必要であったメソッドの呼び出しに対して GET_ACCOUNTS Permission が不要になり、代わりに署名が一致する場合や Authenticator アプリ側で setAccountVisibility メソッドによってアカウント情報の提供先アプリとして指定された場合にのみアカウントの情報が取得できるようになった。ただし、フレームワークの課すこのルールにはいくつかの例外があり、注意が必要である。その例外について以下に述べる。

まず、アカウント情報利用側アプリの targetSdkVersion が 25 (Android 7.1) 以下の場合には、上記のルールが適用されず、GET_ACCOUNTS Permission を持っているアプリは署名に関係なく端末内のアカウント情報を取得できる。ただし、この挙動は Authenticator 側の実装によって変更することができることを後に述べる。

次に、WRITE_CONTACTS Permission を利用宣言している Authenticator のアカウント情報は、READ_CONTACTS Permission を持っている他アプリから、署名に関係なく読めてしまう。これは、バグではなくフレームワークの仕様である[*8]。ただし、この挙動もまた、Authenticator 側の実装によって変更することができる。

以上のように署名が一致しておらず、かつ setAccountVisibility メソッドの呼び出しによってアカウント情報の提供先に指定していないアプリにもアカウント情報を読まれてしまう例外的なケースはあるのだが、これらの挙動は Authenticator 側で予め次のスニペットのように setAccountVisibility メソッドを呼び出しておくことで変更できる。

第三者アプリにアカウント情報を提供しない

```
accountManager.setAccountVisibility(account, // visibility を変更するアカウント
        AccountManager.PACKAGE_NAME_KEY_LEGACY_VISIBLE,
        AccountManager.VISIBILITY_USER_MANAGED_NOT_VISIBLE);
```

[*8] WRITE_CONTACTS Permission を利用宣言している Authenticator はアカウント情報を ContactsProvider に書き込むとの想定で、READ_CONTACTS Permission を持つアプリにアカウント情報取得を許可していると考えられる。

この通りに setAccountVisibility メソッドを呼び出した Authenticator のアカウント情報については、フレームワークはデフォルトの挙動ではなく、targetSdkVersion <= 25 のケースや READ_CONTACTS を持っている場合であってもアカウント情報を提供しないように挙動を変更する。

5.4 HTTPS で通信する

スマートフォンアプリはインターネット上の Web サーバーと通信するものが多い。その通信方式として当ガイドでは HTTP と HTTPS の 2 方式に着目する。この 2 方式のうち、セキュリティの観点では HTTPS による通信が望ましい。近年 Google や Facebook など大手の Web サービスは HTTPS による接続を基本とするように変わってきた。ただし、HTTPS による接続の中でも SSLv3 を用いた接続に関しては脆弱性の存在 (通称 POODLE) が知られており、極力使用しないことを推奨する[*9]。

2012 年以降 Android アプリの HTTPS 通信の実装方法における欠陥が多く指摘されている。これは信頼できる第三者認証局から発行されたサーバー証明書ではなく、私的に発行されたサーバー証明書（以降、プライベート証明書と呼ぶ）により運用されているテスト用 Web サーバーに接続するために実装された欠陥であると推察される。

この記事では、HTTP および HTTPS 通信の方法について説明する。HTTPS 通信の方法には、プライベート証明書で運用されている Web サーバーに安全に接続する方法も含む。

5.4.1 サンプルコード

開発しているアプリの通信処理の特性を踏まえ、図 5.4.1 に従いサンプルコードを選択すること。

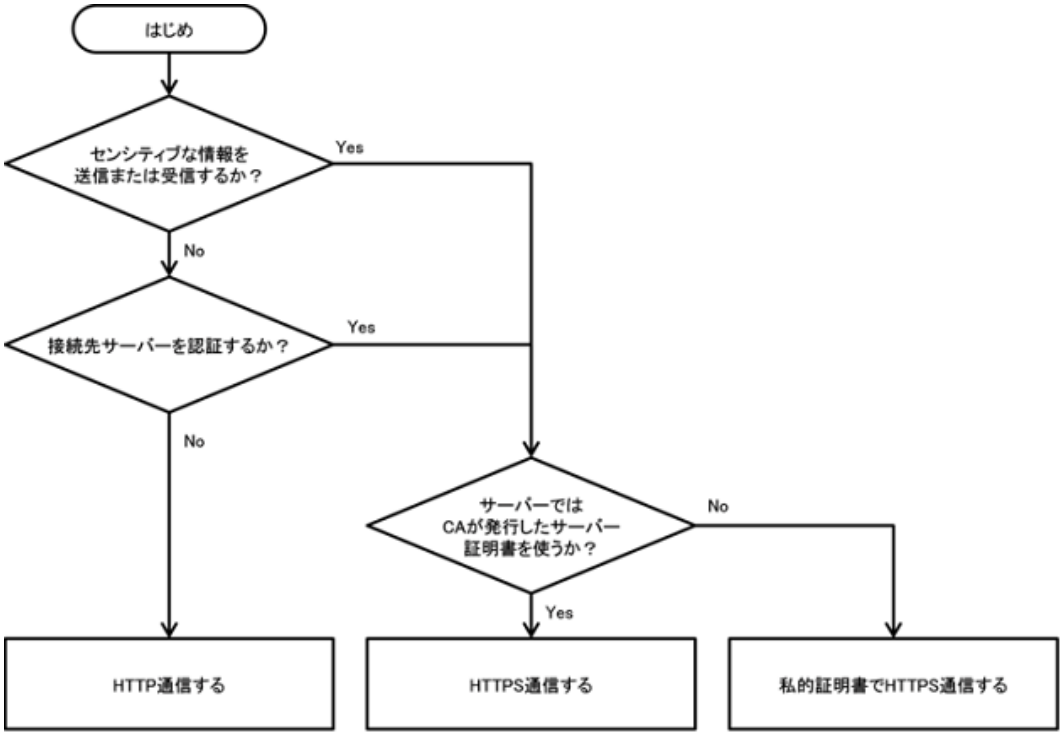

図 5.4.1　HTTP/HTTPS のサンプルコードを選択するフローチャート

センシティブな情報を送受信する場合は SSL/TLS で通信経路が暗号化される HTTPS 通信を用いる。HTTPS 通信を用いたほうが良いセンシティブな情報としては以下のようなものがある。

[*9] Android 8.0(API Level 26) 以降ではプラットフォームレベルで SSLv3 を用いた接続が非サポートになっている。

- Web サービスへのログイン ID、パスワード
- 認証状態を維持するための情報（セッション ID、トークン、Cookie の情報など）
- その他 Web サービスの特性に応じた重要情報・秘密情報（個人情報やクレジットカード情報など）

ここで対象となっているスマートフォンアプリは、サーバーと通信を行うことで連携し、構築されるシステムの一部を担っている。従って、通信のどの部分を HTTP もしくは HTTPS とするのかについては、システム全体を考慮して適切なセキュア設計・セキュアコーディングを施すこと。HTTP と HTTPS の通信方式の違いは 表 5.4.1 を参考にすること。またサンプルコードの違いについては 表 5.4.2 を参考にすること。

表 5.4.1: HTTP 通信方式、HTTPS 通信方式の比較

		HTTP	**HTTPS**
特徴	URL	http://で始まる	https://で始まる
	通信内容の暗号化	なし	あり
	通信内容の改ざん検知	不可	可
	接続先サーバーの認証	不可	可
被害リスク	攻撃者による通信内容の読み取り	高	低
	攻撃者による通信内容の書き換え	高	低
	アプリの偽サーバーへの接続	高	低

表 5.4.2　HTTP/HTTPS 通信のサンプルコードの説明

サンプルコード	通信	センシティブな情報の送受信	サーバー証明書
HTTP 通信する	HTTP	X	-
HTTPS 通信する	HTTPS	o	Cybertrust や VeriSign 等の第三者認証局により発行されたサーバー証明書
プライベート証明書で HTTPS 通信する	HTTPS	o	プライベート証明書※イントラサーバーやテストサーバーで良くみられる運用形態

なお Android がサポートし現在広く使われている HTTP/HTTPS 通信用 API は、Java SDK 由来の java.net.HttpURLConnection/javax.net.ssl.HttpsURLConnection である。Apache HTTPComponent 由来の Apache HttpClient ライブラリについては、Android 6.0（API Level 23) でサポートが打ち切られている。

5.4.1.1 HTTP 通信する

HTTP 通信で送受信する情報はすべて攻撃者に盗聴・改ざんされる可能性があることを前提としなければならない。また接続先サーバーも攻撃者が用意した偽物のサーバーに接続することがあることも前提としなければならない。このような前提においても被害が生じない、または許容範囲に収まる用途のアプリにおいてのみ、HTTP 通信を利用できる。こうした前提を受け入れられないアプリについては「5.4.1.2. *HTTPS* 通信する」や「5.4.1.3. プライベート証明書で *HTTPS* 通信する」を参照すること。以下のサンプルコードは、Web サーバー上で画像検索を行い、検索画像を取得して表示するアプリである。1 回の検索でサーバーと HTTP 通信を 2 回行う。1 回目の通信で画像検索を実施し、2 回目の通信で画像を取得する。UI スレッドでの通信を避けるために、AsyncTask を利用して通信処理用のワーカースレッドを作成している。Web サーバーとの通信で送受信する情報は、画像の検索文字列、画像の URL、画像データだが、どれもセンシティブな情報はないとみなしている。そのため、受信データである画像の URL と画像データは、攻撃者が用意した攻撃用のデータである可能性がある。簡単のため、サンプルコードでは受信データが攻撃データであっても許容されるとして対

策を施していない。同様の理由により、JSON パース時や画像データを表示する時に発生する可能性のある例外に対する例外処理を省略している。アプリの仕様に応じて適切に処理を実装する必要があることに注意すること[*10]。

ポイント：

1. 送信データにセンシティブな情報を含めない
2. 受信データが攻撃者からの送信データである場合を想定する

HttpImageSearch.java

```
package org.jssec.android.https.imagesearch;

import android.os.AsyncTask;

import org.json.JSONException;
import org.json.JSONObject;

import java.io.BufferedInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.net.HttpURLConnection;
import java.net.URL;

public abstract class HttpImageSearch extends AsyncTask<String, Void, Object> {

    @Override
    protected Object doInBackground(String... params) {
        byte[] responseArray;
        // ---------------------------------------------------------
        // 通信 1 回目：画像検索する
        // ---------------------------------------------------------

        // ★ポイント 1★ 送信データにセンシティブな情報を含めない
        // 画像検索文字列を送信する
        StringBuilder s = new StringBuilder();
        for (String param : params){
            s.append(param);

            s.append('+');
        }
        s.deleteCharAt(s.length() - 1);

        String search_url = "http://ajax.googleapis.com/ajax/services/search/images?v=1.0&q=" +
                s.toString();

        responseArray = getByteArray(search_url);
        if (responseArray == null) {
            return null;
        }

        // ★ポイント 2★ 受信データが攻撃者からの送信データである場合を想定する
        // サンプルにつき検索結果が攻撃者からのデータである場合の処理は割愛
        // サンプルにつき JSON パース時の例外処理は割愛
        String image_url;
```

(continues on next page)

[*10] 本サンプルコード内で画像検索 API として利用している Google Image Search API は 2016 年 2 月 15 日をもって正式にサービス提供を終了している。そのため、サンプルコードをそのまま動作させるには同等のサービスに置き換える必要がある。

(continued from previous page)

```
        try {
            String json = new String(responseArray);
            image_url = new JSONObject(json).getJSONObject("responseData")
                    .getJSONArray("results").getJSONObject(0).getString("url");
        } catch(JSONException e) {
            return e;
        }

        // ------------------------------------------------------
        // 通信２回目：画像を取得する
        // ------------------------------------------------------
        // ★ポイント１★ 送信データにセンシティブな情報を含めない
        if (image_url != null ) {
            responseArray = getByteArray(image_url);
            if (responseArray == null) {
                return null;
            }
        }

        // ★ポイント２★ 受信データが攻撃者からの送信データである場合を想定する
        return responseArray;
    }

    private byte[] getByteArray(String strUrl) {
        byte[] buff = new byte[1024];
        byte[] result = null;
        HttpURLConnection response;
        BufferedInputStream inputStream = null;
        ByteArrayOutputStream responseArray = null;
        int length;

        try {
            URL url = new URL(strUrl);
            response = (HttpURLConnection) url.openConnection();
            response.setRequestMethod("GET");
            response.connect();
            checkResponse(response);

            inputStream = new BufferedInputStream(response.getInputStream());
            responseArray = new ByteArrayOutputStream();

            while ((length = inputStream.read(buff)) != -1) {
                if (length > 0) {
                    responseArray.write(buff, 0, length);
                }
            }
            result = responseArray.toByteArray();
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (inputStream != null) {
                try {
                    inputStream.close();
                } catch (IOException e) {
                    // 例外処理は割愛
```

(continues on next page)

(continued from previous page)

```
                }
            }
            if (responseArray != null) {
                try {
                    responseArray.close();
                } catch (IOException e) {
                    // 例外処理は割愛
                }
            }
        }
        return result;
    }

    private void checkResponse(HttpURLConnection response) throws IOException {

        int statusCode = response.getResponseCode();
        if (HttpURLConnection.HTTP_OK != statusCode) {
            throw new IOException("HttpStatus: " + statusCode);
        }
    }
}
```

ImageSearchActivity.java

```
package org.jssec.android.https.imagesearch;

import android.app.Activity;
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.os.AsyncTask;
import android.os.Bundle;
import android.view.View;
import android.widget.EditText;
import android.widget.ImageView;
import android.widget.TextView;

public class ImageSearchActivity extends Activity {

    private EditText mQueryBox;
    private TextView mMsgBox;
    private ImageView mImgBox;
    private AsyncTask<String, Void, Object> mAsyncTask ;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        mQueryBox = (EditText)findViewById(R.id.querybox);
        mMsgBox = (TextView)findViewById(R.id.msgbox);
        mImgBox = (ImageView)findViewById(R.id.imageview);
    }

    @Override
    protected void onPause() {
```

(continues on next page)

(continued from previous page)

```
        // このあと Activity が破棄される可能性があるので非同期処理をキャンセルしておく
        if (mAsyncTask != null) mAsyncTask.cancel(true);
        super.onPause();
    }

    public void onHttpSearchClick(View view) {
        String query = mQueryBox.getText().toString();
        mMsgBox.setText("HTTP:" + query);
        mImgBox.setImageBitmap(null);

        // 直前の非同期処理が終わってないこともあるのでキャンセルしておく
        if (mAsyncTask != null) mAsyncTask.cancel(true);

        // UI スレッドで通信してはならないので、AsyncTask によりワーカースレッドで通信する
        mAsyncTask = new HttpImageSearch() {
            @Override
            protected void onPostExecute(Object result) {
                // UI スレッドで通信結果を処理する
                if (result == null) {
                    mMsgBox.append("\n 例外発生\n");
                } else if (result instanceof Exception) {
                    Exception e = (Exception)result;
                    mMsgBox.append("\n 例外発生\n" + e.toString());
                } else {
                    // サンプルにつき画像表示の際の例外処理は割愛
                    byte[] data = (byte[])result;
                    Bitmap bmp = BitmapFactory.decodeByteArray(data, 0, data.length);
                    mImgBox.setImageBitmap(bmp);
                }
            }
                }.execute(query);    // 検索文字列を渡して非同期処理を開始
    }

    public void onHttpsSearchClick(View view) {
        String query = mQueryBox.getText().toString();
        mMsgBox.setText("HTTPS:" + query);
        mImgBox.setImageBitmap(null);

        // 直前の非同期処理が終わってないこともあるのでキャンセルしておく
        if (mAsyncTask != null) mAsyncTask.cancel(true);

        // UI スレッドで通信してはならないので、AsyncTask によりワーカースレッドで通信する
        mAsyncTask = new HttpsImageSearch() {
            @Override
            protected void onPostExecute(Object result) {
                // UI スレッドで通信結果を処理する
                if (result instanceof Exception) {
                    Exception e = (Exception)result;
                    mMsgBox.append("\n 例外発生\n" + e.toString());
                } else {
                    byte[] data = (byte[])result;
                    Bitmap bmp = BitmapFactory.decodeByteArray(data, 0, data.length);
                    mImgBox.setImageBitmap(bmp);
                }
            }
```

(continues on next page)

(continued from previous page)

```
                }.execute(query);    // 検索文字列を渡して非同期処理を開始
    }
}
```

```
AndroidManifest.xml
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="org.jssec.android.https.imagesearch"
    android:versionCode="1"
    android:versionName="1.0">

    <uses-permission android:name="android.permission.INTERNET"/>

    <application
        android:icon="@drawable/ic_launcher"
        android:allowBackup="false"
        android:label="@string/app_name" >
        <activity
            android:name=".ImageSearchActivity"
            android:label="@string/app_name"
            android:theme="@android:style/Theme.Light"
            android:exported="true" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

5.4.1.2 HTTPS 通信する

HTTPS 通信では送受信するデータが暗号化されるだけでなく、接続先サーバーが本物かどうかの検証も行われる。そのために HTTPS 通信開始時のハンドシェイク処理において、サーバーから送られてくるサーバー証明書に対して、Android の HTTPS ライブラリ内部で次のような観点で検証が行われる。

- 第三者認証局により署名されたサーバー証明書であること
- サーバー証明書の期限等が有効であること
- サーバー証明書の Subject の CN（Common Name）または SAN（Subject Alternname Names）の DNS 名が接続先サーバーのホスト名と一致していること

これらの検証に失敗するとサーバー証明書検証エラー（SSLException）が発生する。サーバー証明書に不備がある場合、もしくは攻撃者が中間者攻撃[*11] をしている場合にこのエラーが発生する。エラーが発生した場合には、アプリの仕様に応じて適切な処理を実行する必要がある点に注意すること。

ここでは第三者認証局から発行されたサーバー証明書で運用されている Web サーバーに接続する HTTPS 通信のサンプルコードを示す。第三者認証局から発行されたサーバー証明書ではなく、私的に発行したサーバー証明書で HTTPS 通信を実現したい場合には「5.4.1.3. プライベート証明書で *HTTPS* 通信する」を参照すること。

以下のサンプルコードは、Web サーバー上で画像検索を行い、検索画像を取得して表示するアプリである。1 回の検索で

[*11] 中間者攻撃については次のページを参照。https://www.ipa.go.jp/about/press/20140919_1.html

サーバーと HTTPS 通信を 2 回行う。1 回目の通信で画像検索を実施し、2 回目の通信で画像を取得する。UI スレッドでの通信を避けるために、AsyncTask を利用して通信処理用のワーカースレッドを作成している。Web サーバーとの通信で送受信する情報は、画像の検索文字列、画像の URL、画像データで、全てセンシティブな情報とみなしている。なお、簡単のため、SSLException に対してはユーザーへの通知などの例外処理を行っていないが、アプリの仕様に応じて適切な処理を実装する必要がある[*12]。また、以下のサンプルコードでは SSLv3 を用いた通信が許容されている[*13]。SSLv3 の脆弱性（通称 POODLE）に対する攻撃を回避するためには、接続先サーバーにおいて SSLv3 を無効化する設定を施すことをお勧めする。

RFC2818 の記載によると、サーバー証明書の検証において既存の慣行である CN の使用は非推奨であり、ドメイン名と証明書を照合するためには SAN を使用することが強く推奨されている。そのため Android 9.0(API Level 28) では検証に SAN のみを用いるように変更され、サーバー側は SAN を含む証明書を提示する必要があり、含まれない証明書は信頼されなくなる。

ポイント：

1. URI は https://で始める
2. 送信データにセンシティブな情報を含めてよい
3. HTTPS 接続したサーバーからのデータであっても、受信データの安全性を確認する
4. SSLException に対してアプリに適した例外処理を行う

HttpsImageSearch.java

```
package org.jssec.android.https.imagesearch;

import org.json.JSONException;
import org.json.JSONObject;

import android.os.AsyncTask;

import java.io.BufferedInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.net.HttpURLConnection;
import java.net.URL;

public abstract class HttpsImageSearch extends AsyncTask<String, Void, Object> {

    @Override
    protected Object doInBackground(String... params) {
        byte[] responseArray;
        // ---------------------------------------------------------
        // 通信 1 回目：画像検索する
        // ---------------------------------------------------------

        // ★ポイント 1★ URI は https://で始める
        // ★ポイント 2★ 送信データにセンシティブな情報を含めてよい
        StringBuilder s = new StringBuilder();
        for (String param : params){
```

(continues on next page)

[*12] 本サンプルコード内で画像検索 API として利用している Google Image Search API は 2016 年 2 月 15 日をもって正式にサービス提供を終了している。そのため、サンプルコードをそのまま動作させるには同等のサービスに置き換える必要がある。

[*13] Android 8.0(API Level 26) 以降ではプラットフォームレベルで禁止しているため SSLv3 での接続は起こらないが、サーバー側での SSLv3 無効化対策は行うことをお勧めする。

(continued from previous page)

```
            s.append(param);
            s.append('+');
        }
        s.deleteCharAt(s.length() - 1);

        String search_url = "https://ajax.googleapis.com/ajax/services/search/images?v=1.0&q=" +
                s.toString();

        responseArray = getByteArray(search_url);
        if (responseArray == null) {
            return null;
        }

        // ★ポイント 3★ HTTPS 接続したサーバーからのデータであっても、受信データの安全性を確認する
        // サンプルにつき割愛。「3.2 入力データの安全性を確認する」を参照。
        String image_url;
        try {
            String json = new String(responseArray);
            image_url = new JSONObject(json).getJSONObject("responseData")
                    .getJSONArray("results").getJSONObject(0).getString("url");
        } catch(JSONException e) {
            return e;

        }

        // --------------------------------------------------------
        // 通信 2回目：画像を取得する
        // --------------------------------------------------------

        // ★ポイント 1★ URI は https://で始める
        // ★ポイント 2★ 送信データにセンシティブな情報を含めてよい
        if (image_url != null ) {
            responseArray = getByteArray(image_url);
            if (responseArray == null) {
                return null;
            }
        }

        return responseArray;
    }

    private byte[] getByteArray(String strUrl) {
        byte[] buff = new byte[1024];
        byte[] result = null;
        HttpURLConnection response;
        BufferedInputStream inputStream = null;
        ByteArrayOutputStream responseArray = null;
        int length;

        try {
            URL url = new URL(strUrl);
            response = (HttpURLConnection) url.openConnection();
            response.setRequestMethod("GET");
            response.connect();
            checkResponse(response);
```

(continues on next page)

(continued from previous page)

```
            inputStream = new BufferedInputStream(response.getInputStream());
            responseArray = new ByteArrayOutputStream();

            while ((length = inputStream.read(buff)) != -1) {
                if (length > 0) {
                    responseArray.write(buff, 0, length);
                }
            }
            result = responseArray.toByteArray();
        } catch(SSLException e) {
            // ★ ポイント 4 ★ SSLException に対してアプリに適した例外処理を行う
            // サンプルにつき例外処理は割愛
            return e;
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (inputStream != null) {
                try {
                    inputStream.close();
                } catch (IOException e) {
                    // 例外処理は割愛
                }
            }
            if (responseArray != null) {
                try {
                    responseArray.close();
                } catch (IOException e) {
                    // 例外処理は割愛
                }
            }
        }
        return result;
    }

    private void checkResponse(HttpURLConnection response) throws IOException {
        int statusCode = response.getResponseCode();
        if (HttpURLConnection.HTTP_OK != statusCode) {
            throw new IOException("HttpStatus: " + statusCode);
        }
    }
}
```

サンプルコードの他のファイルについては「5.4.1.1. *HTTP* 通信する」と共用しているので「5.4.1.1. *HTTP* 通信する」も参照すること。

5.4.1.3 プライベート証明書で HTTPS 通信する

ここでは第三者認証局から発行されたサーバー証明書ではなく、私的に発行したサーバー証明書（プライベート証明書）で HTTPS 通信をするサンプルコードを示す。プライベート認証局のルート証明書とプライベート証明書の作成方法および Web サーバーの HTTPS 設定については「5.4.3.1. プライベート証明書の作成方法とサーバー設定」を参考にすること。またサンプルプログラムの assets 中の cacert.crt ファイルはプライベート認証局のルート証明書ファイルである。

以下のサンプルコードは、Web サーバー上の画像を取得して表示するアプリである。Web サーバーとは HTTPS を用い

た通信を行う。UI スレッドでの通信を避けるために、AsyncTask を利用して通信処理用のワーカースレッドを作成している。Web サーバーとの通信で送受信する情報は画像の URL と画像データで、このサンプルではどちらもセンシティブな情報とみなしている。また、簡単のため、SSLException に対してはユーザーへの通知などの例外処理を行っていないが、アプリの仕様に応じて適切な処理を実装する必要がある。

ポイント：

1. プライベート認証局のルート証明書でサーバー証明書を検証する
2. URI は https://で始める
3. 送信データにセンシティブな情報を含めてよい
4. 受信データを接続先サーバーと同じ程度に信用してよい
5. SSLException に対しユーザーに通知する等の適切な例外処理をする

PrivateCertificateHttpsGet.java

```
package org.jssec.android.https.privatecertificate;

import java.io.BufferedInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.net.HttpURLConnection;
import java.net.URL;
import java.security.KeyStore;
import java.security.SecureRandom;

import javax.net.ssl.HostnameVerifier;
import javax.net.ssl.HttpsURLConnection;
import javax.net.ssl.SSLContext;
import javax.net.ssl.SSLException;
import javax.net.ssl.SSLSession;
import javax.net.ssl.TrustManagerFactory;

import android.content.Context;
import android.os.AsyncTask;

public abstract class PrivateCertificateHttpsGet extends AsyncTask<String, Void, Object> {

    private Context mContext;

    public PrivateCertificateHttpsGet(Context context) {
        mContext = context;
    }

    @Override
    protected Object doInBackground(String... params) {
        TrustManagerFactory trustManager;
        BufferedInputStream inputStream = null;
        ByteArrayOutputStream responseArray = null;
        byte[] buff = new byte[1024];
        int length;

        try {
            URL url = new URL(params[0]);
```

(continues on next page)

(continued from previous page)

```
        // ★ポイント 1★ プライベート証明書でサーバー証明書を検証する
        // assets に格納しておいたプライベート証明書だけを含む KeyStore を設定
        KeyStore ks = KeyStoreUtil.getEmptyKeyStore();
        KeyStoreUtil.loadX509Certificate(ks,
                mContext.getResources().getAssets().open("cacert.crt"));

        // ホスト名の検証を行う
        HttpsURLConnection.setDefaultHostnameVerifier(new HostnameVerifier() {
            @Override
            public boolean verify(String hostname, SSLSession session) {
                if (!hostname.equals(session.getPeerHost())) {
                    return false;
                }
                return true;
            }
        });


        // ★ポイント 2★ URI は https://で始める
        // ★ポイント 3★ 送信データにセンシティブな情報を含めてよい
        trustManager = TrustManagerFactory.getInstance(TrustManagerFactory.getDefaultAlgorithm());
        trustManager.init(ks);
        SSLContext sslCon = SSLContext.getInstance("TLS");
        sslCon.init(null, trustManager.getTrustManagers(), new SecureRandom());

        HttpURLConnection con = (HttpURLConnection)url.openConnection();

        HttpsURLConnection response = (HttpsURLConnection)con;
        response.setDefaultSSLSocketFactory(sslCon.getSocketFactory());

        response.setSSLSocketFactory(sslCon.getSocketFactory());
        checkResponse(response);

        // ★ポイント 4★ 受信データを接続先サーバーと同じ程度に信用してよい
        inputStream = new BufferedInputStream(response.getInputStream());
        responseArray = new ByteArrayOutputStream();
        while ((length = inputStream.read(buff)) != -1) {
            if (length > 0) {
                responseArray.write(buff, 0, length);
            }
        }
        return responseArray.toByteArray();
    } catch(SSLException e) {
        // ★ポイント 5★ SSLException に対しユーザーに通知する等の適切な例外処理をする
        // サンプルにつき例外処理は割愛
        return e;
    } catch(Exception e) {
        return e;
    } finally {
        if (inputStream != null) {
            try {
                inputStream.close();
            } catch (Exception e) {
                // 例外処理は割愛
            }
```

(continues on next page)

(continued from previous page)

```
            }
            if (responseArray != null) {
                try {
                    responseArray.close();
                } catch (Exception e) {
                    // 例外処理は割愛
                }
            }
        }
    }

    private void checkResponse(HttpURLConnection response) throws IOException {
        int statusCode = response.getResponseCode();
        if (HttpURLConnection.HTTP_OK != statusCode) {
            throw new IOException("HttpStatus: " + statusCode);
        }
    }
}
```

KeyStoreUtil.java

```
package org.jssec.android.https.privatecertificate;

import java.io.IOException;
import java.io.InputStream;
import java.security.KeyStore;
import java.security.KeyStoreException;
import java.security.NoSuchAlgorithmException;
import java.security.cert.Certificate;
import java.security.cert.CertificateException;
import java.security.cert.CertificateFactory;
import java.security.cert.X509Certificate;
import java.util.Enumeration;

public class KeyStoreUtil {
    public static KeyStore getEmptyKeyStore() throws KeyStoreException,
            NoSuchAlgorithmException, CertificateException, IOException {
        KeyStore ks = KeyStore.getInstance("BKS");
        ks.load(null);
        return ks;
    }

    public static void loadAndroidCAStore(KeyStore ks)
            throws KeyStoreException, NoSuchAlgorithmException,
            CertificateException, IOException {
        KeyStore aks = KeyStore.getInstance("AndroidCAStore");
        aks.load(null);
        Enumeration<String> aliases = aks.aliases();
        while (aliases.hasMoreElements()) {
            String alias = aliases.nextElement();
            Certificate cert = aks.getCertificate(alias);
            ks.setCertificateEntry(alias, cert);
        }
    }
```

(continues on next page)

(continued from previous page)

```
    public static void loadX509Certificate(KeyStore ks, InputStream is)
            throws CertificateException, KeyStoreException {
        try {
            CertificateFactory factory = CertificateFactory.getInstance("X509");
            X509Certificate x509 = (X509Certificate)factory.generateCertificate(is);
            String alias = x509.getSubjectDN().getName();
            ks.setCertificateEntry(alias, x509);
        } finally {
            try { is.close(); } catch (IOException e) { /* 例外処理は割愛 */ }
        }
    }
}
```

PrivateCertificateHttpsActivity.java

```
package org.jssec.android.https.privatecertificate;

import android.app.Activity;
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.os.AsyncTask;
import android.os.Bundle;
import android.view.View;
import android.widget.EditText;
import android.widget.ImageView;
import android.widget.TextView;

public class PrivateCertificateHttpsActivity extends Activity {

    private EditText mUrlBox;
    private TextView mMsgBox;
    private ImageView mImgBox;
    private AsyncTask<String, Void, Object> mAsyncTask ;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        mUrlBox = (EditText)findViewById(R.id.urlbox);
        mMsgBox = (TextView)findViewById(R.id.msgbox);
        mImgBox = (ImageView)findViewById(R.id.imageview);
    }

    @Override
    protected void onPause() {
        // このあと Activity が破棄される可能性があるので非同期処理をキャンセルしておく
        if (mAsyncTask != null) mAsyncTask.cancel(true);
        super.onPause();
    }

    public void onClick(View view) {
        String url = mUrlBox.getText().toString();
        mMsgBox.setText(url);
        mImgBox.setImageBitmap(null);
```

(continues on next page)

(continued from previous page)

```
        // 直前の非同期処理が終わってないこともあるのでキャンセルしておく
        if (mAsyncTask != null) mAsyncTask.cancel(true);

        // UI スレッドで通信してはならないので、AsyncTask によりワーカースレッドで通信する
        mAsyncTask = new PrivateCertificateHttpsGet(this) {
            @Override
            protected void onPostExecute(Object result) {
                // UI スレッドで通信結果を処理する
                if (result instanceof Exception) {
                    Exception e = (Exception)result;
                    mMsgBox.append("\n 例外発生\n" + e.toString());
                } else {
                    byte[] data = (byte[])result;
                    Bitmap bmp = BitmapFactory.decodeByteArray(data, 0, data.length);
                    mImgBox.setImageBitmap(bmp);
                }
            }
        }.execute(url); // URL を渡して非同期処理を開始
    }
}
```

5.4.2 ルールブック

HTTP 通信、HTTPS 通信する場合には以下のルールを守ること。

1. センシティブな情報は *HTTPS* 通信で送受信する （必須）
2. *HTTP* 通信では受信データの安全性を確認する （必須）
3. *SSLException* に対しユーザーに通知する等の適切な例外処理をする （必須）
4. 独自の *TrustManager* を作らない （必須）
5. 独自の *HostnameVerifier* は作らない （必須）

5.4.2.1 センシティブな情報は HTTPS 通信で送受信する （必須）

HTTP を使った通信では、送受信する情報の盗聴・改ざん、または、接続先サーバーのなりすましが起こる可能性がある。センシティブな情報は HTTPS 通信で送受信すること。

5.4.2.2 HTTP 通信では受信データの安全性を確認する （必須）

HTTP 通信における受信データは攻撃者が制御可能であるため、コード脆弱性を狙った攻撃データを受信する可能性がある。あらゆる値、形式のデータを受信することを想定して、受信データを処理するコードに脆弱性がないように気を付けてコーディングする必要がある。また、HTTPS 通信における受信データについても、受信データを無条件に安全であると考えてはならない。HTTPS 接続先のサーバーが攻撃者によって用意されたものである場合や、受信データが接続先サーバーとは別の場所で生成されたデータである場合もあるためである。「3.2. 入力データの安全性を確認する」も参照すること。

5.4.2.3 **SSLException** に対しユーザーに通知する等の適切な例外処理をする （必須）

HTTPS 通信ではサーバー証明書の検証時に SSLException が発生することがある。SSLException はサーバー証明書の不備が原因となって発生する。証明書の不備は攻撃者による中間者攻撃によって発生している可能性があるので、SSLException に対しては適切な例外処理を実装することが必要である。例外処理の例としては、SSLException による通信失敗をユーザーに通知すること、あるいはログに記録することが考えられる。その一方で、アプリによってはユーザーに対する特別な通知は必要とされないこともありうる。このように、実装すべき処理はアプリの仕様や特性によって異なるので、それらを十分に検討した上で決定しなければならない。

加えて、SSLException が発生した場合には中間者攻撃を受けている可能性があるので、HTTP などの非暗号化通信によってセンシティブな情報の送受信を再度試みるような実装してはならない。

5.4.2.4 独自の **TrustManager** を作らない （必須）

自己署名証明書などのプライベート証明書で HTTPS 通信するためには、サーバー証明書検証に使う KeyStore を変更するだけで済む。しかしながら「5.4.3.3. 証明書検証を無効化する危険なコード」で説明しているように、インターネット上で公開されているサンプルコードには、危険な TrustManager を実装する例を紹介しているものが多くある。これらのサンプルを参考にして実装されたアプリは、脆弱性を作りこむ可能性がある。

プライベート証明書で HTTPS 通信をしたい場合には「5.4.1.3. プライベート証明書で *HTTPS* 通信する」の安全なサンプルコードを参照すること。

本来ならば独自の TrustManager を安全に実装することも可能であるが、暗号処理や暗号通信に十分な知識をもった技術者でなければミスを作り込む危険性があるため、このルールはあえて必須とした。

5.4.2.5 独自の **HostnameVerifier** は作らない （必須）

自己署名証明書などのプライベート証明書で HTTPS 通信するためには、サーバー証明書検証に使う KeyStore を変更するだけで済む。しかしながら「5.4.3.3. 証明書検証を無効化する危険なコード」で説明しているように、インターネット上で公開されているサンプルコードには、危険な HostnameVerifier を利用する例を紹介しているものが多くある。これらのサンプルを参考にして実装したアプリは、脆弱性を作りこむ可能性がある。

プライベート証明書で HTTPS 通信をしたい場合には「5.4.1.3. プライベート証明書で *HTTPS* 通信する」の安全なサンプルコードを参照すること。

本来ならば独自の HostnameVerifier を安全に実装することも可能であるが、暗号処理や暗号通信に十分な知識をもった技術者でなければミスを作り込む危険性があるため、このルールはあえて必須とした。

5.4.3 アドバンスト

5.4.3.1 プライベート証明書の作成方法とサーバー設定

ここでは Ubuntu や CentOS などの Linux 環境におけるプライベート証明書の作成方法とサーバー設定について説明する。プライベート証明書は私的に発行されたサーバー証明書のことである。Cybertrust や VeriSign などの第三者認証局から発行されたサーバー証明書と区別してプライベート証明書と呼ばれる。

プライベート認証局の作成

まずプライベート証明書を発行するためのプライベート認証局を作成する。Cybertrust や VeriSign などの第三者認証局と区別してプライベート認証局と呼ばれる。1 つのプライベート認証局で複数のプライベート証明書を発行できる。プライベート認証局を作成した PC は、限られた信頼できる人物しかアクセスできないように厳重に管理されなければならない。

プライベート認証局を作成するには、下記のシェルスクリプト newca.sh および設定ファイル openssl.cnf を作成し実行する。シェルスクリプト中の CASTART および CAEND は認証局の有効期間、CASUBJ は認証局の名称であるので、作成する認証局に合わせて変更すること。シェルスクリプト実行の際には認証局アクセスのためのパスワードが合計 3 回聞かれるので、同じパスワードを入力すること。

newca.sh -- プライベート認証局を作成するシェルスクリプト

```
#!/bin/bash

umask 0077

CONFIG=openssl.cnf
CATOP=./CA
CAKEY=cakey.pem
CAREQ=careq.pem
CACERT=cacert.pem
CAX509=cacert.crt
CASTART=1301010000000Z    # 2013/01/01 00:00:00 GMT
CAEND=2301010000000Z      # 2023/01/01 00:00:00 GMT
CASUBJ="/CN=JSSEC Private CA/O=JSSEC/ST=Tokyo/C=JP"

mkdir -p ${CATOP}
mkdir -p ${CATOP}/certs
mkdir -p ${CATOP}/crl
mkdir -p ${CATOP}/newcerts
mkdir -p ${CATOP}/private
touch ${CATOP}/index.txt

openssl req -new -newkey rsa:2048 -sha256 -subj "${CASUBJ}" \
        -keyout ${CATOP}/private/${CAKEY} -out ${CATOP}/${CAREQ}
openssl ca -selfsign -md sha256 -create_serial -batch \
        -keyfile ${CATOP}/private/${CAKEY} \
        -startdate ${CASTART} -enddate ${CAEND} -extensions v3_ca \
        -in ${CATOP}/${CAREQ} -out ${CATOP}/${CACERT} \
        -config ${CONFIG}
openssl x509 -in ${CATOP}/${CACERT} -outform DER -out ${CATOP}/${CAX509}
```

openssl.cnf - 2 つのシェルスクリプトが共通に参照する openssl コマンドの設定ファイル

```
[ ca ]
default_ca      = CA_default            # The default ca section

[ CA_default ]
dir             = ./CA                  # Where everything is kept
certs           = $dir/certs            # Where the issued certs are kept
crl_dir         = $dir/crl              # Where the issued crl are kept
database        = $dir/index.txt        # database index file.
#unique_subject = no                    # Set to 'no' to allow creation of several ctificates with same␣
↪subject.
```

(continues on next page)

(continued from previous page)

```
new_certs_dir   = $dir/newcerts          # default place for new certs.
certificate     = $dir/cacert.pem        # The CA certificate
serial          = $dir/serial            # The current serial number
crlnumber       = $dir/crlnumber         # the current crl number must be commented out to leave a V1 CRL
crl             = $dir/crl.pem           # The current CRL
private_key     = $dir/private/cakey.pem # The private key
RANDFILE        = $dir/private/.rand     # private random number file
x509_extensions = usr_cert               # The extentions to add to the cert
name_opt        = ca_default             # Subject Name options
cert_opt        = ca_default             # Certificate field options
policy          = policy_match

[ policy_match ]
countryName             = match
stateOrProvinceName     = match
organizationName        = supplied
organizationalUnitName  = optional
commonName              = supplied
emailAddress            = optional

[ usr_cert ]
basicConstraints        = CA:FALSE
nsComment               = "OpenSSL Generated Certificate"
subjectKeyIdentifier    = hash
authorityKeyIdentifier  = keyid,issuer
subjectAltName          = @alt_names

[ v3_ca ]
subjectKeyIdentifier    = hash
authorityKeyIdentifier  = keyid:always,issuer
basicConstraints        = CA:true

[ alt_names ]
DNS.1                   = ${ENV::HOSTNAME}
DNS.2                   = *.${ENV::HOSTNAME}
```

上記シェルスクリプトを実行すると、作業ディレクトリ直下に CA というディレクトリが作成される。この CA ディレクトリがプライベート認証局である。CA/cacert.crt ファイルがプライベート認証局のルート証明書であり、「5.4.1.3. プライベート証明書で *HTTPS* 通信する」の assets に使用されたり、「5.4.3.2. *Android OS* の証明書ストアにプライベート認証局のルート証明書をインストールする」で Android 端末にインストールされたりする。

プライベート証明書の作成

プライベート証明書を作成するには、下記のシェルスクリプト newsv.sh を作成し実行する。シェルスクリプト中の SVSTART および SVEND はプライベート証明書の有効期間、SVSUBJ は Web サーバーの名称であるので、対象 Web サーバーに合わせて変更すること。特に SVSUBJ の/CN で指定する Web サーバーのホスト名はタイプミスがないように気を付けること。シェルスクリプトを実行すると認証局アクセスのためのパスワードが聞かれるので、プライベート認証局を作成するときに指定したパスワードを入力すること。その後、合計 2 回の y/n を聞かれるので y を入力すること。

newsv.sh - プライベート証明書を発行するシェルスクリプト

```
#!/bin/bash
```

(continues on next page)

(continued from previous page)

```
umask 0077

CONFIG=openssl.cnf
CATOP=./CA
CAKEY=cakey.pem
CACERT=cacert.pem
SVKEY=svkey.pem
SVREQ=svreq.pem
SVCERT=svcert.pem
SVX509=svcert.crt
SVSTART=130101000000Z    # 2013/01/01 00:00:00 GMT
SVEND=230101000000Z      # 2023/01/01 00:00:00 GMT
HOSTNAME=selfsigned.jssec.org
SVSUBJ="/CN="${HOSTNAME}"/O=JSSEC Secure Coding Group/ST=Tokyo/C=JP"

openssl genrsa -out ${SVKEY} 2048
openssl req -new -key ${SVKEY} -subj "${SVSUBJ}" -out ${SVREQ}
openssl ca -md sha256 \
        -keyfile ${CATOP}/private/${CAKEY} -cert ${CATOP}/${CACERT} \
        -startdate ${SVSTART} -enddate ${SVEND} \
        -in ${SVREQ} -out ${SVCERT} -config ${CONFIG}
openssl x509 -in ${SVCERT} -outform DER -out ${SVX509}
```

上記シェルスクリプトを実行すると、作業ディレクトリ直下に Web サーバー用のプライベートキーファイル svkey.pem およびプライベート証明書ファイル svcert.pem が生成される。

Web サーバーが Apache である場合には、設定ファイル中に上で作成した prikey.pem と cert.pem を次のように指定するとよい。

```
SSLCertificateFile "/path/to/svcert.pem"
SSLCertificateKeyFile "/path/to/svkey.pem"
```

5.4.3.2 Android OS の証明書ストアにプライベート認証局のルート証明書をインストールする

「5.4.1.3. プライベート証明書で *HTTPS* 通信する」のサンプルコードは、1 つのアプリにプライベート認証局のルート証明書を持たせることで、プライベート証明書で運用する Web サーバーに HTTPS 接続する方法を紹介した。ここでは Android OS にプライベート認証局のルート証明書をインストールすることで、すべてのアプリがプライベート証明書で運用する Web サーバーに HTTPS 接続する方法を紹介する。インストールしてよいのは、信頼できる認証局の発行した証明書に限ることに注意すること。

ただし、ここで述べる方法は Android 6.0(API Level 23) 以前のバージョンにのみ適用可能なものである。Android 7.0(API Level 24) 以降では、プライベート認証局のルート証明書をインストールしても、システムはこれを無視する。API Level 24 以降でプライベート証明書を利用したい場合は、「5.4.3.7. *Network Security Configuration*」の「プライベート証明書で HTTPS 通信する」の項を参照されたい。

まずプライベート認証局のルート証明書ファイル cacert.crt を Android 端末の内部ストレージにコピーする。なおサンプルコードで使用しているルート証明書ファイルは https://www.jssec.org/dl/android_securecoding_sample_cacert.crt からも取得できる。

次に Android の設定メニューのセキュリティを開き、下図のような手順を進めることで Android OS にルート証明書をインストールすることができる。

図 5.4.2　プライベート認証局のルート証明書のインストール手順

図 5.4.3　ルート証明書がインストールされていることの確認

Android OS にプライベート認証局のルート証明書をインストールすると、その認証局から発行されたプライベート証明書をすべてのアプリで正しく証明書検証できるようになる。下図は Chrome ブラウザで https://selfsigned.jssec.org/droid_knight.png を表示した場合の例である。

図 5.4.4　ルート証明書のインストール後はプライベート証明書を正しく検証できるようになる

この方法を使えば「5.4.1.2. *HTTPS* 通信する」のサンプルコードでもプライベート証明書で運用する Web サーバーに HTTPS 接続できるようになる。

5.4.3.3 証明書検証を無効化する危険なコード

インターネット上にはサーバー証明書検証エラーを無視して HTTPS 通信をするサンプルコードが多数掲載されている。これらのサンプルコードはプライベート証明書を使って HTTPS 通信を実現する方法として紹介されているため、そうしたサンプルコードをコピー＆ペーストして利用しているアプリが多数存在している。残念ながらこうしたサンプルコードは中間者攻撃に脆弱なものであることが多く、この記事の冒頭で「2012 年には Android アプリの HTTPS 通信の実装方法における欠陥が多く指摘された。」と述べたように、こうしたインターネット上の脆弱なサンプルコードを利用してしまったと思われる多くの脆弱な Android アプリが報告されている。

ここではこうした脆弱な HTTPS 通信のサンプルコードの断片を紹介する。こうしたサンプルコードを見かけた場合には「5.4.1.3. プライベート証明書で *HTTPS* 通信する」のサンプルコードに置き換えるなどしていただきたい。

危険：空っぽの TrustManager を作るケース

```
        TrustManager tm = new X509TrustManager() {

            @Override
            public void checkClientTrusted(X509Certificate[] chain,
                    String authType) throws CertificateException {
                // 何もしない → どんな証明書でも受付ける
            }

            @Override
            public void checkServerTrusted(X509Certificate[] chain,
                    String authType) throws CertificateException {
```

(continues on next page)

(continued from previous page)

```
            // 何もしない → どんな証明書でも受付ける
        }

        @Override
        public X509Certificate[] getAcceptedIssuers() {
            return null;
        }
    };
```

危険：空っぽの HostnameVerifier を作るケース

```
    HostnameVerifier hv = new HostnameVerifier() {
        @Override
        public boolean verify(String hostname, SSLSession session) {
            // 常に true を返す → どんなホスト名でも受付ける
            return true;
        }
    };
```

危険：ALLOW_ALL_HOSTNAME_VERIFIER を使っているケース

```
    SSLSocketFactory sf;
    // …
    sf.setHostnameVerifier(SSLSocketFactory.ALLOW_ALL_HOSTNAME_VERIFIER);
```

5.4.3.4 HTTP リクエストヘッダを設定する際の注意点

HTTP および HTTPS 通信において、独自の HTTP リクエストヘッダを設定したい場合は、URLConnection クラスの setRequestProperty() メソッド、もしくは addRequestProperty() メソッドを使用する。これらメソッドの引数に外部からの入力データを用いる場合は、HTTP ヘッダ・インジェクションの対策が必要となる。HTTP ヘッダ・インジェクションによる攻撃の最初のステップとなるのは、HTTP ヘッダの区切り文字である改行コードを入力データに含めることであるため、入力データから改行コードを排除するようにしなければならない。

HTTP リクエストヘッダを設定する

```
public byte[] openConnection(String strUrl, String strLanguage, String strCookie) {
    // HttpURLConnection は URLConnection の派生クラス
    HttpURLConnection connection;

    try {
        URL url = new URL(strUrl);
        connection = (HttpURLConnection) url.openConnection();
        connection.setRequestMethod("GET");

        // ★ポイント★ HTTP リクエストヘッダに入力値を使用する場合は、アプリケーション要件に従って
        // 入力データをチェックする（※）
        if (strLanguage.matches("^[a-zA-Z ,-]+$")) {
            connection.addRequestProperty("Accept-Language", strLanguage);
        } else {
            throw new IllegalArgumentException("Invalid Language : " + strLanguage);
        }
        // ★ポイント★ もしくは入力データを URL エンコードする（というアプリケーション要件にする）
```

(continues on next page)

(continued from previous page)

```
            connection.setRequestProperty("Cookie", URLEncoder.encode(strCookie, "UTF-8"));

            connection.connect();

            // ～省略～
```

※「3.2. 入力データの安全性を確認する」を参照

5.4.3.5 ピンニングによる検証の注意点と実装例

アプリが HTTPS 通信を行う際は、通信開始時のハンドシェイク処理において、接続先サーバーから送られてくる証明書が第三者認証局により署名されているかどうかの検証が行われる。しかし、攻撃者が第三者認証局から不正な証明書を入手したり、認証局の署名鍵を入手して不正な証明書を作成したりした場合、その攻撃者により不正なサーバーへの誘導や中間者攻撃が行われても、アプリはそれらの攻撃をハンドシェイク処理で検出することができず、結果として被害につながってしまう可能性がある。

このような不正な第三者認証局の証明書を用いた中間者攻撃に対しては「ピンニングによる検証」が有効である。これは、あらかじめ接続先サーバーの証明書や公開鍵をアプリ内に保持しておき、それらの情報をハンドシェイク処理で用いたり、ハンドシェイク処理後に再検証したりする方法である。

ピンニングによる検証は、公開鍵基盤（PKI）の基礎である第三者認証局の信頼性が損なわれた場合に備え、通信の安全性を補填する目的で用いられる。開発者は自身のアプリが扱う資産レベルに応じて、この検証を行うかどうか検討してほしい。

アプリ内に保持した証明書・公開鍵をハンドシェイク処理で使用する

アプリ内に保持しておいた接続先サーバーの証明書や公開鍵の情報をハンドシェイク処理で用いるためには、それらの情報を含めた独自の KeyStore を作成して通信に用いる。これにより、上記のような不正な第三者認証局の証明書を用いた中間者攻撃が行われても、ハンドシェイク処理において不正を検出することができるようになる。独自の KeyStore を設定して HTTPS 通信を行う具体的な方法は「5.4.1.3. プライベート証明書で *HTTPS* 通信する」で紹介したサンプルコードを参照すること。

アプリ内に保持した証明書・公開鍵を用いてハンドシェイク処理後に再検証する

ハンドシェイク処理が行われた後に接続先を再検証するためには、まずハンドシェイク処理で検証されシステムに信頼された証明書チェーンを取得し、その証明書チェーンを、あらかじめアプリ内に保持しておいた情報と照合する。照合の結果、保持しておいた情報と一致するものが含まれていれば通信を許可し、含まれていなければ通信処理を中断させればよい。

ただし、ハンドシェイク処理でシステムに信頼された証明書チェーンを取得する際に以下のメソッドを使用すると、期待通りの証明書チェーンが得られず、結果としてピンニングによる検証が正常に機能しなくなってしまう危険がある[*14]。

- javax.net.ssl.SSLSession.getPeerCertificates()
- javax.net.ssl.SSLSession.getPeerCertificateChain()

[*14] この危険性については以下の記事で詳しく説明されている https://www.cigital.com/blog/ineffective-certificate-pinning-implementations/

これらのメソッドが返すのは、ハンドシェイク処理でシステムに信頼された証明書チェーンではなく、アプリが通信相手から受け取った証明書チェーンそのものである。そのため、中間者攻撃により不正な第三者認証局の証明書が証明書チェーンに付け加えられても、上記のメソッドはハンドシェイク処理でシステムが信用した証明書だけでなく、本来アプリが接続しようとしていたサーバーの証明書も一緒に返してしまう。この「本来アプリが接続しようとしていたサーバーの証明書」は、ピンニングによる検証のためアプリ内にあらかじめ保持しておいたものと同等の証明書なので、再検証を行っても不正を検出することができない。このような理由から、ハンドシェイク処理後の再検証を実装する際に上記のメソッドを使用することは避けるべきである。

Android 4.2（API Level 17）以上であれば、上記のメソッドの代わりに net.http.X509TrustManagerExtensions の checkServerTrusted() を使用することで、ハンドシェイク処理でシステムに信頼された証明書チェーンのみを取得することができる。

X509TrustManagerExtensions を用いたピンニング検証の例

```
// 正しい通信先サーバーの証明書に含まれる公開鍵の SHA-256 ハッシュ値を保持（ピンニング）
private static final Set<String> PINS = new HashSet<>(Arrays.asList(
        new String[] {
                "d9b1a68fceaa460ac492fb8452ce13bd8c78c6013f989b76f186b1cbba1315c1",
                "cd13bb83c426551c67fabcff38d4496e094d50a20c7c15e886c151deb8531cdc"
        }
));

// AsyncTask のワーカースレッドで通信する
protected Object doInBackground(String... strings) {

    // ～省略～

   // ハンドシェイク時の検証によりシステムに信頼された証明書チェーンを取得する
   X509Certificate[] chain = (X509Certificate[]) connection.getServerCertificates();
   X509TrustManagerExtensions trustManagerExt = new X509TrustManagerExtensions((X509TrustManager)␣
↪(trustManagerFactory.getTrustManagers()[0]));
   List<X509Certificate> trustedChain = trustManagerExt.checkServerTrusted(chain, "RSA", url.getHost());

   // 公開鍵ピンニングを用いて検証する
   boolean isValidChain = false;
   for (X509Certificate cert : trustedChain) {
       PublicKey key = cert.getPublicKey();
       MessageDigest md = MessageDigest.getInstance("SHA-256");
       String keyHash = bytesToHex(md.digest(key.getEncoded()));

       // ピンニングしておいた公開鍵のハッシュ値と比較する
       if(PINS.contains(keyHash)) isValidChain = true;
   }
   if (isValidChain) {
       // 処理を継続する
   } else {
       // 処理を継続しない
   }

    // ～省略～
}

private String bytesToHex(byte[] bytes) {
    StringBuilder sb = new StringBuilder();
    for (byte b : bytes) {
```

(continues on next page)

(continued from previous page)

```
        String s = String.format("%02x", b);
        sb.append(s);
    }
    return sb.toString();
}
```

5.4.3.6 Google Play 開発者サービスを利用した OpenSSL の脆弱性対策

Google Play 開発者サービス（バージョン 5.0 以降）では、Provider Installer という仕組みが提供されている。これは、OpenSSL を含む暗号関連技術の実装である Security Provider の脆弱性対策に利用できる。詳しくは「5.6.3.5. *Google Play 開発者サービスによる Security Provider の脆弱性対策*」を参照のこと。

5.4.3.7 Network Security Configuration

Android 7.0（API Level 24）において、ネットワーク通信時のセキュリティ設定をアプリ毎に行うことができる Network Security Configuration が導入された。この仕組みを利用することにより、プライベート証明書での HTTPS 通信やピンニングによる証明書検証のほか、非暗号化（HTTP）通信の抑制、デバッグ時のみ有効なプライベート証明書の導入など、アプリのセキュリティを向上させる種々の施策をアプリに簡単に取り入れることができる[*15]。

Network Security Configuration の各種機能は xml ファイルの設定を行うだけで使用でき、アプリが行う HTTP および HTTPS 通信全てに適用することができる。その結果、アプリのコードに修正や追加処理を行う必要がなくなるため、実装がシンプルになりバグや脆弱性の作り込み防止に効果があると考えられる。

プライベート証明書で HTTPS 通信する

「5.4.1.3. プライベート証明書で *HTTPS* 通信する」で、私的に発行したサーバー証明書（プライベート証明書）で HTTPS 通信をするためのサンプルコードを示した。Network Security Configuration を用いれば、開発者がプライベート証明書の検証処理を明示的に実装しなくても、「5.4.1.2. *HTTPS* 通信する」のサンプルコードで示した通常の HTTPS 通信と同じ実装でプライベート証明書を用いることができる。

特定ドメインへの通信時にプライベート証明書を用いる

```
<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
    <domain-config>
        <domain includeSubdomains="true">jssec.org</domain>
        <trust-anchors>
            <certificates src="@raw/private_ca" />
        </trust-anchors>
    </domain-config>
</network-security-config>
```

上記の例では、通信で使用するプライベート証明書（private_ca）をアプリ内にリソースとして保持しておき、それらを利用する条件や適用範囲を xml ファイルに記述している。<domain-config>タグを使用することで特定のドメインに対してのみプライベート証明書が適用される。アプリが行う全ての HTTPS 通信に対してプライベート証明書を用いるためには、以下のように<base-config>タグを用いればよい。

アプリが行う全ての HTTPS 通信時にプライベート証明書を用いる

[*15] Network Security Configuration の詳細については以下を参照すること https://developer.android.com/training/articles/security-config.html

```
<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
    <base-config>
        <trust-anchors>
            <certificates src="@raw/private_ca" />
        </trust-anchors>
    </base-config>
</network-security-config>
```

ピンニングによる検証

「5.4.3.5. ピンニングによる検証の注意点と実装例」でピンニングによる証明書の検証について説明した。Network Security Configuration を用い以下のように設定すれば、コード上の検証処理が不要となり、xml の記述だけで検証を行うことができる。

HTTPS 通信時にピンニングによる検証を行う

```
<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
    <domain-config>
        <domain includeSubdomains="true">jssec.org</domain>
        <pin-set expiration="2018-12-31">
            <pin digest="SHA-256">e30Lky+iWK21yHSls5DJoRzNikOdvQUOGXvurPidc2E=</pin>
            <!-- バックアップ用 -->
            <pin digest="SHA-256">fwza0LRMXouZHRC8Ei+4PyuldPDcf3UKgO/04cDM1oE=</pin>
        </pin-set>
    </domain-config>
</network-security-config>
```

上記の<pin>タグに記述するのは、ピンニング検証の対象となる公開鍵のハッシュ値を base64 でエンコードしたものである。また、ハッシュ関数は SHA-256 のみサポートされている。

非暗号化（**HTTP**）通信の抑制

Network Security Configuration を用いて、アプリの HTTP 通信（非暗号化通信）を抑制することができる。

非暗号化通信を抑制する方法は次の通りである：

1. 基本的には、*<base-config>* タグにより、すべてのドメインとの通信で非暗号化通信 (HTTP 通信) を抑制する[*16]
2. やむを得えず非暗号化通信が必要なドメインのみ *<domain-config>* タグにより個別に非暗号化通信を許可する例外設定をおこなう。なお、非暗号化通信を許可して良いかどうかの判断には「5.4.1.1. *HTTP* 通信する」を参考にすること

非暗号化通信の抑制は cleartextTrafficPermitted 属性を false に設定することで行われる。下にこの例を示す。

```
<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
    <!-- 非暗号化通信はデフォルトで許可しない -->
```

(continues on next page)

[*16] HTTP 以外の通信方式に対してどのような制御が行われるかについては、以下を参照すること https://developer.android.com/reference/android/security/NetworkSecurityPolicy.html#isCleartextTrafficPermitted

(continued from previous page)

```
    <base-config cleartextTrafficPermitted="false">
    </base-config>
    <!-- 例外的に非暗号化通信を許可する必要のあるサイトを、
         <domain-config> タグにより明示的に "true" に設定する -->
    <domain-config cleartextTrafficPermitted="true">
       <domain includeSubdomains="true">www.jssec.org</domain>
    </domain-config>
</network-security-config>
```

この設定は Android 8.0（API Level 26）以降では WebView にも適用されるが、Android 7.0(API Level 25) 以前では WebView には適用されないことに注意する必要がある。

Android 9.0 (API Level 28) 未満では、属性 cleartextTrafficPermitted のデフォルト値が true であったが、Android 9.0 以降では false となった。従って API Level 28 以降をターゲットとしている場合は上の例の <base-config>での宣言は不要である。しかし意図を明確にし、またターゲット API Level による挙動の違いの影響を避けるため、上の例のように明示的に記載することを推奨する。

デバッグ専用のプライベート証明書

アプリ開発時にデバッグ目的でプライベート証明書を用いた開発用サーバーとの HTTPS 通信を行う場合、開発者は「5.4.3.3. 証明書検証を無効化する危険なコード」で述べたような、証明書検証を無効化させる危険な実装をアプリに組み込んでしまわないよう注意する必要がある。Network Security Configuration で以下のような設定を行えば、デバッグ時にのみ（AndroidManifest.xml 内の android:debuggable が”true” である場合のみ）使用する証明書を指定することができるため、前述のような危険なコードを製品版に残してしまう危険性がなくなり、脆弱性の作り込み防止に役立てることができる。

デバッグ時にのみプライベート証明書を用いる

```
    <?xml version="1.0" encoding="utf-8"?>
    <network-security-config>
        <debug-overrides>
            <trust-anchors>
                <certificates src="@raw/private_cas" />
            </trust-anchors>
        </debug-overrides>
    </network-security-config>
```

5.4.3.8 （コラム） セキュア接続の TLS1.2 への移行について

米国立標準技術研究所 (NIST)[*17] では、SSL と TLS 1.0 についてセキュリティ上の問題を報告しておりサーバーでの利用が非推奨となっている。特に 2014 年から 2015 年には Heartbleed（2014 年 4 月）[*18], POODLE（2014 年 10 月）[*19], FREAK（2015 年 3 月）[*20] 等の脆弱性が発表されており、特に OpenSSL(暗号ソフトウェアライブラリ) 上で発見された脆弱性 Heartbleed では、日本の企業でも不正アクセスを受け顧客データを閲覧されるなどの被害が出ている。[*21]

このような背景から米国の政府調達においては、これらの使用を禁止しているが、商用においては影響範囲の大きさか

[*17] 米国立標準技術研究所, NIST (https://www.nist.gov/)
[*18] Heartbleed（CVE-2014-0160）, IPA (https://www.ipa.go.jp/security/ciadr/vul/20140408-openssl.html)
[*19] POODLE（CVE-2014-3566）, IPA (https://www.ipa.go.jp/security/announce/20141017-ssl.html)
[*20] FREAK（CVE-2015-0204）, NIST (https://nvd.nist.gov/vuln/detail/CVE-2015-0204)
[*21] TLS/SSL 既知の脆弱性, Wiki (https://ja.wikipedia.org/wiki/Transport_Layer_Security#TLS/SSL の既知の脆弱性)

ら (特に TLS 1.0 はセキュリティパッチを施しながら) 現在においてもインターネットの暗号化技術として広く使われているのが現状である。しかし、ここ数年のセキュリティ騒動や新しいバージョンの TLS の普及から「SSL や TLS の古いバージョン」のサポートを取りやめるサイトやサービスも増えており、TLS1.2 への移行が確実に進みつつある。[*22]

例えば、移行の一例として PCI SSC(Payment Card Industry Security Standards Council) によって策定された「ペイメントカード業界データセキュリティ基準 (PCI DSS: Payment Card Industry Data Security Standard)」と呼ばれるセキュリティ基準がある。[*23]

現在 E コマースはスマートフォンやタブレットでも広く利用されており、決済にはクレジットカードを利用する事が普通であろう。本書（Android アプリのセキュア設計・セキュアコーディングガイド）を利用するユーザーにおいても、アプリケーションを介してクレジットカード情報などをサーバサイドに送信するサービスを企画するケースも多いと予想するが、ネットワークでクレジットカードを利用する際にはその経路の安全性を確保することが必要で、PCI DSS は、このようなサービスにおける会員データを取り扱う際の基準であり、カードの不正利用や情報漏えいなどを防止する目的で定められている。このセキュリティ基準の中で、インターネット上でのクレジットカードの処理に TLS 1.0 の使用は非推奨とされ、TLS 1.2（ハッシュ関数 SHA-2(SHA-256 や SHA-384) の利用、認証付暗号利用モードが利用可能な暗号スイートのサポートなど、より強力な暗号アルゴリズムの利用が可能になっている）などに対応することが求められている。

スマートフォンとサーバー間の通信において、経路上の安全を確保することは、クレジット情報の取り扱いに限らずプライバシー情報のやりとりやその他情報のやりとりにおいても非常に重要な点であると言えるため、サービスを提供する (サーバー) 側が TLS1.2 を利用したセキュア接続へ移行する事は差し迫った課題であるといえる。

一方、クライアント側である Android において TLS 1.1 以降に対応できる WebView 機能は Android 4.4(Kitkat) 以降であり、HTTP 通信を直接行う場合は多少追加実装を伴うが Android 4.1(Jelly Bean 前期) からとなる。

サービス開発者の中には、TLS 1.2 を採用すると Android 4.3 以前のユーザーを切り捨てることになり、少なからず影響があると思われる向きがあるかもしれない。しかしながら、下図の通り、最新のデータ（2018 年 8 月現在）[*24] によると現在利用されている Android OS のシェアは 4.4 以降が 95.9% と圧倒的であるため、扱う資産などの安全性を考慮して、TLS 1.2 への移行を真剣に検討されることをお薦めする。

[*22] SSL/TLS 暗号設定ガイドライン, IPA (https://www.ipa.go.jp/security/vuln/ssl_crypt_config.html)

[*23] ペイメントカード業界データセキュリティ基準 (PCI DSS), PCI SSC (https://ja.pcisecuritystandards.org/minisite/env2/)

[*24] Android OS のシェア, Andorid Developpers ダッシュボード (https://developer.android.com/about/dashboards/index.html)

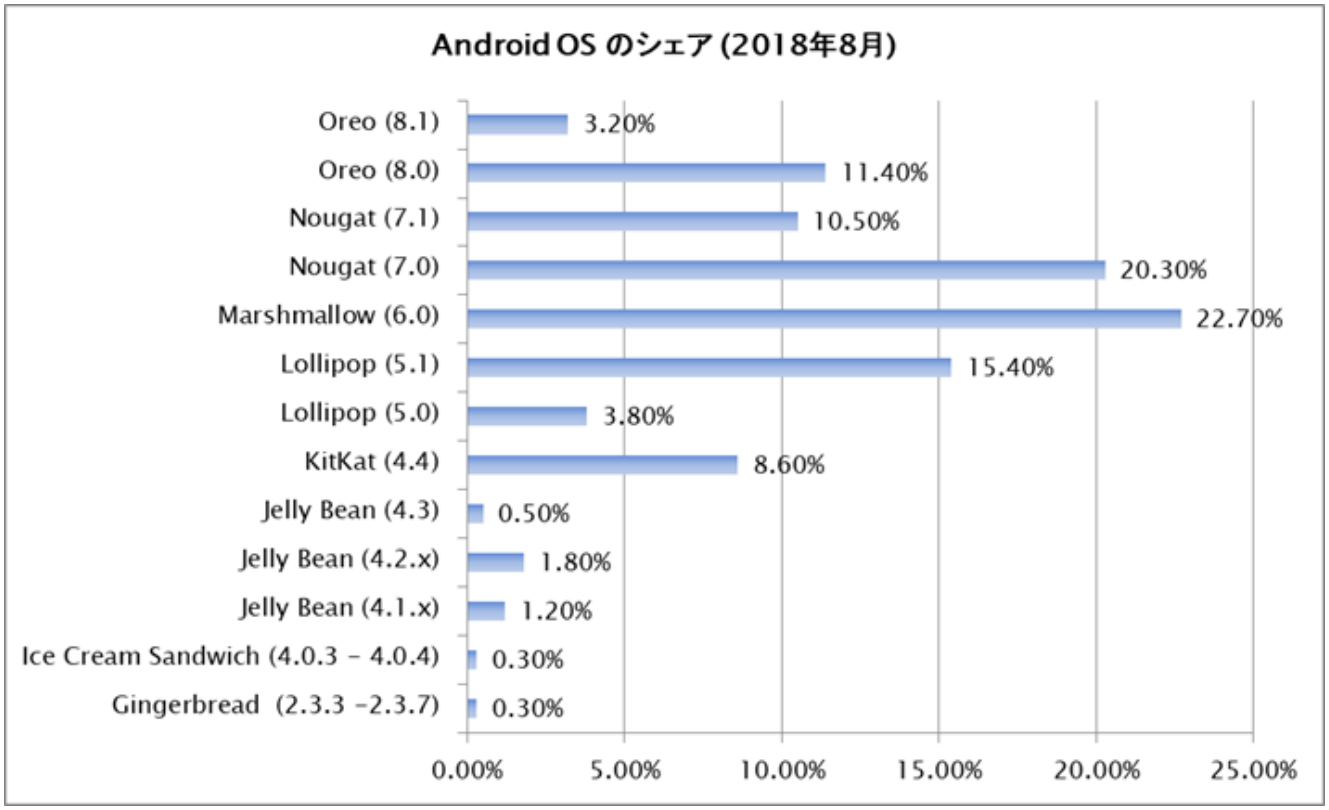

図 5.4.5　Android OS のシェア （Andorid Developpers サイトより引用）

5.5 プライバシー情報を扱う

近年、プライバシー情報を守るための世界的な潮流として「プライバシー・バイ・デザイン」が提唱されており、この概念に基づき各国政府においてもプライバシー保護のための法制化を進めているところである。

スマートフォン内の利用者情報を活用するアプリは、利用者が個人情報やプライバシーの観点から安全・安心にアプリを活用できるように、利用者情報を適切に取り扱うとともに、利用者に対して分かりやすい説明を行い、利用者に利用の可否の選択を促すことが求められる。そのためには、アプリがどのような情報をどのように扱うか等を示したアプリ毎のアプリケーション・プライバシーポリシー（以下、アプリ・プライバシーポリシー）を作成・提示するとともに、慎重な取り扱いが求められる利用者情報の取得・利用については事前にユーザーの同意を得る必要がある。なお、アプリケーション・プライバシーポリシーは従来から存在する「個人情報保護方針」や「利用規約」等とは異なり別途作成を要するものであることに留意すること。

プライバシーポリシーの作成や運用に関して、詳しくは総務省が提唱する『スマートフォンプライバシー イニシアティブ』、『スマートフォン プライバシーイニシアティブⅡ』及び『スマートフォン プライバシーイニシアティブⅢ』（以下総務省 SPI と省略）を参照のこと。

また、本記事で扱う用語については、本文内の解説および「5.5.3.2. 用語解説」を参照すること。

5.5.1 サンプルコード

アプリ・プライバシーポリシーの作成には、一般に公開されている「アプリケーション・プライバシーポリシー作成支援ツール[*25]」を利用することもできる。このツールの出力は HTML 形式および XML 形式となっており、概要版アプリ

[*25] http://www.kddi-research.jp/newsrelease/2013/090401.html

ケーション・プライバシーポリシーと詳細版アプリケーション・プライバシーポリシーのそれぞれのファイルが作成される。作成された XML ファイルには検査用のタグがつくなど、総務省 SPI に準拠した形となっている。以下のサンプルコードでは、上記ツールを使って作成した HTML ファイルを利用してアプリ・プライバシーポリシーを提示する例を示す。

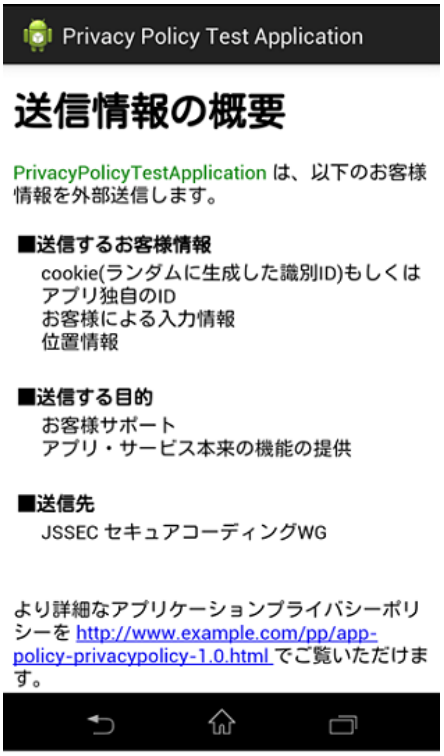

図 5.5.1　概要版アプリケーション・プライバシーポリシーの例

具体的には、次の判定フローに従うことで利用するサンプルコードを判断できる。

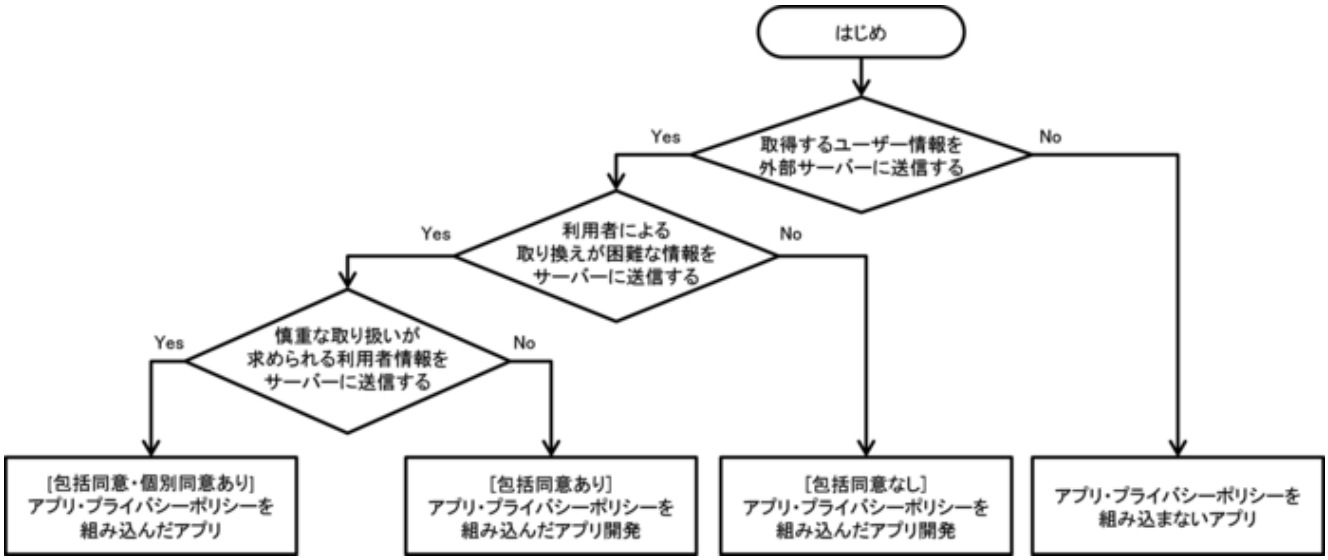

図 5.5.2　プライバシー情報を扱うサンプルコードを選択するフローチャート

ここで、包括同意とは、アプリ初回起動時のアプリ・プライバシーポリシーの提示・確認により、アプリがサーバーに送信する利用者情報について包括的に同意を得ることである。

また、個別同意とは、個々の利用者情報について送信の直前に個別の同意を得ることである。

5.5.1.1 [包括同意・個別同意あり] アプリ・プライバシーポリシーを組み込んだアプリ

ポイント：[包括同意・個別同意あり] アプリ・プライバシーポリシーを組み込んだアプリ

1. 初回起動時 (アップデート時) に、アプリが扱う利用者情報の送信について包括同意を得る
2. ユーザーの包括同意が得られていない場合は、利用者情報の送信はしない
3. 慎重な取り扱いが求められる利用者情報を送信する場合は、個別にユーザーの同意を得る
4. ユーザーの個別同意が得られていない場合は、該当情報の送信はしない
5. ユーザーがアプリ・プライバシーポリシーを確認できる手段を用意する
6. 送信した情報をユーザー操作により削除する手段を用意する
7. ユーザー操作により利用者情報の送信を停止する手段を用意する
8. 利用者情報の紐づけには UUID/cookie を利用する
9. アプリ・プライバシーポリシー概要版を assets フォルダ内に配置しておく

MainActivity.java

```
package org.jssec.android.privacypolicy;

import java.io.IOException;
import org.json.JSONException;
import org.json.JSONObject;
import org.jssec.android.privacypolicy.ConfirmFragment.DialogListener;

import com.google.android.gms.common.ConnectionResult;
import com.google.android.gms.common.GooglePlayServicesClient;
import com.google.android.gms.common.GooglePlayServicesUtil;
import com.google.android.gms.location.LocationClient;

import android.location.Location;
import android.os.AsyncTask;
import android.os.Bundle;
import android.content.Intent;
import android.content.IntentSender;
import android.content.SharedPreferences;
import android.content.pm.PackageInfo;
import android.content.pm.PackageManager;
import android.content.pm.PackageManager.NameNotFoundException;
import android.support.v4.app.FragmentActivity;
import android.support.v4.app.FragmentManager;
import android.text.Editable;
import android.text.TextWatcher;
import android.view.Menu;
import android.view.MenuItem;
import android.view.View;
import android.widget.TextView;
import android.widget.Toast;

public class MainActivity extends FragmentActivity implements GooglePlayServicesClient.
↪ConnectionCallbacks, GooglePlayServicesClient.OnConnectionFailedListener, DialogListener {
    private static final String BASE_URL = "https://www.example.com/pp";
    private static final String GET_ID_URI = BASE_URL + "/get_id.php";
```

(continues on next page)

(continued from previous page)

```
    private static final String SEND_DATA_URI = BASE_URL + "/send_data.php";
    private static final String DEL_ID_URI = BASE_URL + "/del_id.php";

    private static final String ID_KEY = "id";
    private static final String LOCATION_KEY = "location";
    private static final String NICK_NAME_KEY = "nickname";

    private static final String PRIVACY_POLICY_COMPREHENSIVE_AGREED_KEY =
↪"privacyPolicyComprehensiveAgreed";
    private static final String PRIVACY_POLICY_DISCRETE_TYPE1_AGREED_KEY =
↪"privacyPolicyDiscreteType1Agreed";

    private static final String PRIVACY_POLICY_PREF_NAME = "privacypolicy_preference";
    private static final int CONNECTION_FAILURE_RESOLUTION_REQUEST = 257;

    private String UserId = "";
    private LocationClient mLocationClient = null;

    private final int DIALOG_TYPE_COMPREHENSIVE_AGREEMENT = 1;
    private final int DIALOG_TYPE_PRE_CONFIRMATION = 2;

    private static final int VERSION_TO_SHOW_COMPREHENSIVE_AGREEMENT_ANEW = 1;

    private TextWatcher watchHandler = new TextWatcher() {

        @Override
        public void beforeTextChanged(CharSequence s, int start, int count, int after) {
        }

        @Override
        public void onTextChanged(CharSequence s, int start, int before, int count) {
            boolean buttonEnable = (s.length() > 0);

            MainActivity.this.findViewById(R.id.buttonStart).setEnabled(buttonEnable);
        }

        @Override
        public void afterTextChanged(Editable s) {
        }
    };

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        // ユーザー識別用 ID をサーバーから取得する
        new GetDataAsyncTask().execute();

        findViewById(R.id.buttonStart).setEnabled(false);
        ((TextView) findViewById(R.id.editTextNickname)).addTextChangedListener(watchHandler);

        int resultCode = GooglePlayServicesUtil.isGooglePlayServicesAvailable(this);
        if (resultCode == ConnectionResult.SUCCESS) {
            mLocationClient = new LocationClient(this, this, this);
```

(continues on next page)

(continued from previous page)

```
        }
    }

    @Override
    protected void onStart() {
        super.onStart();

        SharedPreferences pref = getSharedPreferences(PRIVACY_POLICY_PREF_NAME, MODE_PRIVATE);
        int privacyPolicyAgreed = pref.getInt(PRIVACY_POLICY_COMPREHENSIVE_AGREED_KEY, -1);

        if (privacyPolicyAgreed <= VERSION_TO_SHOW_COMPREHENSIVE_AGREEMENT_ANEW) {
            // ★ポイント 1★ 初回起動時（アップデート時）に、アプリが扱う利用者情報の送信について包括同意を得る
            // アップデート時については、新しい利用者情報を扱うようになった場合にのみ、再度包括同意を得る必要がある。
            ConfirmFragment dialog = ConfirmFragment.newInstance(R.string.privacyPolicy, R.string.
↪agreePrivacyPolicy, DIALOG_TYPE_COMPREHENSIVE_AGREEMENT);
            dialog.setDialogListener(this);
            FragmentManager fragmentManager = getSupportFragmentManager();
            dialog.show(fragmentManager, "dialog");
        }

        // Location 情報取得用
        if (mLocationClient != null) {
            mLocationClient.connect();
        }
    }

    @Override
    protected void onStop() {
        if (mLocationClient != null) {
            mLocationClient.disconnect();
        }
        super.onStop();
    }

    public void onSendToServer(View view) {
        // 慎重な取り扱いが求められる利用者情報を送信について既に同意を得ているか確認する
        // 実際には送信する情報の種別毎に同意を得る必要があることに注意すること
        SharedPreferences pref = getSharedPreferences(PRIVACY_POLICY_PREF_NAME, MODE_PRIVATE);
        int privacyPolicyAgreed = pref.getInt(PRIVACY_POLICY_DISCRETE_TYPE1_AGREED_KEY, -1);
        if (privacyPolicyAgreed <= VERSION_TO_SHOW_COMPREHENSIVE_AGREEMENT_ANEW) {
            // ★ポイント 3★ 慎重な取り扱いが求められる利用者情報を送信する場合は、個別にユーザーの同意を得る
            ConfirmFragment dialog = ConfirmFragment.newInstance(R.string.sendLocation, R.string.
↪cofirmSendLocation, DIALOG_TYPE_PRE_CONFIRMATION);
            dialog.setDialogListener(this);
            FragmentManager fragmentManager = getSupportFragmentManager();
            dialog.show(fragmentManager, "dialog");
        } else {
            // 同意済みのため、送信処理を開始する
            onPositiveButtonClick(DIALOG_TYPE_PRE_CONFIRMATION);
        }
    }

    public void onPositiveButtonClick(int type) {
        if (type == DIALOG_TYPE_COMPREHENSIVE_AGREEMENT) {
            // ★ポイント 1★ 初回起動時（アップデート時）に、アプリが扱う利用者情報の送信について包括同意を得る
```

(continues on next page)

(continued from previous page)

```
            SharedPreferences.Editor pref = getSharedPreferences(PRIVACY_POLICY_PREF_NAME, MODE_PRIVATE).
↪edit();
            pref.putInt(PRIVACY_POLICY_COMPREHENSIVE_AGREED_KEY, getVersionCode());
            pref.apply();
        } else if (type == DIALOG_TYPE_PRE_CONFIRMATION) {
            // ★ポイント 3★ 慎重な取り扱いが求められる利用者情報を送信する場合は、個別にユーザーの同意を得る
            if (mLocationClient != null && mLocationClient.isConnected()) {
                Location currentLocation = mLocationClient.getLastLocation();
                if (currentLocation != null) {
                    String locationData = "Latitude:" + currentLocation.getLatitude() + ", Longitude:" +␣
↪currentLocation.getLongitude();
                    String nickname = ((TextView) findViewById(R.id.editTextNickname)).getText().
↪toString();
                    Toast.makeText(MainActivity.this, this.getClass().getSimpleName() + "\n - nickname :
↪" + nickname + "\n - location : " + locationData, Toast.LENGTH_SHORT).show();
                    new SendDataAsyncTack().execute(SEND_DATA_URI, UserId, locationData, nickname);
                }
            }
            // 同意を得た旨、状態を保存する
            // 実際には送信する情報の種別毎に同意を得る必要があることに注意すること
            SharedPreferences.Editor pref = getSharedPreferences(PRIVACY_POLICY_PREF_NAME, MODE_PRIVATE).
↪edit();
            pref.putInt(PRIVACY_POLICY_DISCRETE_TYPE1_AGREED_KEY, getVersionCode());
            pref.apply();
        }
    }

    public void onNegativeButtonClick(int type) {
        if (type == DIALOG_TYPE_COMPREHENSIVE_AGREEMENT) {
            // ★ポイント 2★ ユーザーの包括同意が得られていない場合は、利用者情報の送信はしない
            // サンプルアプリではアプリケーションを終了する
            finish();
        } else if (type == DIALOG_TYPE_PRE_CONFIRMATION) {
            // ★ポイント 4★ ユーザーの個別同意が得られていない場合は、該当情報の送信はしない
            // ユーザー同意が得られなかったので何もしない
        }
    }

    private int getVersionCode() {
        int versionCode = -1;
        PackageManager packageManager = this.getPackageManager();
        try {
            PackageInfo packageInfo = packageManager.getPackageInfo(this.getPackageName(),␣
↪PackageManager.GET_ACTIVITIES);
            versionCode = packageInfo.versionCode;
        } catch (NameNotFoundException e) {
            // 例外処理は割愛
        }

        return versionCode;
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.main, menu);
```

(continues on next page)

(continued from previous page)

```
        return true;
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        switch (item.getItemId()) {
            case R.id.action_show_pp:
                // ★ポイント 5★ ユーザーがアプリ・プライバシーポリシーを確認できる手段を用意する
                Intent intent = new Intent();
                intent.setClass(this, WebViewAssetsActivity.class);
                startActivity(intent);
                return true;
            case R.id.action_del_id:
                // ★ポイント 6★ 送信した情報をユーザー操作により削除する手段を用意する
                new SendDataAsyncTack().execute(DEL_ID_URI, UserId);
                return true;
            case R.id.action_donot_send_id:
                // ★ポイント 7★ ユーザー操作により利用者情報の送信を停止する手段を用意する

                // 利用者情報の送信を停止した場合、包括同意に関する同意は破棄されたものとする
                SharedPreferences.Editor pref = getSharedPreferences(PRIVACY_POLICY_PREF_NAME, MODE_
↪PRIVATE).edit();
                pref.putInt(PRIVACY_POLICY_COMPREHENSIVE_AGREED_KEY, 0);
                pref.apply();

                // 本サンプルでは利用者情報を送信しない場合、ユーザーに提供する機能が無くなるため
                // この段階でアプリを終了する。この処理はアプリ毎の都合に合わせて変更すること。
                String message  = getString(R.string.stopSendUserData);
                Toast.makeText(MainActivity.this, this.getClass().getSimpleName() + " - " + message,␣
↪Toast.LENGTH_SHORT).show();
                finish();

                return true;
        }

        return false;
    }

    @Override
    public void onConnected(Bundle connectionHint) {
        if (mLocationClient != null && mLocationClient.isConnected()) {
            Location currentLocation = mLocationClient.getLastLocation();
            if (currentLocation != null) {
                String locationData = "Latitude \t: " + currentLocation.getLatitude() + "\n\tLongitude␣
↪\t: " + currentLocation.getLongitude();

                String text = "\n" + getString(R.string.your_location_title) + "\n\t" + locationData;

                TextView appText = (TextView) findViewById(R.id.appText);
                appText.setText(text);
            }
        }
    }

    @Override
```

(continues on next page)

(continued from previous page)

```
    public void onConnectionFailed(ConnectionResult result) {
        if (result.hasResolution()) {
            try {
                result.startResolutionForResult(this, CONNECTION_FAILURE_RESOLUTION_REQUEST);
            } catch (IntentSender.SendIntentException e) {
                e.printStackTrace();
            }
        }
    }

    @Override
    public void onDisconnected() {
        mLocationClient = null;
    }

    private class GetDataAsyncTask extends AsyncTask<String, Void, String> {
        private String extMessage = "";

        @Override
        protected String doInBackground(String... params) {
            // ★ポイント 8★ 利用者情報の紐づけには UUID/cookie を利用する
            // 本サンプルではサーバー側で生成した ID を利用する
            SharedPreferences sp = getSharedPreferences(PRIVACY_POLICY_PREF_NAME, MODE_PRIVATE);
            UserId = sp.getString(ID_KEY, null);
            if (UserId == null) {
                // SharedPreferences 内にトークンが存在しなため、サーバーから ID を取り寄せる。
                try {
                    UserId = NetworkUtil.getCookie(GET_ID_URI, "", "id");
                } catch (IOException e) {
                    // 証明書エラーなどの例外をキャッチする
                    extMessage = e.toString();
                }

                // 取り寄せた ID を SharedPreferences に保存する。
                sp.edit().putString(ID_KEY, UserId).commit();
            }
            return UserId;
        }

        @Override
        protected void onPostExecute(final String data) {
            String status = (data != null) ? "success" : "error";
            Toast.makeText(MainActivity.this, this.getClass().getSimpleName() + " - " + status + " : " +␣
↪extMessage, Toast.LENGTH_SHORT).show();
        }
    }

    private class SendDataAsyncTack extends AsyncTask<String, Void, Boolean> {
        private String extMessage = "";

        @Override
        protected Boolean doInBackground(String... params) {
            String url = params[0];
            String id = params[1];
            String location = params.length > 2 ? params[2] : null;
```

(continues on next page)

(continued from previous page)

```
            String nickname = params.length > 3 ? params[3] : null;

            Boolean result = false;
            try {
                JSONObject jsonData = new JSONObject();
                jsonData.put(ID_KEY, id);
                if (location != null)
                    jsonData.put(LOCATION_KEY, location);

                if (nickname != null)
                    jsonData.put(NICK_NAME_KEY, nickname);

                NetworkUtil.sendJSON(url, "", jsonData.toString());

                result = true;
            } catch (IOException e) {
                // 証明書エラーなどの例外をキャッチする
                extMessage = e.toString();
            } catch (JSONException e) {
                extMessage = e.toString();
            }
            return result;
        }


        @Override
        protected void onPostExecute(Boolean result) {
            String status = result ? "Success" : "Error";
            Toast.makeText(MainActivity.this, this.getClass().getSimpleName() + " - " + status + " : " +␣
↪extMessage, Toast.LENGTH_SHORT).show();
        }
    }
}
```

ConfirmFragment.java

```
package org.jssec.android.privacypolicy;

import android.app.Activity;
import android.app.AlertDialog;
import android.app.Dialog;
import android.content.Context;
import android.content.DialogInterface;
import android.content.Intent;
import android.os.Bundle;
import android.support.v4.app.DialogFragment;
import android.view.LayoutInflater;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.TextView;

public class ConfirmFragment extends DialogFragment {

    private DialogListener mListener = null;
```

(continues on next page)

(continued from previous page)

```
    public static interface DialogListener {
        public void onPositiveButtonClick(int type);

        public void onNegativeButtonClick(int type);
    }

    public static ConfirmFragment newInstance(int title, int sentence, int type) {
        ConfirmFragment fragment = new ConfirmFragment();
        Bundle args = new Bundle();
        args.putInt("title", title);
        args.putInt("sentence", sentence);
        args.putInt("type", type);
        fragment.setArguments(args);
        return fragment;
    }

    @Override
    public Dialog onCreateDialog(Bundle args) {
        // ★ポイント1★ 初回起動時に、アプリが扱う利用者情報の送信について包括同意を得る
        // ★ポイント3★ 慎重な取り扱いが求められる利用者情報を送信する場合は、個別にユーザーの同意を得る
        final int title = getArguments().getInt("title");
        final int sentence = getArguments().getInt("sentence");
        final int type = getArguments().getInt("type");

        LayoutInflater inflater = (LayoutInflater) getActivity().getSystemService(Context.LAYOUT_
↪INFLATER_SERVICE);
        View content = inflater.inflate(R.layout.fragment_comfirm, null);
        TextView linkPP = (TextView) content.findViewById(R.id.tx_link_pp);
        linkPP.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                // ★ポイント5★ ユーザーがアプリ・プライバシーポリシーを確認できる手段を用意する
                Intent intent = new Intent();
                intent.setClass(getActivity(), WebViewAssetsActivity.class);
                startActivity(intent);
            }
        });

        AlertDialog.Builder builder = new AlertDialog.Builder(getActivity());
        builder.setIcon(R.drawable.ic_launcher);
        builder.setTitle(title);
        builder.setMessage(sentence);
        builder.setView(content);

        builder.setPositiveButton(R.string.buttonOK, new DialogInterface.OnClickListener() {
            public void onClick(DialogInterface dialog, int whichButton) {
                if (mListener != null) {
                    mListener.onPositiveButtonClick(type);
                }
            }
        });

        builder.setNegativeButton(R.string.buttonNG, new DialogInterface.OnClickListener() {
            public void onClick(DialogInterface dialog, int whichButton) {
                if (mListener != null) {
```

(continues on next page)

(continued from previous page)

```
                    mListener.onNegativeButtonClick(type);

                }
            }
        });

        Dialog dialog = builder.create();
        dialog.setCanceledOnTouchOutside(false);

        return dialog;
    }

    @Override
    public void onAttach(Activity activity) {
        super.onAttach(activity);
        if (!(activity instanceof DialogListener)) {
            throw new ClassCastException(activity.toString() + " must implement DialogListener.");
        }
        mListener = (DialogListener) activity;
    }

    public void setDialogListener(DialogListener listener) {
        mListener = listener;
    }
}
```

WebViewAssetsActivity.java

```
package org.jssec.android.privacypolicy;

import android.app.Activity;
import android.os.Bundle;
import android.webkit.WebSettings;
import android.webkit.WebView;

public class WebViewAssetsActivity extends Activity {
    // ★ポイント 9★ アプリ・プライバシーポリシー概要版を assets フォルダ内に配置しておく
    private static final String ABST_PP_URL = "file:///android_asset/PrivacyPolicy/app-policy-abst-
↪privacypolicy-1.0.html";

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_webview);

        WebView webView = (WebView) findViewById(R.id.webView);
        WebSettings webSettings = webView.getSettings();

        webSettings.setAllowFileAccess(false);

        webView.loadUrl(ABST_PP_URL);
    }
}
```

5.5.1.2 [包括同意あり] アプリ・プライバシーポリシーを組み込んだアプリ

ポイント：[包括同意あり] アプリ・プライバシーポリシーを組み込んだアプリ

1. 初回起動時 (アップデート時) に、アプリが扱う利用者情報の送信について包括同意を得る
2. ユーザーの包括同意が得られていない場合は、利用者情報の送信はしない
3. ユーザーがアプリ・プライバシーポリシーを確認できる手段を用意する
4. 送信した情報をユーザー操作により削除する手段を用意する
5. ユーザー操作により利用者情報の送信を停止する手段を用意する
6. 利用者情報の紐づけには UUID/cookie を利用する
7. アプリ・プライバシーポリシー概要版を assets フォルダ内に配置しておく

```
MainActivity.java
package org.jssec.android.privacypolicynopreconfirm;

import java.io.IOException;
import org.json.JSONException;
import org.json.JSONObject;
import org.jssec.android.privacypolicynopreconfirm.MainActivity;
import org.jssec.android.privacypolicynopreconfirm.R;
import org.jssec.android.privacypolicynopreconfirm.ConfirmFragment.DialogListener;

import android.os.AsyncTask;

import android.os.Bundle;
import android.content.Intent;
import android.content.SharedPreferences;
import android.content.pm.PackageInfo;
import android.content.pm.PackageManager;
import android.content.pm.PackageManager.NameNotFoundException;
import android.support.v4.app.FragmentActivity;
import android.support.v4.app.FragmentManager;
import android.telephony.TelephonyManager;
import android.text.Editable;
import android.text.TextWatcher;
import android.view.Menu;
import android.view.MenuItem;
import android.view.View;
import android.widget.TextView;
import android.widget.Toast;

public class MainActivity extends FragmentActivity implements DialogListener {
    private final String BASE_URL = "https://www.example.com/pp";
    private final String GET_ID_URI = BASE_URL + "/get_id.php";
    private final String SEND_DATA_URI = BASE_URL + "/send_data.php";
    private final String DEL_ID_URI = BASE_URL + "/del_id.php";

    private final String ID_KEY = "id";
    private final String NICK_NAME_KEY = "nickname";
    private final String IMEI_KEY = "imei";
```

(continues on next page)

(continued from previous page)

```
    private final String PRIVACY_POLICY_AGREED_KEY = "privacyPolicyAgreed";

    private final String PRIVACY_POLICY_PREF_NAME = "privacypolicy_preference";

    private String UserId = "";

    private final int DIALOG_TYPE_COMPREHENSIVE_AGREEMENT = 1;

    private final int VERSION_TO_SHOW_COMPREHENSIVE_AGREEMENT_ANEW = 1;

    private TextWatcher watchHandler = new TextWatcher() {

        @Override
        public void beforeTextChanged(CharSequence s, int start, int count, int after) {
        }

        @Override
        public void onTextChanged(CharSequence s, int start, int before, int count) {
            boolean buttonEnable = (s.length() > 0);

            MainActivity.this.findViewById(R.id.buttonStart).setEnabled(buttonEnable);
        }

        @Override
        public void afterTextChanged(Editable s) {
        }
    };

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        // ユーザー識別用 ID をサーバーから取得する
        new GetDataAsyncTask().execute();

        findViewById(R.id.buttonStart).setEnabled(false);
        ((TextView) findViewById(R.id.editTextNickname)).addTextChangedListener(watchHandler);
    }

    @Override
    protected void onStart() {
        super.onStart();

        SharedPreferences pref = getSharedPreferences(PRIVACY_POLICY_PREF_NAME, MODE_PRIVATE);
        int privacyPolicyAgreed = pref.getInt(PRIVACY_POLICY_AGREED_KEY, -1);

        if (privacyPolicyAgreed <= VERSION_TO_SHOW_COMPREHENSIVE_AGREEMENT_ANEW) {
            // ★ポイント 1★ 初回起動時 (アップデート時) に、アプリが扱う利用者情報の送信について包括同意を得る
            // アップデート時については、新しい利用者情報を扱うようになった場合にのみ、再度包括同意を得る必要がある。
            ConfirmFragment dialog = ConfirmFragment.newInstance(R.string.privacyPolicy, R.string.
↪agreePrivacyPolicy, DIALOG_TYPE_COMPREHENSIVE_AGREEMENT);
            dialog.setDialogListener(this);
            FragmentManager fragmentManager = getSupportFragmentManager();
            dialog.show(fragmentManager, "dialog");
```

(continues on next page)

(continued from previous page)

```
        }
    }


    public void onSendToServer(View view) {
        String nickname = ((TextView) findViewById(R.id.editTextNickname)).getText().toString();
        TelephonyManager tm = (TelephonyManager) getSystemService(TELEPHONY_SERVICE);
        String imei = tm.getDeviceId();
        Toast.makeText(MainActivity.this, this.getClass().getSimpleName() + "\n - nickname : " +␣
↪nickname + ", imei = " + imei, Toast.LENGTH_SHORT).show();
        new SendDataAsyncTack().execute(SEND_DATA_URI, UserId, nickname, imei);
    }

    public void onPositiveButtonClick(int type) {
        if (type == DIALOG_TYPE_COMPREHENSIVE_AGREEMENT) {
            // ★ポイント 1★ 初回起動時に、アプリが扱う利用者情報の送信について包括同意を得る
            SharedPreferences.Editor pref = getSharedPreferences(PRIVACY_POLICY_PREF_NAME, MODE_PRIVATE).
↪edit();
            pref.putInt(PRIVACY_POLICY_AGREED_KEY, getVersionCode());
            pref.apply();
        }
    }

    public void onNegativeButtonClick(int type) {
        if (type == DIALOG_TYPE_COMPREHENSIVE_AGREEMENT) {
            // ★ポイント 2★ ユーザーの包括同意が得られていない場合は、利用者情報の送信はしない
            // サンプルアプリではアプリケーションを終了する
            finish();
        }
    }

    private int getVersionCode() {
        int versionCode = -1;
        PackageManager packageManager = this.getPackageManager();
        try {
            PackageInfo packageInfo = packageManager.getPackageInfo(this.getPackageName(),␣
↪PackageManager.GET_ACTIVITIES);
            versionCode = packageInfo.versionCode;
        } catch (NameNotFoundException e) {
            // 例外処理は割愛
        }

        return versionCode;
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.main, menu);
        return true;
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        switch (item.getItemId()) {
        case R.id.action_show_pp:
```

(continues on next page)

(continued from previous page)

```
            // ★ポイント 3★ ユーザーがアプリ・プライバシーポリシーを確認できる手段を用意する
            Intent intent = new Intent();
            intent.setClass(this, WebViewAssetsActivity.class);
            startActivity(intent);
            return true;
        case R.id.action_del_id:
            // ★ポイント 4★ 送信した情報をユーザー操作により削除する手段を用意する
            new SendDataAsyncTack().execute(DEL_ID_URI, UserId);
            return true;
        case R.id.action_donot_send_id:
            // ★ポイント 5★ ユーザー操作により利用者情報の送信を停止する手段を用意する

            // 利用者情報の送信を停止した場合、包括同意に関する同意は破棄されたものとする
            SharedPreferences.Editor pref = getSharedPreferences(PRIVACY_POLICY_PREF_NAME, MODE_PRIVATE).
↪edit();
            pref.putInt(PRIVACY_POLICY_AGREED_KEY, 0);
            pref.apply();

            // 本サンプルでは利用者情報を送信しない場合、ユーザーに提供する機能が無くなるため
            // この段階でアプリを終了する。この処理はアプリ毎の都合に合わせて変更すること。
            String message  = getString(R.string.stopSendUserData);
            Toast.makeText(MainActivity.this, this.getClass().getSimpleName() + " - " + message, Toast.
↪LENGTH_SHORT).show();
            finish();

            return true;        }
        return false;
    }

    private class GetDataAsyncTask extends AsyncTask<String, Void, String> {
        private String extMessage = "";

        @Override
        protected String doInBackground(String... params) {
            // ★ポイント 6★ 利用者情報の紐づけには UUID/cookie を利用する
            // 本サンプルではサーバー側で生成した ID を利用する
            SharedPreferences sp = getSharedPreferences(PRIVACY_POLICY_PREF_NAME, MODE_PRIVATE);
            UserId = sp.getString(ID_KEY, null);

            if (UserId == null) {
                // SharedPreferences 内にトークンが存在しなため、サーバーから ID を取り寄せる。
                try {
                    UserId = NetworkUtil.getCookie(GET_ID_URI, "", "id");
                } catch (IOException e) {
                    // 証明書エラーなどの例外をキャッチする
                    extMessage = e.toString();
                }

                // 取り寄せた ID を SharedPreferences に保存する。
                sp.edit().putString(ID_KEY, UserId).commit();
            }
            return UserId;
        }

        @Override
```

(continues on next page)

(continued from previous page)

```
        protected void onPostExecute(final String data) {
            String status = (data != null) ? "success" : "error";
            Toast.makeText(MainActivity.this, this.getClass().getSimpleName() + " - " + status + " : " +␣
↪extMessage, Toast.LENGTH_SHORT).show();
        }
    }

    private class SendDataAsyncTack extends AsyncTask<String, Void, Boolean> {
        private String extMessage = "";

        @Override
        protected Boolean doInBackground(String... params) {
            String url = params[0];
            String id = params[1];
            String nickname = params.length > 2 ? params[2] : null;
            String imei = params.length > 3 ? params[3] : null;

            Boolean result = false;
            try {
                JSONObject jsonData = new JSONObject();
                jsonData.put(ID_KEY, id);

                if (nickname != null)
                    jsonData.put(NICK_NAME_KEY, nickname);

                if (imei != null)
                    jsonData.put(IMEI_KEY, imei);

                NetworkUtil.sendJSON(url, "", jsonData.toString());

                result = true;
            } catch (IOException e) {
                // 証明書エラーなどの例外をキャッチする
                extMessage = e.toString();
            } catch (JSONException e) {
                extMessage = e.toString();
            }
            return result;
        }

        @Override
        protected void onPostExecute(Boolean result) {
            String status = result ? "Success" : "Error";
            Toast.makeText(MainActivity.this, this.getClass().getSimpleName() + " - " + status + " : " +␣
↪extMessage, Toast.LENGTH_SHORT).show();
        }
    }
}
```

ConfirmFragment.java

```
package org.jssec.android.privacypolicynopreconfirm;

import android.app.Activity;
import android.app.AlertDialog;
```

(continues on next page)

(continued from previous page)

```
import android.app.Dialog;
import android.content.Context;
import android.content.DialogInterface;
import android.content.Intent;
import android.os.Bundle;
import android.support.v4.app.DialogFragment;
import android.view.LayoutInflater;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.TextView;

public class ConfirmFragment extends DialogFragment {

    private DialogListener mListener = null;


    public static interface DialogListener {
        public void onPositiveButtonClick(int type);

        public void onNegativeButtonClick(int type);
    }

    public static ConfirmFragment newInstance(int title, int sentence, int type) {
        ConfirmFragment fragment = new ConfirmFragment();
        Bundle args = new Bundle();
        args.putInt("title", title);
        args.putInt("sentence", sentence);
        args.putInt("type", type);
        fragment.setArguments(args);
        return fragment;
    }

    @Override
    public Dialog onCreateDialog(Bundle args) {
        // ★ポイント 1★ 初回起動時に、アプリが扱う利用者情報の送信について包括同意を得る
        final int title = getArguments().getInt("title");
        final int sentence = getArguments().getInt("sentence");
        final int type = getArguments().getInt("type");

        LayoutInflater inflater = (LayoutInflater) getActivity().getSystemService(Context.LAYOUT_
↪INFLATER_SERVICE);
        View content = inflater.inflate(R.layout.fragment_comfirm, null);
        TextView linkPP = (TextView) content.findViewById(R.id.tx_link_pp);
        linkPP.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                // ★ポイント 3★ ユーザーがアプリ・プライバシーポリシーを確認できる手段を用意する
                Intent intent = new Intent();
                intent.setClass(getActivity(), WebViewAssetsActivity.class);
                startActivity(intent);
            }
        });

        AlertDialog.Builder builder = new AlertDialog.Builder(getActivity());
        builder.setIcon(R.drawable.ic_launcher);
```

(continues on next page)

(continued from previous page)

```
        builder.setTitle(title);
        builder.setMessage(sentence);
        builder.setView(content);

        builder.setPositiveButton(R.string.buttonOK, new DialogInterface.OnClickListener() {
            public void onClick(DialogInterface dialog, int whichButton) {
                if (mListener != null) {
                    mListener.onPositiveButtonClick(type);
                }
            }
        });
        builder.setNegativeButton(R.string.buttonNG, new DialogInterface.OnClickListener() {
            public void onClick(DialogInterface dialog, int whichButton) {
                if (mListener != null) {
                    mListener.onNegativeButtonClick(type);
                }
            }
        });

        Dialog dialog = builder.create();
        dialog.setCanceledOnTouchOutside(false);

        return dialog;
    }

    @Override
    public void onAttach(Activity activity) {
        super.onAttach(activity);
        if (!(activity instanceof DialogListener)) {
            throw new ClassCastException(activity.toString() + " must implement DialogListener.");
        }
        mListener = (DialogListener) activity;
    }

    public void setDialogListener(DialogListener listener) {
        mListener = listener;
    }
}
```

WebViewAssetsActivity.java

```
package org.jssec.android.privacypolicynopreconfirm;

import org.jssec.android.privacypolicynopreconfirm.R;

import android.app.Activity;
import android.os.Bundle;
import android.webkit.WebSettings;
import android.webkit.WebView;

public class WebViewAssetsActivity extends Activity {
    // ★ポイント 7★ アプリ・プライバシーポリシー概要版を assets フォルダ内に配置しておく
    private final String ABST_PP_URL = "file:///android_asset/PrivacyPolicy/app-policy-abst-
↪privacypolicy-1.0.html";
```

(continues on next page)

(continued from previous page)

```
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_webview);

        WebView webView = (WebView) findViewById(R.id.webView);
        WebSettings webSettings = webView.getSettings();

        webSettings.setAllowFileAccess(false);

        webView.loadUrl(ABST_PP_URL);
    }
}
```

5.5.1.3 [包括同意なし] アプリ・プライバシーポリシーを組み込んだアプリ

ポイント：[包括同意なし] アプリ・プライバシーポリシーを組み込んだアプリ

1. ユーザーがアプリ・プライバシーポリシーを確認できる手段を用意する
2. 送信した情報をユーザー操作により削除する手段を用意する
3. ユーザー操作により利用者情報の送信を停止する手段を用意する
4. 利用者情報の紐づけには UUID/cookie を利用する
5. アプリ・プライバシーポリシー概要版を assets フォルダ内に配置しておく

MainActivity.java

```
package org.jssec.android.privacypolicynocomprehensive;

import java.io.IOException;
import org.json.JSONException;
import org.json.JSONObject;

import android.os.AsyncTask;
import android.os.Bundle;
import android.content.Intent;
import android.content.SharedPreferences;
import android.support.v4.app.FragmentActivity;
import android.text.Editable;
import android.text.TextWatcher;
import android.view.Menu;
import android.view.MenuItem;
import android.view.View;
import android.widget.TextView;
import android.widget.Toast;

public class MainActivity extends FragmentActivity {
    private static final String BASE_URL = "https://www.example.com/pp";
    private static final String GET_ID_URI = BASE_URL + "/get_id.php";
    private static final String SEND_DATA_URI = BASE_URL + "/send_data.php";
    private static final String DEL_ID_URI = BASE_URL + "/del_id.php";
```

(continues on next page)

(continued from previous page)

```
    private static final String ID_KEY = "id";
    private static final String NICK_NAME_KEY = "nickname";

    private static final String PRIVACY_POLICY_PREF_NAME = "privacypolicy_preference";

    private String UserId = "";

    private TextWatcher watchHandler = new TextWatcher() {

        @Override
        public void beforeTextChanged(CharSequence s, int start, int count, int after) {
        }

        @Override
        public void onTextChanged(CharSequence s, int start, int before, int count) {
            boolean buttonEnable = (s.length() > 0);

            MainActivity.this.findViewById(R.id.buttonStart).setEnabled(buttonEnable);
        }

        @Override
        public void afterTextChanged(Editable s) {
        }
    };

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        // ユーザー識別用 ID をサーバーから取得する
        new GetDataAsyncTask().execute();

        findViewById(R.id.buttonStart).setEnabled(false);
        ((TextView) findViewById(R.id.editTextNickname)).addTextChangedListener(watchHandler);
    }

    public void onSendToServer(View view) {
        String nickname = ((TextView) findViewById(R.id.editTextNickname)).getText().toString();
        Toast.makeText(MainActivity.this, this.getClass().getSimpleName() + "\n - nickname : " +␣
↪nickname, Toast.LENGTH_SHORT).show();
        new sendDataAsyncTack().execute(SEND_DATA_URI, UserId, nickname);
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.main, menu);
        return true;
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        switch (item.getItemId()) {
        case R.id.action_show_pp:
            // ★ポイント 1★ ユーザーがアプリ・プライバシーポリシーを確認できる手段を用意する
```

(continues on next page)

(continued from previous page)

```
            Intent intent = new Intent();
            intent.setClass(this, WebViewAssetsActivity.class);
            startActivity(intent);
            return true;
        case R.id.action_del_id:
            // ★ポイント 2★ 送信した情報をユーザー操作により削除する手段を用意する
            new sendDataAsyncTack().execute(DEL_ID_URI, UserId);
            return true;
        case R.id.action_donot_send_id:
            // ★ポイント 3★ ユーザー操作により利用者情報の送信を停止する手段を用意する

            // 本サンプルでは利用者情報を送信しない場合、ユーザーに提供する機能が無くなるため
            // この段階でアプリを終了する。この処理はアプリ毎の都合に合わせて変更すること。
            String message  = getString(R.string.stopSendUserData);
            Toast.makeText(MainActivity.this, this.getClass().getSimpleName() + " - " + message, Toast.
↪LENGTH_SHORT).show();
            finish();

            return true;
        }
        return false;
    }


    private class GetDataAsyncTask extends AsyncTask<String, Void, String> {
        private String extMessage = "";

        @Override
        protected String doInBackground(String... params) {
            // ★ポイント 4★ 利用者情報の紐づけには UUID/cookie を利用する
            // 本サンプルではサーバー側で生成した ID を利用する
            SharedPreferences sp = getSharedPreferences(PRIVACY_POLICY_PREF_NAME, MODE_PRIVATE);
            UserId = sp.getString(ID_KEY, null);
            if (UserId == null) {
                // SharedPreferences 内にトークンが存在しなため、サーバーから ID を取り寄せる。
                try {
                    UserId = NetworkUtil.getCookie(GET_ID_URI, "", "id");
                } catch (IOException e) {
                    // 証明書エラーなどの例外をキャッチする
                    extMessage = e.toString();
                }

                // 取り寄せた ID を SharedPreferences に保存する。
                sp.edit().putString(ID_KEY, UserId).commit();
            }
            return UserId;
        }

        @Override
        protected void onPostExecute(final String data) {
            String status = (data != null) ? "success" : "error";
            Toast.makeText(MainActivity.this, this.getClass().getSimpleName() + " - " + status + " : " +␣
↪extMessage, Toast.LENGTH_SHORT).show();
        }
    }
```

(continues on next page)

(continued from previous page)

```
    private class sendDataAsyncTack extends AsyncTask<String, Void, Boolean> {
        private String extMessage = "";

        @Override
        protected Boolean doInBackground(String... params) {
            String url = params[0];
            String id = params[1];
            String nickname = params.length > 2 ? params[2] : null;

            Boolean result = false;
            try {
                JSONObject jsonData = new JSONObject();
                jsonData.put(ID_KEY, id);

                if (nickname != null)
                    jsonData.put(NICK_NAME_KEY, nickname);

                NetworkUtil.sendJSON(url, "", jsonData.toString());

                result = true;
            } catch (IOException e) {
                // 証明書エラーなどの例外をキャッチする
                extMessage = e.toString();
            } catch (JSONException e) {
                extMessage = e.toString();
            }
            return result;
        }

        @Override
        protected void onPostExecute(Boolean result) {
            String status = result ? "Success" : "Error";
            Toast.makeText(MainActivity.this, this.getClass().getSimpleName() + " - " + status + " : " +␣
↪extMessage, Toast.LENGTH_SHORT).show();
        }
    }
}
```

WebViewAssetsActivity.java

```
package org.jssec.android.privacypolicynocomprehensive;

import org.jssec.android.privacypolicynocomprehensive.R;

import android.app.Activity;
import android.os.Bundle;
import android.webkit.WebSettings;
import android.webkit.WebView;

public class WebViewAssetsActivity extends Activity {
    // ★ポイント 5★ アプリ・プライバシーポリシー概要版を assets フォルダ内に配置しておく
    private static final String ABST_PP_URL = "file:///android_asset/PrivacyPolicy/app-policy-abst-
↪privacypolicy-1.0.html";
```

(continues on next page)

(continued from previous page)

```
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_webview);

        WebView webView = (WebView) findViewById(R.id.webView);
        WebSettings webSettings = webView.getSettings();

        webSettings.setAllowFileAccess(false);

        webView.loadUrl(ABST_PP_URL);
    }
}
```

5.5.1.4 アプリ・プライバシーポリシーを組み込まないアプリ

ポイント：アプリ・プライバシーポリシーを組み込まないアプリ

1. 取得した情報を端末内部でのみ利用する場合、アプリ・プライバシーポリシーを表示しなくても良い
2. マーケットプレイス等のアプリ説明欄に、取得した情報を外部送信しない旨を記載する

MainActivity.java

```
package org.jssec.android.privacypolicynoinfosent;

import com.google.android.gms.common.ConnectionResult;
import com.google.android.gms.common.GooglePlayServicesClient;
import com.google.android.gms.location.LocationClient;

import android.location.Location;
import android.net.Uri;
import android.os.Bundle;
import android.content.Intent;
import android.content.IntentSender;
import android.support.v4.app.FragmentActivity;
import android.view.Menu;
import android.view.View;
import android.widget.TextView;
import android.widget.Toast;

public class MainActivity extends FragmentActivity implements GooglePlayServicesClient.
↪ConnectionCallbacks, GooglePlayServicesClient.OnConnectionFailedListener {
    private LocationClient mLocationClient = null;

    private final int CONNECTION_FAILURE_RESOLUTION_REQUEST = 257;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        mLocationClient = new LocationClient(this, this, this);
    }
```

(continues on next page)

(continued from previous page)

```
    @Override
    protected void onStart() {
        super.onStart();

        // Location 情報取得用
        if (mLocationClient != null) {
            mLocationClient.connect();
        }
    }

    @Override
    protected void onStop() {
        if (mLocationClient != null) {
            mLocationClient.disconnect();
        }
        super.onStop();
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.main, menu);
        return true;
    }

    public void onStartMap(View view) {
        // ★ポイント 1★ 取得した情報を端末内部でのみ利用する場合、アプリ・プライバシーポリシーを表示しなくても良い
        if (mLocationClient != null && mLocationClient.isConnected()) {
            Location currentLocation = mLocationClient.getLastLocation();
            if (currentLocation != null) {
                Intent intent = new Intent(Intent.ACTION_VIEW, Uri.parse("geo:" + currentLocation.
↪getLatitude() + "," + currentLocation.getLongitude()));
                startActivity(intent);
            }
        }
    }

    @Override
    public void onConnected(Bundle connectionHint) {
        if (mLocationClient != null && mLocationClient.isConnected()) {
            Location currentLocation = mLocationClient.getLastLocation();
            if (currentLocation != null) {
                String locationData = "Latitude \t: " + currentLocation.getLatitude() + "\n\tLongitude␣
↪\t: " + currentLocation.getLongitude();

                String text = "\n" + getString(R.string.your_location_title) + "\n\t" + locationData;

                Toast.makeText(MainActivity.this, this.getClass().getSimpleName() + text, Toast.LENGTH_
↪SHORT).show();

                TextView appText = (TextView) findViewById(R.id.appText);
                appText.setText(text);
            }
        }
    }
```

(continues on next page)

(continued from previous page)

```
    @Override
    public void onConnectionFailed(ConnectionResult result) {
        if (result.hasResolution()) {
            try {
                result.startResolutionForResult(this, CONNECTION_FAILURE_RESOLUTION_REQUEST);
            } catch (IntentSender.SendIntentException e) {
                e.printStackTrace();
            }
        }
    }

    @Override
    public void onDisconnected() {
        mLocationClient = null;
        Toast.makeText(this, "Disconnected. Please re-connect.", Toast.LENGTH_SHORT).show();
    }
}
```

★ポイント 2 ★マーケットプレイス等のアプリ説明欄に、取得した情報を外部送信しない旨を記載する

図 5.5.3　マーケットプレイス上での説明例

5.5.2 ルールブック

プライバシー情報を扱う際には以下のルールを守ること。

1. 送信する利用者情報は必要最低限に留める （必須）
2. 初回起動時 (アップデート時) に、ユーザーによる取り換えが困難、または、慎重な取り扱いが求められる利用者情報の送信について包括同意を得る （必須）
3. 慎重な取り扱いが求められる利用者情報を送信する場合は、ユーザーに個別同意を得る （必須）

4. ユーザーがアプリ・プライバシーポリシーを確認できる手段を用意する （必須）
5. アプリ・プライバシーポリシー概要版を *assets* フォルダ内に配置しておく （推奨）
6. 送信した利用者情報をユーザー操作により削除および送信停止する手段を用意する （推奨）
7. 端末固有 *ID* と *UUID/cookie* を使い分ける （推奨）
8. 利用者情報を端末内のみで利用する場合、外部送信しない旨をユーザーに通知する （推奨）

5.5.2.1 送信する利用者情報は必要最低限に留める （必須）

アプリマニュアルなどの説明を元にユーザーが想起できるアプリの動作および目的以外で利用者情報にアクセスしないよう設計すること。また、アプリの動作および目的に必要な利用者情報であっても、アプリ内部でのみ必要な情報と外部サーバー等に送信すべき情報を精査し、必要最低限の情報のみを送信すること。

例えば、アラームアプリで、指定した時間にアラームが鳴る機能しか提供していないにもかかわらず位置情報をサーバーに送信している場合、ユーザーは位置情報とアプリの機能を関連付けることは難しい。一方で、このアラームアプリが、ユーザーがいる場所に応じて設定されたアラームを鳴らす機能を持つ場合は、アプリが位置情報を利用する理由を説明することができる。このように、ユーザーが機能と利用者情報との関連を理解できる説明が可能であれば、利用者情報の利用に妥当性があるとみなせる。

5.5.2.2 初回起動時 (アップデート時) に、ユーザーによる取り換えが困難、または、慎重な取り扱いが求められる利用者情報の送信について包括同意を得る （必須）

ユーザーによる取り換えが困難な利用者情報や慎重な取り扱いが求められる利用者情報を外部サーバーに送信する場合、ユーザーがアプリを使用する前に、アプリがどのような情報をどのような目的でサーバーに送信するか、第三者提供はあるかなどについて、事前にユーザーの同意 (オプトイン) を得ることが求められる。具体的には、アプリの初回起動時にアプリ・プライバシーポリシーを提示して、ユーザーの確認と同意を得るようにするべきである。また、アプリケーションのアップデートにより、新たな利用者情報を外部サーバーに送信するようになった時にも、再度ユーザーの確認と同意を得るようにする必要がある。同意が得られない場合は、アプリを終了するなど、情報の送信が必要な機能 (処理) を無効にすること。

これにより、ユーザーが利用者情報の利用状況を理解した上でアプリを使用していることが保証され、ユーザーに安心感を提供するとともにアプリへの信頼感を得ることが期待できる。

MainActivity.java

```
protected void onStart() {
    super.onStart();

    // ～省略～
    if (privacyPolicyAgreed <= VERSION_TO_SHOW_COMPREHENSIVE_AGREEMENT_ANEW) {
        // ★ポイント★ 初回起動時（アップデート時）に、ユーザーによる取り換えが困難、または、慎重な取り扱いが求められる利用者情報の送信について包括同意を得る
        // アップデート時については、新しい利用者情報を扱うようになった場合にのみ、再度包括同意を得る必要がある。
        ConfirmFragment dialog = ConfirmFragment.newInstance(
        R.string.privacyPolicy, R.string.agreePrivacyPolicy,
        DIALOG_TYPE_COMPREHENSIVE_AGREEMENT);
        dialog.setDialogListener(this);
        FragmentManager fragmentManager = getSupportFragmentManager();
        dialog.show(fragmentManager, "dialog");
    }
```

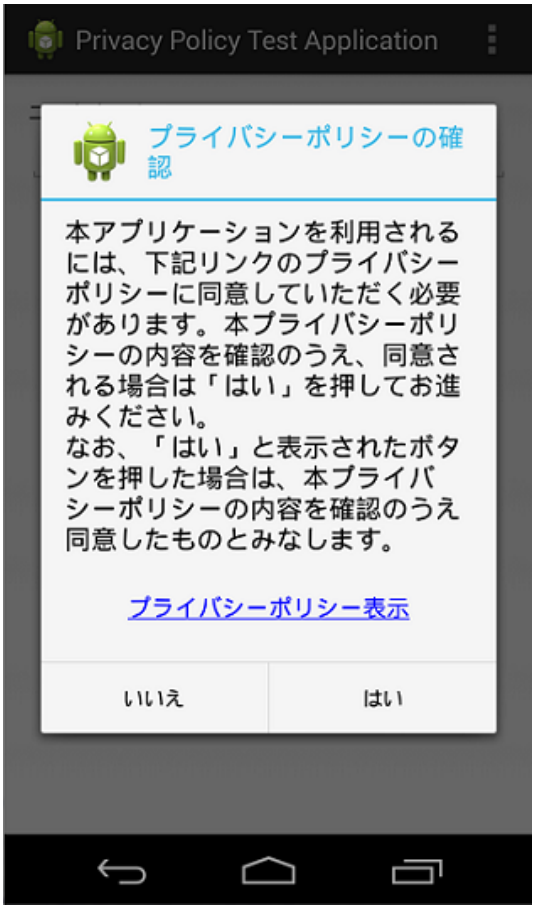

図 5.5.4　包括同意の例

5.5.2.3 慎重な取り扱いが求められる利用者情報を送信する場合は、ユーザーに個別同意を得る （必須）

慎重な取扱いが求められる利用者情報を外部サーバーに送信する場合は、包括同意に加えて、利用者情報 (または利用者情報の送信を伴う機能) 毎に、事前にユーザーの同意 (オプトイン) を得ることが必要である。ユーザーの同意が得られなかった場合は、外部サーバーへの情報送信は実施してはならない。

これにより、ユーザーは包括同意で確認した利用者情報の送信に関して、より具体的なアプリ機能 (提供サービス) との関連を知ることができ、アプリ提供者はユーザーのより正確な判断による同意を得ることが期待できる。

```
MainActivity.java
public void onSendToServer(View view) {
    // ★ポイント★慎重な取り扱いが求められる利用者情報を送信する場合は、個別にユーザーの同意を得る
    ConfirmFragment dialog =
        ConfirmFragment.newInstance(R.string.sendLocation, R.string.cofirmSendLocation, DIALOG_TYPE_PRE_
↪CONFIRMATION);
    dialog.setDialogListener(this);
    FragmentManager fragmentManager = getSupportFragmentManager();
    dialog.show(fragmentManager, "dialog");
}
```

図 5.5.5　個別同意の例

5.5.2.4 ユーザーがアプリ・プライバシーポリシーを確認できる手段を用意する （必須）

一般に、ユーザーが該当アプリをインストールする前にアプリ・プライバシーポリシーを確認する手段として、Android アプリのマーケットプレイス上にアプリ・プライバシーポリシーのリンクを表示する機能がある。この機能への対応に加え、アプリを端末にインストールした後でも、ユーザーがアプリ・プライバシーポリシーを参照できる手段を用意すること。特に利用者情報を外部サーバー等に送信する場合の個別同意確認時にはアプリ・プライバシーポリシーを容易に確認できる手段を用意し、ユーザーが正しく判断できるようにすることが望ましい。

```
MainActivity.java
    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        switch (item.getItemId()) {
        case R.id.action_show_pp:
            // ★ポイント★ ユーザーがアプリ・プライバシーポリシーを確認できる手段を用意する
            Intent intent = new Intent();
            intent.setClass(this, WebViewAssetsActivity.class);
            startActivity(intent);
            return true;
```

図 5.5.6　プライバシーポリシー表示用メニュー

5.5.2.5 アプリ・プライバシーポリシー概要版を **assets** フォルダ内に配置しておく （推奨）

アプリ・プライバシーポリシーの概要版は assets フォルダ内に配置しておき、必要に応じて閲覧に利用することが望ましい。アプリ・プライバシーポリシーが assets フォルダにあれば、いつでも容易にアクセス可能であると同時に、悪意のある第三者により、偽造や改造されたアプリ・プライバシーポリシーを参照させられるリスクが回避可能なためである。また、assets フォルダに配置するファイルは、利用者情報に関する第三者検証が可能であることが望ましい。本節で紹介した「アプリケーション・プライバシーポリシー作成支援ツール」を利用する場合、検査用タグのある XML ファイルも assets フォルダに配置することで第三者検証が可能になる。

5.5.2.6 送信した利用者情報をユーザー操作により削除および送信停止する手段を用意する （推奨）

利用者の求めに応じ、外部サーバー上に送信された利用者情報を削除する機能を提供することが望ましい。同様に、アプリ自身で端末内に利用者情報（もしくはそのコピー）を保存した場合も、その情報を削除する機能をユーザーに提供することが望ましい。また、利用者の求めに応じ、利用者情報の送信を停止する機能を提供することが望ましい。

本ルール（推奨）の内、利用者情報の削除については、EU で提唱されている『忘れられる権利』により定められたものであるが、今後は利用者データ保護に関する個人の権利の強化も提案されているため、特に理由がない限り、本ガイドでは利用者情報の削除機能を用意することを推奨する。また、利用者情報の送信停止については、主にブラウザによる対応が進んでいる『Do Not Track（追跡拒否）』の観点により定められたものであり、削除同様に送信停止機能を用意するこ

とを推奨する。

```
MainActivity.java
    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        switch (item.getItemId()) {
        // ～省略～
        case R.id.action_del_id:
            // ★ポイント★ 送信した情報をユーザー操作により削除する手段を用意する
            new SendDataAsyncTack().execute(DEL_ID_URI, UserId);
            return true;
        }
```

5.5.2.7 端末固有 ID と UUID/cookie を使い分ける （推奨）

IMEI などの端末固有 ID は、利用者情報と紐付けて送信すべきでない。端末固有 ID と利用者情報が紐付いた形で一度でも公開や漏洩してしまうと、後から端末固有 ID の変更が不可能なため、ID と利用者情報の紐づけを切ることができない（難しい）ことが、その理由である。この場合、端末固有 ID に変わって、UUID/cookie などの乱数をベースにした都度作成する変更可能な ID を使って、利用者情報との紐づけおよび送信をすると良い。これにより、上記で説明した『忘れられる権利』を考慮した実装とすることができる。

```
MainActivity.java
    @Override
    protected String doInBackground(String... params) {
        // ★ポイント★ 利用者情報の紐づけには UUID/cookie を利用する
        // 本サンプルではサーバー側で生成した ID を利用する
        SharedPreferences sp = getSharedPreferences(PRIVACY_POLICY_PREF_NAME, MODE_PRIVATE);
        UserId = sp.getString(ID_KEY, null);
        if (UserId == null) {
            // SharedPreferences 内にトークンが存在しなため、サーバーから ID を取り寄せる。
            try {
                UserId = NetworkUtil.getCookie(GET_ID_URI, "", "id");
            } catch (IOException e) {
                // 証明書エラーなどの例外をキャッチする
                extMessage = e.toString();
            }

            // 取り寄せた ID を SharedPreferences に保存する。
            sp.edit().putString(ID_KEY, UserId).commit();

        return UserId;
    }
```

5.5.2.8 利用者情報を端末内のみで利用する場合、外部送信しない旨をユーザーに通知する （推奨）

利用者の端末内部で一時的に利用者情報にアクセスするのみの場合であっても、利用者の理解を助け透明性を高めるために、その旨を伝えることが望ましい。具体的には、アクセスした利用者情報は、ある決まった目的のために端末内部で一時的に使用するのみであり、端末内に保存したり、外部サーバに送信したりしない旨をを利用者に通知すると良い。通知方法としては、マーケットプレイス上でのアプリ説明欄に記載するなどの方法が考えられる。なお、端末内での一時利用のみの場合は、アプリ・プライバシーポリシーへの記載は必須ではない。

図 5.5.7　マーケットプレイス上での説明例

5.5.3 アドバンスト

5.5.3.1 プライバシーポリシーを取りまく状況

スマートフォンにおける利用者情報を取得し、外部に送信する場合には、利用者に対してアプリがどのような情報をどのように扱うか等を示したアプリ・プライバシーポリシーを作成・提示する必要がある。アプリ・プライバシーポリシーに記載すべき内容は、総務省 SPI にて定義されている。

アプリ・プライバシーポリシーは個々のアプリが扱うすべての利用者情報と、その用途や情報の保存先、送信先等を明らかにすることに主眼が置かれている。それとは別に事業者が個々のアプリから収集したすべての利用者情報をどのように保管・管理・廃棄をするかを示す事業者プライバシーポリシーも必要となる。従来、個人情報保護法をもとに作成されているプライバシーポリシーとはこの事業者プライバシーポリシーが相当する。

プライバシーポリシーの作成・提示方法やそれぞれのプラバシーポリシーの役割分担などは、『JSSEC スマホ・アプリのプライバシーポリシー作成・開示についての考察』(https://www.jssec.org/event/20140206/03-1_app_policy.pdf）に詳細説明が記載されているので参照のこと。

5.5.3.2 用語解説

以下に、本ガイドで使用している用語について、『JSSEC スマホ・アプリのプライバシーポリシー作成・開示についての考察』(https://www.jssec.org/event/20140206/03-1_app_policy.pdf) に記載されている用語解説を引用しておく。

表 5.5.1: 用語解説

用語	解説
事業者プライバシーポリシー	事業者における個人情報保護方針を記したプライバシーポリシー。個人情報保護法に基づき作成。
アプリ・プライバシーポリシー	アプリケーション向けプライバシーポリシー。総務省 SPI の指針に基づき作成。概要版と詳細版での解りやすい説明が望まれる。
概要版アプリ・プライバシーポリシー	アプリが取得する利用者情報、取得目的、第三者提供の有無などを簡潔に記述した文書。
詳細版アプリ・プライバシーポリシー	総務省 SPI の定める８項目に準拠した詳細な内容を記述した文書。
利用者による取り換えが容易な利用者情報	cookie、UUID など。
利用者による取り換えが困難な利用者情報	IMEI、IMSI、ICCID、MAC アドレス、OS が生成する ID など。
慎重な取り扱いが求められる利用者情報	位置情報、アドレス帳、電話番号、メールアドレスなど。

5.5.3.3 Android ID のバージョンによる違いについて

Android ID(Settings.Secure.ANDROID_ID) は、ランダムに生成される 64 ビットの数値を 16 進数文字列で表現したものであり、(ごく稀に重複する可能性はあるが) 端末を個別に識別することができる識別子である。そのため、使用法を間違えるとユーザーのトラッキングに繋がるリスクが大きくなり、使用の際には注意を必要とするが、Android 7.1(API Level 25) 以前と Android 8.0(API Level26) 以降の端末では、生成規則やアクセス可能範囲などが異なるため以下ではその違いについて説明する。

Android 7.1(API Level 25) 以前の端末

Android 7.1(API Level 25) 以前の端末では端末内に 1 つの値を持ちすべてのアプリからアクセス可能となる。ただし、マルチユーザーをサポートする端末ではユーザー毎に別の値が生成される。生成のタイミングとしては、最初は端末の工場出荷後の初回起動時であり、その後は、ファクトリーリセットの度に新しく生成される。

Android 8.0(API Level 26) 以降の端末

Android 8.0(API Level 26) 以降の端末ではアプリ (開発者) 毎に異なる値を持ち、対象のアプリだけがアクセスできるように変更されている。具体的には、Android 7.1(API Level 25) 以前のユーザーおよび端末に加えて、アプリの署名を要素として値が一意に決定される仕組みとなっており、署名の異なるアプリは別の (署名が同じ場合は同じ)Android ID の値を持つことになる。

値が変更されるタイミングに関しては以前とほぼ変わらないが、いくつか注意する点があるので以下に示す。

- パッケージのアンインストール・再インストール時

Android ID の値は、署名が同じである限り、アプリをアンインストール・再インストールされても変わらない。逆に、署名する鍵が変わった場合は、パッケージ名が同じでも再インストール時に Android ID の値が異なることに注意する。

- Android 8.0(API Level 26) 以降の端末へのアップデート時

Android 7.1(API Level 25) 以前の端末で既にアプリがインストールされていた場合、端末を Android 8.0(API Level 26) 以降にアップデートしてもアプリで取得できる Android ID の値は変わらない。ただし、アップデート後にアプリがアンインストール・再インストールする場合は除く。

なお、いずれの Android ID でも「5.5.3.2. 用語解説」における「利用者による取り換えが困難な利用者情報」に分類されるため、冒頭に言及したように使用の際は同様の注意を払うことを推奨する。

5.6 暗号技術を利用する

セキュリティの世界では、脅威を分析し対策するための観点として、機密性 (Confidentiality)、完全性 (Integrity)、可用性 (Availavility) という用語が使われる。それぞれ秘密のデータを第三者に見られないようにすること、参照するデータが改ざんされないように (または、改ざんされたことを検知) すること、サービスやデータが必要な時にいつでも利用できることを意味しており、セキュリティを確保する際に考慮すべき大切な要素である。中でも、機密性や完全性のために暗号技術が用いられることが多く、Android においてもアプリが機密性や完全性を実現するための様々な暗号機能が用意されている。

ここでは、Android アプリにおける暗号化・復号 (機密性の実現)、メッセージ認証コード/デジタル署名 (完全性の実現) の安全な実装方法をサンプルコードで示す。

5.6.1 サンプルコード

「暗号化・復号する (機密性を確保する)」、「改ざんを検知する (完全性を確認する)」というユースケースの具体的な用途や条件に対して、様々な暗号方式が開発されている。ここでは、暗号技術をどのような用途で利用するかという観点から、暗号方式を大きく 3 つに分類してサンプルコードを用意している。それぞれの暗号技術の特徴から利用すべき暗号方式・鍵の種類を判断することができる。より細やかな判断が必要な場合には、「5.6.3.1. 暗号方式の選択」も参照すること。

また、暗号技術を利用する実装をする際には、「5.6.3.3. 乱数生成における脆弱性と対策」もあらかじめ参照すること。

- 第三者の盗聴からデータを守る

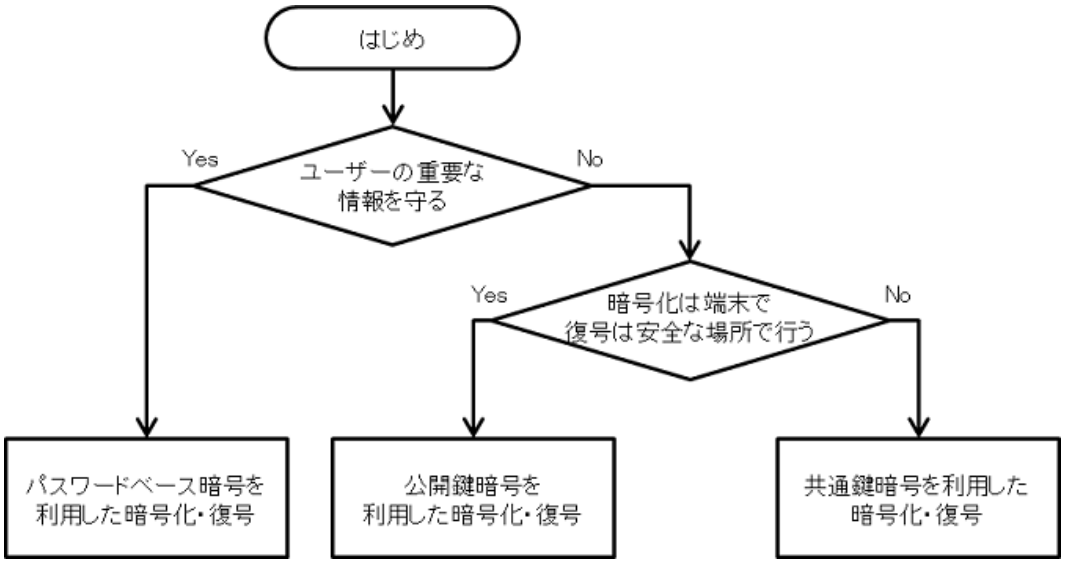

図 5.6.1　盗聴からデータを守るサンプルコードを選択するフローチャート

- 第三者によるデータの改ざんを検知する

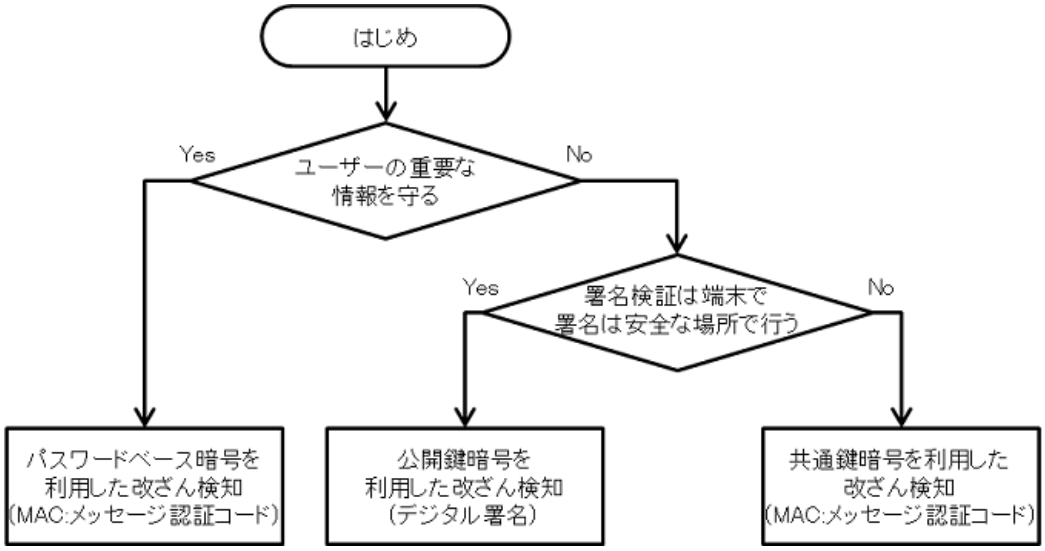

図 5.6.2　データの改ざんを検知するサンプルコードを選択するフローチャート

5.6.1.1 パスワード鍵を利用して暗号化・復号する

ユーザーの情報資産の秘匿性を守る目的にはパスワードベース鍵暗号を使うことができる。

ポイント：

1. 明示的に暗号モードとパディングを設定する
2. 脆弱でない (基準を満たす) 暗号技術（アルゴリズム・モード・パディング等）を使用する
3. パスワードから鍵を生成する場合は、Salt を使用する
4. パスワードから鍵を生成する場合は、適正なハッシュの繰り返し回数を指定する
5. 十分安全な長さを持つ鍵を利用する

AesCryptoPBEKey.java

```
package org.jssec.android.cryptsymmetricpasswordbasedkey;

import java.security.InvalidAlgorithmParameterException;
import java.security.InvalidKeyException;
import java.security.NoSuchAlgorithmException;
import java.security.SecureRandom;
import java.security.spec.InvalidKeySpecException;
import java.util.Arrays;

import javax.crypto.BadPaddingException;
import javax.crypto.Cipher;
import javax.crypto.IllegalBlockSizeException;
import javax.crypto.NoSuchPaddingException;
import javax.crypto.SecretKey;
import javax.crypto.SecretKeyFactory;
import javax.crypto.spec.IvParameterSpec;
import javax.crypto.spec.PBEKeySpec;

public final class AesCryptoPBEKey {

    // ★ポイント 1★ 明示的に暗号モードとパディングを設定する
```

(continues on next page)

(continued from previous page)

```
    // ★ポイント 2★ 脆弱でない（基準を満たす）暗号技術（アルゴリズム・モード・パディング等）を使用する
    // Cipher クラスの getInstance に渡すパラメータ （/[暗号アルゴリズム]/[ブロック暗号モード]/[パディングルール]）
    // サンプルでは、暗号アルゴリズム=AES、ブロック暗号モード=CBC、パディングルール=PKCS7Padding
    private static final String TRANSFORMATION = "AES/CBC/PKCS7Padding";

    // 鍵を生成するクラスのインスタンスを取得するための文字列
    private static final String KEY_GENERATOR_MODE = "PBEWITHSHA256AND128BITAES-CBC-BC";

    // ★ポイント 3★ パスワードから鍵を生成する場合は、Salt を使用する
    // Salt のバイト長
    public static final int SALT_LENGTH_BYTES = 20;

    // ★ポイント 4★ パスワードから鍵を生成する場合は、適正なハッシュの繰り返し回数を指定する
    // PBE で鍵を生成する際の攪拌の繰り返し回数
    private static final int KEY_GEN_ITERATION_COUNT = 1024;

    // ★ポイント 5★ 十分安全な長さを持つ鍵を利用する
    // 鍵のビット長
    private static final int KEY_LENGTH_BITS = 128;

    private byte[] mIV = null;
    private byte[] mSalt = null;

    public byte[] getIV() {
        return mIV;
    }

    public byte[] getSalt() {
        return mSalt;
    }

    AesCryptoPBEKey(final byte[] iv, final byte[] salt) {
        mIV = iv;
        mSalt = salt;
    }

    AesCryptoPBEKey() {
        mIV = null;
        initSalt();
    }

    private void initSalt() {
        mSalt = new byte[SALT_LENGTH_BYTES];
        SecureRandom sr = new SecureRandom();
        sr.nextBytes(mSalt);
    }

    public final byte[] encrypt(final byte[] plain, final char[] password) {
        byte[] encrypted = null;

        try {
            // ★ポイント 1★ 明示的に暗号モードとパディングを設定する
            // ★ポイント 2★ 脆弱でない（基準を満たす）暗号技術（アルゴリズム・モード・パディング等）を使用する
            Cipher cipher = Cipher.getInstance(TRANSFORMATION);
```

(continues on next page)

(continued from previous page)

```
            // ★ポイント 3★ パスワードから鍵を生成する場合は、Salt を使用する
            SecretKey secretKey = generateKey(password, mSalt);
            cipher.init(Cipher.ENCRYPT_MODE, secretKey);
            mIV = cipher.getIV();

            encrypted = cipher.doFinal(plain);
        } catch (NoSuchAlgorithmException e) {
        } catch (NoSuchPaddingException e) {

        } catch (InvalidKeyException e) {
        } catch (IllegalBlockSizeException e) {
        } catch (BadPaddingException e) {
        } finally {
        }

        return encrypted;
    }

    public final byte[] decrypt(final byte[] encrypted, final char[] password) {
        byte[] plain = null;

        try {
            // ★ポイント 1★ 明示的に暗号モードとパディングを設定する
            // ★ポイント 2★ 脆弱でない (基準を満たす) 暗号技術 (アルゴリズム・モード・パディング等) を使用する
            Cipher cipher = Cipher.getInstance(TRANSFORMATION);

            // ★ポイント 3★ パスワードから鍵を生成する場合は、Salt を使用する
            SecretKey secretKey = generateKey(password, mSalt);
            IvParameterSpec ivParameterSpec = new IvParameterSpec(mIV);
            cipher.init(Cipher.DECRYPT_MODE, secretKey, ivParameterSpec);

            plain = cipher.doFinal(encrypted);
        } catch (NoSuchAlgorithmException e) {
        } catch (NoSuchPaddingException e) {
        } catch (InvalidKeyException e) {
        } catch (InvalidAlgorithmParameterException e) {
        } catch (IllegalBlockSizeException e) {
        } catch (BadPaddingException e) {
        } finally {
        }

        return plain;
    }

    private static final SecretKey generateKey(final char[] password, final byte[] salt) {
        SecretKey secretKey = null;
        PBEKeySpec keySpec = null;

        try {
            // ★ポイント 2★ 脆弱でない (基準を満たす) アルゴリズム・モード・パディングを使用する
            // 鍵を生成するクラスのインスタンスを取得する
            // 例では、AES-CBC 128 ビット用の鍵を SHA256 を利用して生成する KeyFactory を使用。
            SecretKeyFactory secretKeyFactory = SecretKeyFactory.getInstance(KEY_GENERATOR_MODE);

            // ★ポイント 3★ パスワードから鍵を生成する場合は、Salt を使用する
```

(continues on next page)

(continued from previous page)

```
            // ★ポイント 4★ パスワードから鍵を生成する場合は、適正なハッシュの繰り返し回数を指定する
            // ★ポイント 5★ 十分安全な長さを持つ鍵を利用する
            keySpec = new PBEKeySpec(password, salt, KEY_GEN_ITERATION_COUNT, KEY_LENGTH_BITS);
            // password のクリア
            Arrays.fill(password, '?');
            // 鍵を生成する
            secretKey = secretKeyFactory.generateSecret(keySpec);
        } catch (NoSuchAlgorithmException e) {
        } catch (InvalidKeySpecException e) {
        } finally {
            keySpec.clearPassword();
        }

        return secretKey;
    }
}
```

5.6.1.2 公開鍵を利用して暗号化・復号する

アプリ側では公開鍵を保持してデータの暗号化のみを行い、復号を異なる安全な場所 (サーバーなど) で秘密鍵によって行うような用途には公開鍵 (非対称鍵) 暗号を使うことができる。

ポイント：

1. 明示的に暗号モードとパディングを設定する
2. 脆弱でない (基準を満たす) 暗号技術（アルゴリズム・モード・パディング等）を使用する
3. 十分安全な長さを持つ鍵を利用する

RsaCryptoAsymmetricKey.java

```
package org.jssec.android.cryptasymmetrickey;

import java.security.InvalidKeyException;
import java.security.KeyFactory;
import java.security.NoSuchAlgorithmException;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.interfaces.RSAPublicKey;
import java.security.spec.InvalidKeySpecException;
import java.security.spec.PKCS8EncodedKeySpec;
import java.security.spec.X509EncodedKeySpec;

import javax.crypto.BadPaddingException;
import javax.crypto.Cipher;
import javax.crypto.IllegalBlockSizeException;
import javax.crypto.NoSuchPaddingException;

public final class RsaCryptoAsymmetricKey {

    // ★ポイント 1★ 明示的に暗号モードとパディングを設定する
    // ★ポイント 2★ 脆弱でない (基準を満たす) 暗号技術（アルゴリズム・モード・パディング等）を使用する
    // Cipher クラスの getInstance に渡すパラメータ （/[暗号アルゴリズム]/[ブロック暗号モード]/[パディングルール]）
    // サンプルでは、暗号アルゴリズム=RSA、ブロック暗号モード=NONE、パディングルール=OAEPPADDING
```

(continues on next page)

(continued from previous page)

```
    private static final String TRANSFORMATION = "RSA/NONE/OAEPPADDING";

    // 暗号アルゴリズム
    private static final String KEY_ALGORITHM = "RSA";

    // ★ポイント 3★ 十分安全な長さを持つ鍵を利用する
    // 鍵長チェック
    private static final int MIN_KEY_LENGTH = 2000;

    RsaCryptoAsymmetricKey() {
    }

    public final byte[] encrypt(final byte[] plain, final byte[] keyData) {
        byte[] encrypted = null;

        try {
            // ★ポイント 1★ 明示的に暗号モードとパディングを設定する
            // ★ポイント 2★ 脆弱でない（基準を満たす）暗号技術（アルゴリズム・モード・パディング等）を使用する
            Cipher cipher = Cipher.getInstance(TRANSFORMATION);

            PublicKey publicKey = generatePubKey(keyData);
            if (publicKey != null) {
                cipher.init(Cipher.ENCRYPT_MODE, publicKey);
                encrypted = cipher.doFinal(plain);
            }
        } catch (NoSuchAlgorithmException e) {
        } catch (NoSuchPaddingException e) {
        } catch (InvalidKeyException e) {
        } catch (IllegalBlockSizeException e) {
        } catch (BadPaddingException e) {
        } finally {
        }

        return encrypted;
    }

    public final byte[] decrypt(final byte[] encrypted, final byte[] keyData) {
        // 本来、復号処理はサーバー側で実装すべきものであるが、
        // 本サンプルでは動作確認用に、アプリ内でも復号処理を実装した。
        // 実際にサンプルコードを利用する場合は、アプリ内に秘密鍵を保持しないようにすること。

        byte[] plain = null;

        try {
            // ★ポイント 1★ 明示的に暗号モードとパディングを設定する
            // ★ポイント 2★ 脆弱でない（基準を満たす）暗号技術（アルゴリズム・モード・パディング等）を使用する
            Cipher cipher = Cipher.getInstance(TRANSFORMATION);

            PrivateKey privateKey = generatePriKey(keyData);
            cipher.init(Cipher.DECRYPT_MODE, privateKey);

            plain = cipher.doFinal(encrypted);
        } catch (NoSuchAlgorithmException e) {
        } catch (NoSuchPaddingException e) {
        } catch (InvalidKeyException e) {
```

(continues on next page)

(continued from previous page)

```
        } catch (IllegalBlockSizeException e) {
        } catch (BadPaddingException e) {
        } finally {
        }

        return plain;
    }

    private static final PublicKey generatePubKey(final byte[] keyData) {

        PublicKey publicKey = null;
        KeyFactory keyFactory = null;

        try {
            keyFactory = KeyFactory.getInstance(KEY_ALGORITHM);
            publicKey = keyFactory.generatePublic(new X509EncodedKeySpec(keyData));
        } catch (IllegalArgumentException e) {
        } catch (NoSuchAlgorithmException e) {
        } catch (InvalidKeySpecException e) {
        } finally {
        }

        // ★ポイント 3★ 十分安全な長さを持つ鍵を利用する
        // 鍵長のチェック
        if (publicKey instanceof RSAPublicKey) {
            int len = ((RSAPublicKey) publicKey).getModulus().bitLength();
            if (len < MIN_KEY_LENGTH) {
                publicKey = null;
            }
        }

        return publicKey;
    }

    private static final PrivateKey generatePriKey(final byte[] keyData) {
        PrivateKey privateKey = null;
        KeyFactory keyFactory = null;

        try {
            keyFactory = KeyFactory.getInstance(KEY_ALGORITHM);
            privateKey = keyFactory.generatePrivate(new PKCS8EncodedKeySpec(keyData));
        } catch (IllegalArgumentException e) {
        } catch (NoSuchAlgorithmException e) {
        } catch (InvalidKeySpecException e) {
        } finally {
        }

        return privateKey;
    }
}
```

5.6.1.3 共通鍵を利用して暗号化・復号する

サイズの大きなデータを扱う場合やアプリの資産・ユーザーの資産の秘匿性を守る目的で使うことができる。

ポイント：

1. 明示的に暗号モードとパディングを設定する
2. 脆弱でない (基準を満たす) 暗号技術（アルゴリズム・モード・パディング等）を使用する
3. 十分安全な長さを持つ鍵を利用する

AesCryptoPreSharedKey.java

```
package org.jssec.android.cryptsymmetricpresharedkey;

import java.security.InvalidAlgorithmParameterException;
import java.security.InvalidKeyException;
import java.security.NoSuchAlgorithmException;

import javax.crypto.BadPaddingException;
import javax.crypto.Cipher;
import javax.crypto.IllegalBlockSizeException;
import javax.crypto.NoSuchPaddingException;
import javax.crypto.SecretKey;
import javax.crypto.spec.IvParameterSpec;
import javax.crypto.spec.SecretKeySpec;

public final class AesCryptoPreSharedKey {

    // ★ポイント 1★ 明示的に暗号モードとパディングを設定する
    // ★ポイント 2★ 脆弱でない (基準を満たす) 暗号技術（アルゴリズム・モード・パディング等）を使用する
    // Cipher クラスの getInstance に渡すパラメータ （/[暗号アルゴリズム]/[ブロック暗号モード]/[パディングルール]）
    // サンプルでは、暗号アルゴリズム=AES、ブロック暗号モード=CBC、パディングルール=PKCS7Padding
    private static final String TRANSFORMATION = "AES/CBC/PKCS7Padding";

    // 暗号アルゴリズム
    private static final String KEY_ALGORITHM = "AES";


    // IVのバイト長
    public static final int IV_LENGTH_BYTES = 16;

    // ★ポイント 3★ 十分安全な長さを持つ鍵を利用する
    // 鍵長チェック
    private static final int MIN_KEY_LENGTH_BYTES = 16;

    private byte[] mIV = null;

    public byte[] getIV() {
        return mIV;
    }

    AesCryptoPreSharedKey(final byte[] iv) {
        mIV = iv;
    }

    AesCryptoPreSharedKey() {
    }

    public final byte[] encrypt(final byte[] keyData, final byte[] plain) {
        byte[] encrypted = null;
```

(continues on next page)

(continued from previous page)

```
    try {
        // ★ポイント 1★ 明示的に暗号モードとパディングを設定する
        // ★ポイント 2★ 脆弱でない（基準を満たす）暗号技術（アルゴリズム・モード・パディング等）を使用する
        Cipher cipher = Cipher.getInstance(TRANSFORMATION);

        SecretKey secretKey = generateKey(keyData);
        if (secretKey != null) {
            cipher.init(Cipher.ENCRYPT_MODE, secretKey);
            mIV = cipher.getIV();

            encrypted = cipher.doFinal(plain);
        }
    } catch (NoSuchAlgorithmException e) {
    } catch (NoSuchPaddingException e) {
    } catch (InvalidKeyException e) {
    } catch (IllegalBlockSizeException e) {
    } catch (BadPaddingException e) {
    } finally {
    }

    return encrypted;
}

public final byte[] decrypt(final byte[] keyData, final byte[] encrypted) {
    byte[] plain = null;

    try {
        // ★ポイント 1★ 明示的に暗号モードとパディングを設定する
        // ★ポイント 2★ 脆弱でない（基準を満たす）暗号技術（アルゴリズム・モード・パディング等）を使用する
        Cipher cipher = Cipher.getInstance(TRANSFORMATION);

        SecretKey secretKey = generateKey(keyData);
        if (secretKey != null) {
            IvParameterSpec ivParameterSpec = new IvParameterSpec(mIV);
            cipher.init(Cipher.DECRYPT_MODE, secretKey, ivParameterSpec);

            plain = cipher.doFinal(encrypted);
        }
    } catch (NoSuchAlgorithmException e) {
    } catch (NoSuchPaddingException e) {
    } catch (InvalidKeyException e) {
    } catch (InvalidAlgorithmParameterException e) {
    } catch (IllegalBlockSizeException e) {
    } catch (BadPaddingException e) {
    } finally {
    }

    return plain;
}

private static final SecretKey generateKey(final byte[] keyData) {
    SecretKey secretKey = null;

    try {
```

(continues on next page)

(continued from previous page)

```
            // ★ポイント 3★ 十分安全な長さを持つ鍵を利用する
            if (keyData.length >= MIN_KEY_LENGTH_BYTES) {
                // ★ポイント 2★ 脆弱でない (基準を満たす) 暗号技術（アルゴリズム・モード・パディング等）を使用する
                secretKey = new SecretKeySpec(keyData, KEY_ALGORITHM);
            }
        } catch (IllegalArgumentException e) {
        } finally {
        }


        return secretKey;
    }
}
```

5.6.1.4 パスワード鍵を利用して改ざんを検知する

ユーザーの情報資産の完全性をチェックする目的にはパスワードベース (共通鍵) 暗号を使うことができる。

ポイント：

1. 明示的に暗号モードとパディングを設定する
2. 脆弱でない (基準を満たす) 暗号技術（アルゴリズム・モード・パディング等）を使用する
3. パスワードから鍵を生成する場合は、Salt を使用する
4. パスワードから鍵を生成する場合は、適正なハッシュの繰り返し回数を指定する
5. 十分安全な長さを持つ鍵を利用する

HmacPBEKey.java

```
package org.jssec.android.signsymmetricpasswordbasedkey;

import java.security.InvalidKeyException;
import java.security.NoSuchAlgorithmException;
import java.security.SecureRandom;
import java.security.spec.InvalidKeySpecException;
import java.util.Arrays;

import javax.crypto.Mac;
import javax.crypto.SecretKey;
import javax.crypto.SecretKeyFactory;
import javax.crypto.spec.PBEKeySpec;

public final class HmacPBEKey {
    // ★ポイント 1★ 明示的に暗号モードとパディングを設定する
    // ★ポイント 2★ 脆弱でない (基準を満たす) 暗号技術（アルゴリズム・モード・パディング等）を使用する
    // Mac クラスの getInstance に渡すパラメータ （認証モード）
    private static final String TRANSFORMATION = "PBEWITHHMACSHA1";

    // 鍵を生成するクラスのインスタンスを取得するための文字列
    private static final String KEY_GENERATOR_MODE = "PBEWITHHMACSHA1";

    // ★ポイント 3★ パスワードから鍵を生成する場合は、Salt を使用する
```

(continues on next page)

(continued from previous page)

```
    // Salt のバイト長
    public static final int SALT_LENGTH_BYTES = 20;

    // ★ポイント 4★ パスワードから鍵を生成する場合は、適正なハッシュの繰り返し回数を指定する
    // PBE で鍵を生成する際の撹拌の繰り返し回数
    private static final int KEY_GEN_ITERATION_COUNT = 1024;

    // ★ポイント 5★ 十分安全な長さを持つ鍵を利用する
    // 鍵のビット長
    private static final int KEY_LENGTH_BITS = 160;

    private byte[] mSalt = null;

    public byte[] getSalt() {
        return mSalt;
    }

    HmacPBEKey() {
        initSalt();
    }

    HmacPBEKey(final byte[] salt) {
        mSalt = salt;
    }

    private void initSalt() {
        mSalt = new byte[SALT_LENGTH_BYTES];
        SecureRandom sr = new SecureRandom();
        sr.nextBytes(mSalt);
    }

    public final byte[] sign(final byte[] plain, final char[] password) {
        return calculate(plain, password);
    }

    private final byte[] calculate(final byte[] plain, final char[] password) {
        byte[] hmac = null;

        try {
            // ★ポイント 1★ 明示的に暗号モードとパディングを設定する
            // ★ポイント 2★ 脆弱でない（基準を満たす）暗号技術（アルゴリズム・モード・パディング等）を使用する
            Mac mac = Mac.getInstance(TRANSFORMATION);

            // ★ポイント 3★ パスワードから鍵を生成する場合は、Salt を使用する
            SecretKey secretKey = generateKey(password, mSalt);
            mac.init(secretKey);

            hmac = mac.doFinal(plain);
        } catch (NoSuchAlgorithmException e) {
        } catch (InvalidKeyException e) {
        } finally {
        }

        return hmac;
    }
```

(continues on next page)

(continued from previous page)

```
    public final boolean verify(final byte[] hmac, final byte[] plain, final char[] password) {

        byte[] hmacForPlain = calculate(plain, password);

        if (Arrays.equals(hmac, hmacForPlain)) {
            return true;
        }
        return false;
    }

    private static final SecretKey generateKey(final char[] password, final byte[] salt) {
        SecretKey secretKey = null;
        PBEKeySpec keySpec = null;

        try {
            // ★ポイント 2★ 脆弱でない (基準を満たす) アルゴリズム・モード・パディングを使用する
            // 鍵を生成するクラスのインスタンスを取得する
            // 例では、AES-CBC 128 ビット用の鍵を SHA1 を利用して生成する KeyFactory を使用。
            SecretKeyFactory secretKeyFactory = SecretKeyFactory.getInstance(KEY_GENERATOR_MODE);

            // ★ポイント 3★ パスワードから鍵を生成する場合は、Salt を使用する
            // ★ポイント 4★ パスワードから鍵を生成する場合は、適正なハッシュの繰り返し回数を指定する
            // ★ポイント 5★ 十分安全な長さを持つ鍵を利用する
            keySpec = new PBEKeySpec(password, salt, KEY_GEN_ITERATION_COUNT, KEY_LENGTH_BITS);
            // password のクリア
            Arrays.fill(password, '?');
            // 鍵を生成する
            secretKey = secretKeyFactory.generateSecret(keySpec);
        } catch (NoSuchAlgorithmException e) {
        } catch (InvalidKeySpecException e) {
        } finally {
            keySpec.clearPassword();
        }

        return secretKey;
    }

}
```

5.6.1.5 公開鍵を利用して改ざんを検知する

異なる安全な場所 (サーバーなど) で秘密鍵による署名を行ったデータに対して、アプリ側では公開鍵を保持してデータの署名検証のみを行うような用途には公開鍵 (非対称鍵) 暗号を使うことができる。

ポイント：

1. 明示的に暗号モードとパディングを設定する
2. 脆弱でない (基準を満たす) 暗号技術（アルゴリズム・モード・パディング等）を使用する
3. 十分安全な長さを持つ鍵を利用する

RsaSignAsymmetricKey.java

```
package org.jssec.android.signasymmetrickey;

import java.security.InvalidKeyException;
import java.security.KeyFactory;

import java.security.NoSuchAlgorithmException;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.Signature;
import java.security.SignatureException;
import java.security.interfaces.RSAPublicKey;
import java.security.spec.InvalidKeySpecException;
import java.security.spec.PKCS8EncodedKeySpec;
import java.security.spec.X509EncodedKeySpec;

public final class RsaSignAsymmetricKey {

    // ★ポイント 1★ 明示的に暗号モードとパディングを設定する
    // ★ポイント 2★ 脆弱でない (基準を満たす) 暗号技術 (アルゴリズム・モード・パディング等) を使用する
    // Cipher クラスの getInstance に渡すパラメータ (/[暗号アルゴリズム]/[ブロック暗号モード]/[パディングルール])
    // サンプルでは、暗号アルゴリズム=RSA、ブロック暗号モード=NONE、パディングルール=OAEPPADDING
    private static final String TRANSFORMATION = "SHA256withRSA";

    // 暗号アルゴリズム
    private static final String KEY_ALGORITHM = "RSA";

    // ★ポイント 3★ 十分安全な長さを持つ鍵を利用する
    // 鍵長チェック
    private static final int MIN_KEY_LENGTH = 2000;

    RsaSignAsymmetricKey() {
    }

    public final byte[] sign(final byte[] plain, final byte[] keyData) {
        // 本来、署名処理はサーバー側で実装すべきものであるが、
        // 本サンプルでは動作確認用に、アプリ内でも署名処理を実装した。
        // 実際にサンプルコードを利用する場合は、アプリ内に秘密鍵を保持しないようにすること。

        byte[] sign = null;

        try {
            // ★ポイント 1★ 明示的に暗号モードとパディングを設定する
            // ★ポイント 2★ 脆弱でない (基準を満たす) 暗号技術 (アルゴリズム・モード・パディング等) を使用する
            Signature signature = Signature.getInstance(TRANSFORMATION);

            PrivateKey privateKey = generatePriKey(keyData);
            signature.initSign(privateKey);
            signature.update(plain);

            sign = signature.sign();
        } catch (NoSuchAlgorithmException e) {
        } catch (InvalidKeyException e) {
        } catch (SignatureException e) {
        } finally {
        }
```

(continues on next page)

(continued from previous page)

```
    return sign;
}

public final boolean verify(final byte[] sign, final byte[] plain, final byte[] keyData) {

    boolean ret = false;

    try {
        // ★ポイント 1★ 明示的に暗号モードとパディングを設定する
        // ★ポイント 2★ 脆弱でない (基準を満たす) 暗号技術 (アルゴリズム・モード・パディング等) を使用する
        Signature signature = Signature.getInstance(TRANSFORMATION);

        PublicKey publicKey = generatePubKey(keyData);
        signature.initVerify(publicKey);
        signature.update(plain);

        ret = signature.verify(sign);

    } catch (NoSuchAlgorithmException e) {
    } catch (InvalidKeyException e) {
    } catch (SignatureException e) {
    } finally {
    }

    return ret;
}

private static final PublicKey generatePubKey(final byte[] keyData) {
    PublicKey publicKey = null;
    KeyFactory keyFactory = null;

    try {
        keyFactory = KeyFactory.getInstance(KEY_ALGORITHM);
        publicKey = keyFactory.generatePublic(new X509EncodedKeySpec(keyData));

    } catch (IllegalArgumentException e) {
    } catch (NoSuchAlgorithmException e) {
    } catch (InvalidKeySpecException e) {
    } finally {
    }

    // ★ポイント 3★ 十分安全な長さを持つ鍵を利用する
    // 鍵長のチェック
    if (publicKey instanceof RSAPublicKey) {
        int len = ((RSAPublicKey) publicKey).getModulus().bitLength();
        if (len < MIN_KEY_LENGTH) {
            publicKey = null;
        }
    }

    return publicKey;
}

private static final PrivateKey generatePriKey(final byte[] keyData) {
```

(continues on next page)

(continued from previous page)

```
        PrivateKey privateKey = null;
        KeyFactory keyFactory = null;

        try {
            keyFactory = KeyFactory.getInstance(KEY_ALGORITHM);
            privateKey = keyFactory.generatePrivate(new PKCS8EncodedKeySpec(keyData));
        } catch (IllegalArgumentException e) {
        } catch (NoSuchAlgorithmException e) {
        } catch (InvalidKeySpecException e) {
        } finally {
        }

        return privateKey;
    }
}
```

5.6.1.6 共通鍵を利用して改ざんを検知する

アプリの資産・ユーザーの資産の完全性をチェックする目的で使うことができる。

ポイント：

1. 明示的に暗号モードとパディングを設定する
2. 脆弱でない (基準を満たす) 暗号技術（アルゴリズム・モード・パディング等）を使用する
3. 十分安全な長さを持つ鍵を利用する

HmacPreSharedKey.java

```
package org.jssec.android.signsymmetricpresharedkey;

import java.security.InvalidKeyException;
import java.security.NoSuchAlgorithmException;
import java.util.Arrays;

import javax.crypto.Mac;
import javax.crypto.SecretKey;
import javax.crypto.spec.SecretKeySpec;

public final class HmacPreSharedKey {

    // ★ポイント 1★ 明示的に暗号モードとパディングを設定する
    // ★ポイント 2★ 脆弱でない（基準を満たす）暗号技術（アルゴリズム・モード・パディング等）を使用する
    // Mac クラスの getInstance に渡すパラメータ （認証モード）
    private static final String TRANSFORMATION = "HmacSHA256";

    // 暗号アルゴリズム
    private static final String KEY_ALGORITHM = "HmacSHA256";

    // ★ポイント 3★ 十分安全な長さを持つ鍵を利用する
    // 鍵長チェック
    private static final int MIN_KEY_LENGTH_BYTES = 16;

    HmacPreSharedKey() {
```

(continues on next page)

(continued from previous page)

```
    }

    public final byte[] sign(final byte[] plain, final byte[] keyData) {
        return calculate(plain, keyData);
    }


    public final byte[] calculate(final byte[] plain, final byte[] keyData) {
        byte[] hmac = null;
        int count = 0;

        try {
            // ★ポイント 1★ 明示的に暗号モードとパディングを設定する
            // ★ポイント 2★ 脆弱でない（基準を満たす）暗号技術（アルゴリズム・モード・パディング等）を使用する
            Mac mac = Mac.getInstance(TRANSFORMATION);

            SecretKey secretKey = generateKey(keyData);
            if (secretKey != null) {
                mac.init(secretKey);

                hmac = mac.doFinal(plain);

                StringBuilder sb = new StringBuilder();

                for (int i = 0; i < hmac.length; i++) {
                    //System.out.println(Integer.toHexString(hmac[i] & 0xff));
                    sb.append(Integer.toHexString(hmac[i] & 0xff));
                    count++;
                }

                count = 0;
            }
        } catch (NoSuchAlgorithmException e) {
        } catch (InvalidKeyException e) {
        } finally {
        }

        return hmac;
    }

    public final boolean verify(final byte[] hmac, final byte[] plain, final byte[] keyData) {
        byte[] hmacForPlain = calculate(plain, keyData);

        if (hmacForPlain != null && Arrays.equals(hmac, hmacForPlain)) {
            return true;
        }

        return false;
    }

    private static final SecretKey generateKey(final byte[] keyData) {
        SecretKey secretKey = null;

        try {
            // ★ポイント 3★ 十分安全な長さを持つ鍵を利用する
```

(continues on next page)

(continued from previous page)

```
            if (keyData.length >= MIN_KEY_LENGTH_BYTES) {
                // ★ポイント 2★ 脆弱でない (基準を満たす) 暗号技術 (アルゴリズム・モード・パディング等) を使用する
                secretKey = new SecretKeySpec(keyData, KEY_ALGORITHM);
            }
        } catch (IllegalArgumentException e) {
        } finally {
        }

        return secretKey;
    }
}
```

5.6.2 ルールブック

暗号技術を利用する際には以下のルールを守ること。

1. 暗号を指定する場合は、明示的に暗号モードとパディングを設定する （必須）
2. 脆弱でないアルゴリズム（基準を満たすもの）を使用する （必須）
3. パスワードベース暗号のパスワードを端末内に保存しないこと （必須）
4. パスワードから鍵を生成する場合は、*Salt* を使用する （必須）
5. パスワードから鍵を生成する場合は、適正なハッシュの繰り返し回数を指定する （必須）
6. パスワードの強度を高める工夫をする （推奨）

5.6.2.1 暗号を指定する場合は、明示的に暗号モードとパディングを設定する （必須）

暗号化やデータ検証などで暗号技術を利用する場合、明示的に暗号モードとパディングを設定すること。Android アプリの開発で暗号技術を利用する場合、主に java.crypto のクラスである Cipher クラスを利用する。Cipher クラスを利用する際、初めにどのような暗号を利用するかを指定して Cipher クラスのオブジェクトを生成する。この指定を Transformation と呼ぶが、Transformation の指定の書式には以下の 2 種類が存在する

- ”algorithm/mode/padding”
- ”algorithm”

後者の場合、暗号モードとパディングは、Android で利用可能な暗号化サービスプロバイダ固有のデフォルト値が使用される。このデフォルト値は、利便性や互換性などを優先し、あまり安全でないものが設定されていることがあり、セキュリティを確保するためには必ず、前者の暗号モードとパディングを指定する書式を利用する必要がある。

5.6.2.2 脆弱でないアルゴリズム（基準を満たすもの）を使用する （必須）

暗号技術を利用する場合、アルゴリズムとして脆弱でない、一定の基準を満たしたものを使用すること。また、アルゴリズムとして複数の鍵長を許容する場合、アプリの製品ライフタイムを考慮して十分安全な鍵長を利用すること。さらには、暗号モードやパディングモードについても既知の攻撃方法が存在するものもあり、脆弱でない十分安全な方式を利用すること。

脆弱な暗号技術を利用した場合、例えば第三者からの盗聴を防ぐために暗号化したファイルでも、有効な保護になっておらず、第三者の盗聴を許してしまう為である。IT 技術の進化に伴い、暗号解析の技術も向上されるため、利用する技術はアプリが稼働することを想定する期間中、安全が保護されることが期待できるアルゴリズムを検討し選択する必要がある。

実際の暗号技術の基準としては、以下のように各国の基準が利用できるため参考にすること。

表 5.6.1　NIST（米国） NIST SP800-57

アルゴリズムのライフタイム	対称鍵暗号	非対称暗号	楕円暗号	**HASH (**デジタル署名**, HASH)**	**HASH (HMA, KD,** 乱数生成**)**
～2010	80	1024	160	160	160
～2030	112	2048	224	224	160
2030～	128	3072	256	256	160

単位：bit

表 5.6.2: ECRYPT II (EU)

アルゴリズムのライフタイム	対称鍵暗号	非対称暗号	楕円暗号	**HASH**
2009～2012	80	1248	160	160
2009～2020	96	1776	192	192
2009～2030	112	2432	224	224
2009～2040	128	3248	256	256
2009~	256	15424	512	512

単位：bit

表 5.6.3　CRYPTREC（日本） 電子政府推奨暗号リスト

技術分類		名称
公開鍵暗号	署名	DSA,ECDSA,RSA=PSS,RSASSA=PKCS1=V1_5
	守秘	RSA-OAEP
	鍵共有	DH,ECDH
共通鍵暗号	64 ビットブロック暗号	3-key Triple DES
	128 ビットブロック暗号	AES,Camellia
	ストリーム暗号	KCipher-2
ハッシュ関数		SHA-256,SHA-384,SHA-512
暗号利用モード	秘匿モード	CBC,CFB,CTR,OFB
	認証付き秘匿モード	CCM,GCM
メッセージ認証コード		CMAC,HMAC
エンティティ認証		ISO/IEC 9798-2,ISO/IEC 9798-3

5.6.2.3 パスワードベース暗号のパスワードを端末内に保存しないこと （必須）

パスワードベース暗号で、ユーザーから入力を受けたパスワードをベースに暗号鍵を生成する場合、入力されたパスワードを端末内に保存しないこと。パスワードベース暗号の利点は暗号鍵の管理を不要とすることであり、端末内にパスワー

ドを保存してしまうと、その利点が得られなくなるためである。もちろん、端末内にパスワードを保存することで、他のアプリから盗聴されてしまうリスクが生じることもあり、安全面から考えても端末内への保存はしてはならない。

5.6.2.4 パスワードから鍵を生成する場合は、Salt を使用する （必須）

パスワードベース暗号で、ユーザーから入力を受けたパスワードをベースに暗号鍵を生成する場合、必ず Salt を使用すること。また、端末内の別ユーザーに機能を提供する場合は、ユーザー毎に異なる Salt を利用すること。Salt を利用せずに単純なハッシュ関数のみで暗号鍵を生成した場合、レインボーテーブルと呼ばれる技術を使うことで、容易に元のパスワードを類推することができるためである。Salt を付けた場合、同じパスワードから生成した鍵であっても異なる鍵（ハッシュ値）が生成されるようになり、レインボーテーブルによる鍵の探索を妨害することができる。

パスワードから鍵を生成する場合に、Salt を使用する例

```
public final byte[] encrypt(final byte[] plain, final char[] password) {
    byte[] encrypted = null;

    try {
        // ★ポイント★ 明示的に暗号モードとパディングを設定する
        // ★ポイント★ 脆弱でない（基準を満たす）暗号技術（アルゴリズム・モード・パディング等）を使用する
        Cipher cipher = Cipher.getInstance(TRANSFORMATION);

        // ★ポイント★ パスワードから鍵を生成する場合は、Salt を使用する
        SecretKey secretKey = generateKey(password, mSalt);
```

5.6.2.5 パスワードから鍵を生成する場合は、適正なハッシュの繰り返し回数を指定する （必須）

パスワードベース暗号で、ユーザーから入力を受けたパスワードをベースに暗号鍵を生成する場合、鍵生成で適用するハッシュ処理を十分安全な回数繰り返す（ストレッチングする）こと。一般に 1,000 回以上の繰り返しであれば良いとされる。さらに重要な資産を保護するために利用する場合は 1,000,000 回以上の繰り返しを行うこと。ハッシュ関数は一回の計算にかかる時間が非常に短時間であるため、攻撃者による総当たり攻撃が容易になる原因にもなっている。そのため、ハッシュ処理を繰り返し用いるストレッチング手法で、わざと時間がかかるようにして総当たり攻撃を困難にする。なお、ストレッチング回数はアプリの処理速度にも影響を及ぼすため、設定する回数には注意すること。

パスワードから鍵を生成する場合に、ハッシュの繰り返し回数を指定する例

```
private static final SecretKey generateKey(final char[] password, final byte[] salt) {
    SecretKey secretKey = null;
    PBEKeySpec keySpec = null;

    // ～省略～

        // ★ポイント★ パスワードから鍵を生成する場合は、Salt を使用する
        // ★ポイント★ パスワードから鍵を生成する場合は、適正なハッシュの繰り返し回数を指定する
        // ★ポイント★ 十分安全な長さを持つ鍵を利用する
        keySpec = new PBEKeySpec(password, salt, KEY_GEN_ITERATION_COUNT, KEY_LENGTH_BITS);
```

5.6.2.6 パスワードの強度を高める工夫をする （推奨）

パスワードベース暗号で、ユーザーから入力を受けたパスワードをベースに暗号鍵を生成する場合、暗号鍵の強度はユーザーのパスワードの強度に直結するため、ユーザーから受け付けるパスワードの強度を高める工夫をすることが望まし

い。例えば、パスワードの最低長を 8 文字以上としたり、複数の文字種の利用を必須としたり (英数記号を各 1 文字以上など) するなどの方法が考えられる。

5.6.3 アドバンスト

5.6.3.1 暗号方式の選択

「サンプルコード」では、暗号化・復号、改ざん検知の各々について 3 つの暗号方式の実装例を示した。用途によってどの暗号方式を使用するかも、「図 5.6.1 盗聴からデータを守るサンプルコードを選択するフローチャート」、「図 5.6.2 データの改ざんを検知するサンプルコードを選択するフローチャート」を利用して大まかに判断することができる。一方で、暗号方式の選択は、その特徴を比較することでより細やかに判断することもできる。以下に比較を示す。

暗号化・復号における暗号方式の比較

公開鍵暗号は、処理コストが高いため大きなサイズのデータ処理には向かないが、暗号化と復号に使う鍵が異なるため、公開鍵のみアプリ側で扱い（暗号化のみ行い)、復号を別の（安全な）場所で行うような場合は鍵の管理が比較的容易である。共通鍵暗号は、制限の少ない万能な暗号方式であるが、暗号化・復号ともに同じ鍵を使用するため、アプリ内に鍵を安全に保持する必要があり、鍵の保護が難しい。パスワードベース暗号 (パスワードを元にした共通鍵暗号) は、ユーザーのパスワードから鍵を生成するため、端末内に鍵に関する秘密を保存する必要がない。用途としては、ユーザーの資産を保護する場合のみに使える。また、暗号の強度がパスワードの強度に依存するため、資産の保護レベルに応じてパスワードを複雑にする必要がある。「5.6.2.6. パスワードの強度を高める工夫をする （推奨)」も参照すること。

表 5.6.4: 暗号化・復号における暗号方式の比較

	公開鍵	共通鍵	パスワードベース
サイズの大きなデータの処理	NG(処理コストが高い)	OK	OK
アプリ (サービス) 資産の保護	OK	OK	NG(ユーザーによる盗聴が可能)
ユーザー資産の保護	OK	OK	OK
暗号の強度	鍵の長さに依存	鍵の長さに依存	パスワードの強度、Salt、ハッシュの繰り返し回数に依存
鍵の保護	容易 (公開鍵のみ)	困難	容易
アプリで行う処理	暗号化 (復号はサーバーなど)	暗号化・復号	暗号化・復号

改ざん検知における暗号方式の比較

データのサイズの項を除いた以外は、暗号化・復号における暗号方式の比較とほぼ同様である。

表 5.6.5: 改ざん検知における暗号方式の比較

	公開鍵	共通鍵	パスワードベース
アプリ (サービス) 資産の保護	OK	OK	NG(ユーザーによる改ざんが可能)
ユーザー資産の保護	OK	OK	OK

次のページに続く

表 5.6.5 – 前のページからの続き

暗号の強度	鍵の長さに依存	鍵の長さに依存	パスワードの強度、Salt、ハッシュの繰り返し回数に依存
鍵の保護	容易 (公開鍵のみ)	困難 (「5.6.3.4. 鍵の保護」参照)	容易
アプリで行う処理	署名検証 (署名はサーバーなど)	MAC 計算・MAC 検証	MAC 計算・MAC 検証

MAC：メッセージ認証コード

なお本ガイドでは、「3.1.3. 資産分類と保護施策」における資産レベル低中位の資産を主な保護対象としている。暗号の使用は鍵の管理問題など一般の保護施策 (アクセス制御など) と比べて多くの検討事項を伴うため、Android OS のセキュリティモデルでは資産の保護を充分にできない場合のみ、暗号の使用を検討することが望ましい。

5.6.3.2 乱数の生成

暗号技術の利用おいて、強固な暗号アルゴリズムや暗号モードを選択し、十分な長さの鍵を使用することは、アプリやサービスで扱うデータのセキュリティを確保する上で非常に重要である。しかし、これらの選択が適切であっても秘密の起点となる鍵が漏洩したり、予測されたりすると、アルゴリズム等で保証するセキュリティの強度が全く意味をなさなくなってしまう。また、AES 等の共通暗号で使用する初期化ベクトル (IV) やパスワードベース暗号で使用する Salt(ソルト) に関しても、偏りが大きいと第三者による攻撃が容易になり、情報の漏洩や改ざんなどの被害に繋がる可能性が高くなる。このような事態を防ぐには、第三者に鍵や IV の値の予測が困難な方法で生成する必要があり、それを実現するために極めて重要な役割を果たすのが乱数である。乱数を生成する装置は乱数生成器と呼ばれ、センサー等を用いて予測・再現が不可能とされる自然状態を観測することで乱数を生成するハードウェアの乱数生成器 (RNG) に対して、ソフトウェアで実現する乱数生成器を疑似乱数生成器 (PRNG) と呼ぶのが一般的である。

Android アプリでは、暗号用途での利用に対して十分セキュアな乱数を SecureRandom クラス経由で取得することができる。SecureRandom クラスの機能は、Provider と呼ばれる実装によって提供される。また、内部に複数の Provider(実装) を持つことが可能であり、Provider を明示的に指定しない場合はデフォルトの Provider が選ばれる。そのため、実装時に Provider の存在を意識しないで SecureRandom を使うことも可能である。以下に、SecureRandom の使い方の例を示す。

なお、SecureRandom には Android のバージョンによっていくつか脆弱性があり、実装上の対策が必要になる。「5.6.3.3. 乱数生成における脆弱性と対策」を参照すること。

SecureRandom の使用 (デフォルトの実装を使用する)

```
import java.security.SecureRandom;
// ～省略～
    SecureRandom random = new SecureRandom();
    byte[] randomBuf = new byte [128];

    random.nextBytes(randomBuf);
// ～省略～
```

SecureRandom の使用 (明示的にアルゴリズムを指定する)

```
import java.security.SecureRandom;
// ～省略～
    SecureRandom random = SecureRandom.getInstance("SHA1PRNG");
```

(continues on next page)

(continued from previous page)

```
    byte[] randomBuf = new byte [128];

    random.nextBytes(randomBuf);
// ～省略～
```

SecureRandom の使用 (明示的に実装 (Provider) を指定する)

```
import java.security.SecureRandom;
// ～省略～
    SecureRandom random = SecureRandom.getInstance("SHA1PRNG", "Crypto");
    byte[] randomBuf = new byte [128];

    random.nextBytes(randomBuf);
// ～省略～
```

SecureRandom のようなプログラムで実現されている疑似乱数生成器は一般に「図 5.6.3 疑似乱数生成器の内部プロセス」のように動作しており、乱数の種を入力して内部状態を初期化すると、乱数を生成する度に内部状態を一定のアルゴリズムで更新することで、次々と乱数列の生成が可能になる。

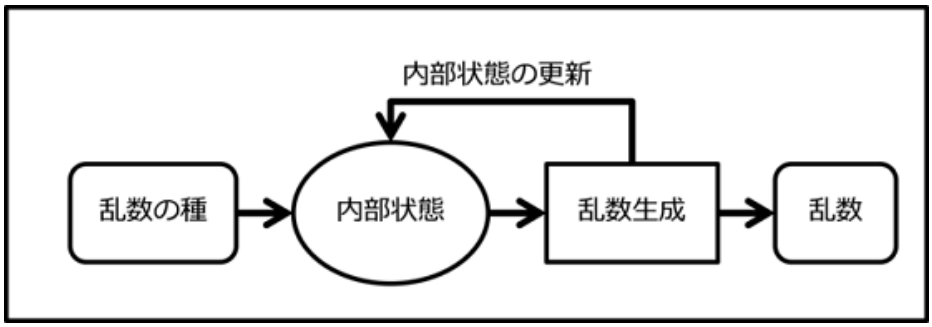

図 5.6.3　疑似乱数生成器の内部プロセス

乱数の種

疑似乱数生成器において乱数の種の役割は極めて重要である。

既に説明したように疑似乱数生成器は乱数の種による初期化が必要である。また、乱数の種で初期化された後は決められたアルゴリズムで乱数の生成が行われるので、同じ乱数の種からは同じ乱数列が生成されることになる。これは乱数の種が第三者に知られ (盗聴され) たり、予測可能であったりすると、第三者が同じ乱数列を取得することになり、乱数を起点にした機密性や完全性が失われることを意味している。

そのため、乱数の種はそれ自身機密性の高い情報であり、かつ予測困難でなければならない。例えば時刻情報や端末固有値 (MAC アドレス、IMEI、Android ID など) を乱数の種として使うべきではない。多くの Android 端末においては、/dev/urandom, /dev/random 等が利用可能であり、Android デフォルトで提供される SecureRandom の実装もそれらのデバイスファイルを使って乱数の種を設定している。また、機密性に関しては、乱数の種がメモリ内にのみ存在するならば、root 権限を取得したマルウェア・ツールでない限り、盗聴の危険性は低い。仮に、root 化された端末でも安全にする必要の場合は、セキュア設計・実装の専門家と相談して対応すること。

疑似乱数生成器の内部状態

疑似乱数生成器の内部状態は、乱数の種によって初期化され、乱数を生成する毎に状態が更新される。また、乱数の種と同様に、同じ内部状態の疑似乱数生成器が存在した場合は、その後に生成される乱数列はどちらもまったく同じものに

なる。よって、内部状態も第三者に対して、盗聴されないように気を付けなければならない。ただし、内部状態はメモリ内に存在するため、root 権限を取得したマルウェア・ツールでない限り、盗聴の危険性は低い。root 化された端末でも安全にする必要の場合は、セキュア設計・実装の専門家と相談して対応すること。

5.6.3.3 乱数生成における脆弱性と対策

Android 4.3.x 以前の"Crypto" Provider の SecureRandom 実装には、内部状態のエントロピー (Randomness) が充分に確保されないという不具合がある。特に Android 4.1.x 以前は、SecureRandom の実装が"Crypto" Provider しかなく、SecureRandom を直接・間接的に使用するほとんどのアプリはこの脆弱性の影響を受ける。また、Android 4.2 以降に SecureRandom のデフォルト実装となる"AndroidOpenSSL" Provider では、OpenSSL が「乱数の種」として使うデータの大部分がアプリ間で共有されるという不具合 (対象は Anroid 4.2.x-4.3.x) により、あるアプリが別のアプリの生成する乱数を推測しやすくなるという脆弱性を抱えている。以下に Android OS バージョンと各脆弱性の影響を受ける機能の整理しておく。

表 5.6.6: Android OS バージョンと各脆弱性の影響を受ける機能

	"Crypto" Provider SecureRandom の実装	別アプリの **OpenSSL** の乱数の種が推測可能
Android 4.1.x 以前	• SecureRandom のデフォルト実装 • Crypto Provider の明示的な使用 • Cipher クラスの提供する暗号機能 • HTTPS 通信機能 etc.	影響なし
Android 4.2 - 4.3.x	Crypto Provider の明示的な使用	• SecureRandom のデフォルト実装 • AndroidOpenSSL Provider の明示的な使用 • OpenSSL の乱数機能の直接使用 • Cipher クラスの提供する暗号機能 • HTTPS 通信機能 etc.
Android 4.4 以降	影響なし	影響なし

2013 年 8 月以降、これら Android OS の脆弱性を修正するパッチが Google からパートナー (端末メーカーなど) に配布されている。しかし、これら SecureRandom に関する脆弱性は暗号機能、HTTPS 通信機能を含めて広い範囲に影響する上に、パッチが適用されていない端末が多数存在することも考えられるため、Android 4.3.x 以前の OS を対象とするアプリでは、以下のサイトで紹介されている対策 (実装) を組み込んでおくことをお薦めする。

https://android-developers.blogspot.jp/2013/08/some-securerandom-thoughts.html

5.6.3.4 鍵の保護

センシティブな情報の安全性 (機密性、完全性) を実現するために暗号技術を利用する際に、堅牢な暗号化アルゴリズムや鍵長を選んでも、鍵そのもののデータが簡単に取得できる状態では、第三者から情報の安全性を保つことはできない。そのため、暗号技術を利用する際の鍵の取り扱いは、もっとも重要な検討項目の一つである。一方で、守るべき資産のレベルによっては鍵の取り扱いは非常に高度な設計・実装を必要とするため本ガイドの範囲を超える。そこで本ガイドでは、鍵の配置場所と用途ごとに安全な鍵の取り扱いをするための基本的な考え方を示すにとどめ、具体的な実現方法に

関しては言及しない。必要に応じて Android セキュリティ設計・実装に詳しい専門家に相談することをお勧めする。

まずは、Android スマートフォン/タブレットにおいて暗号化等に使う鍵が存在する可能性のある場所とその保護方針の概観を「図 5.6.4 暗号鍵が存在する場所と保護方針」に示す。

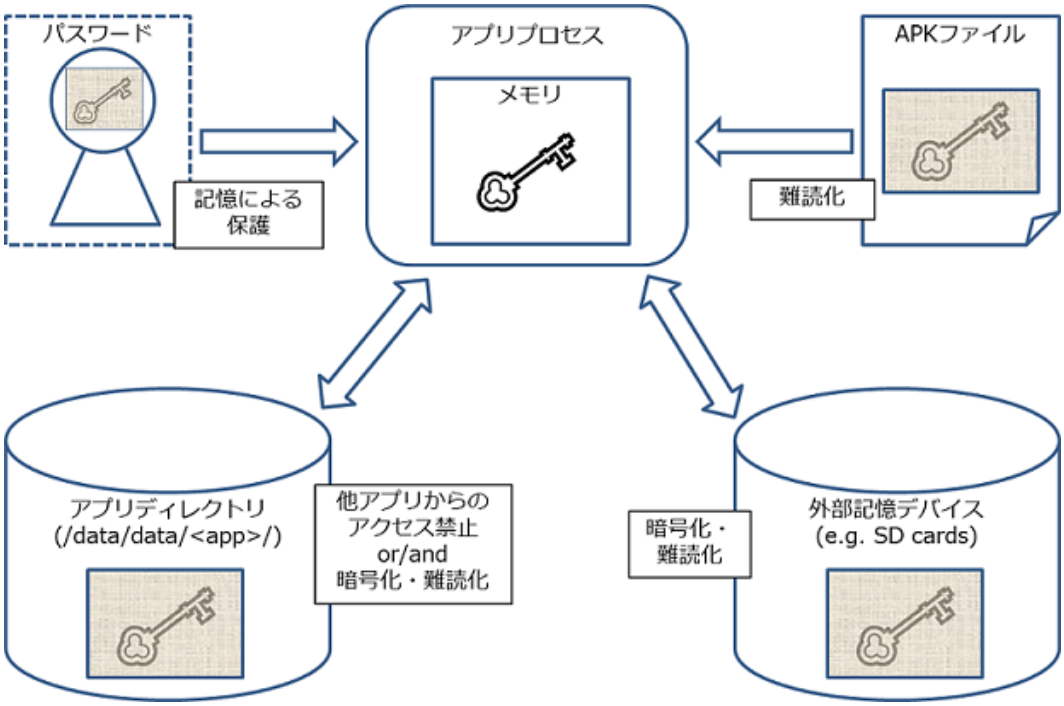

図 5.6.4　暗号鍵が存在する場所と保護方針

また、鍵によって守る資産の資産分類やオーナーによる保護方針の違いを表にまとめておく。資産レベルについては「3.1.3. 資産分類と保護施策」を参照すること。

表 5.6.7　資産分類と保護方針-1

資産のオーナー	端末のユーザー		アプリ・サービス提供者	
資産レベル	高位	中低位	高位	中低位
鍵の保存場所	保護方針			
ユーザーの記憶	パスワードの強度の向上		ユーザーパスワードの利用不可	
アプリディレクトリ (非公開ストレージ)	鍵データの暗号化・難読化	アプリ以外からの読み書き禁止	鍵データの暗号化・難読化	アプリ以外からの読み書き禁止

鍵の保存場所が APK ファイルや SD カードなどの公開ストレージの場合は次のようになる。

表 5.6.8　資産分類と保護方針-2

鍵の保存場所	保護方針
APK ファイル	鍵データの難読化 (Proguard など Java の難読化ツールの多くは、データ (文字) 列を難読化しないことに注意)
SD カードなど (公開ストレージ)	鍵データの暗号化・難読化

以下、鍵の存在場所ごとに保護方針の補足を示す。

ユーザーの記憶

この項ではパスワードベース暗号を想定している。パスワードから生成される鍵の場合は、鍵の保管場所はユーザーの記憶の中にあるため、保存場所からマルウェア等に漏洩することはない。ただし、パスワードの強度によっては簡単に鍵が再現されてしまうため、サービスへのログインパスワード等と同様にユーザーに対して一定のパスワード強度になるように UI によって制限したり、メッセージによる注意喚起したりすることが必要となる。「5.5.2.8. 利用者情報を端末内のみで利用する場合、外部送信しない旨をユーザーに通知する（推奨）」も参照すること。また、ユーザーの記憶にあるパスワードは忘れる可能性も考慮にいれておかなければならない。パスワードを忘れた場合でもデータの復元が必要な場合は、バックアップデータを端末以外の安全な場所 (サーバーなど) に保存しておく必要がある。

アプリディレクトリ内

アプリのディレクトリ内に Private モードで保存した場合は他のアプリから鍵データを読み取られることはない。また、バックアップ機能をアプリで無効にした場合には、ユーザーも参照できなくなるので、アプリの資産を守る鍵をアプリディレクトリ内に保存する場合はバックアップを無効にすれば良い。

ただし、root 権限を持つアプリやユーザーからも鍵を保護したい場合は、鍵データを暗号化もしくは難読化する必要がある。ユーザー資産を守る鍵を暗号化する場合にはパスワードベース暗号が利用できる。ユーザーにも秘密にしたいアプリの資産を暗号化する鍵を保護する場合には、APK ファイルに鍵暗号化鍵を埋める必要があり、鍵暗号化鍵データを難読化する必要がある。

APK ファイル内

APK ファイル内に存在するデータは読み取り可能であるため、基本的には鍵のような秘匿データを含めるべきではない。鍵を含める場合は鍵データを難読化して、容易に APK ファイルから鍵データが取得されないように対策を講じる必要がある。

公開ストレージ **(SD** カードなど**)** 内

公開ストレージはすべてのアプリから読み取り可能なため、基本的には鍵のような秘匿データを含めるべきではない。鍵を含める場合は鍵データを暗号化・難読化して容易に鍵データが取得されないように対策する必要がある。「アプリディレクトリ内」の root 権限を持つアプリやユーザーからも鍵を保護したい場合の対策も参照すること。

プロセスメモリ内の鍵の扱い

Android の持つ暗号技術を使用する場合、上図におけるアプリプロセス以外の場所で暗号化もしくは難読化されていた鍵データは暗号処理の手前で必ず復号 (パスワードベース鍵の場合は生成) が必要なため、プロセスメモリ内に鍵データが生のまま存在することになる。一方、アプリプロセスのメモリは通常別のアプリから読み書きすることはできないので、資産分類が本ガイドの対象範囲であれば、特別な対策を講じる必要性は少なく安全と言える。もし、アプリの扱う資産レベルや用途からプロセスメモリ内であっても鍵データがそのまま出現してはならない場合は、鍵データ・暗号ロジックの難読化など対策が必要であるが、一般に Java 層での実現は難しく、JNI 層での難読化ツールを使用することになる。これらの対応は本ガイドの範囲を超えるため、セキュア設計・実装の専門家と相談して対応すること。

5.6.3.5 Google Play 開発者サービスによる Security Provider の脆弱性対策

Google Play 開発者サービス（バージョン 5.0 以降）では、Provider Installer という仕組みが提供されており、Security Provider の脆弱性対策に利用できる。

まず、Security Provider とは、Java Cryptography Architecture（JCA）に基づいて各種暗号関連アルゴリズムの実装を提供するものである。Android アプリで暗号技術を用いる際には、Cipher、Signature、Mac といったクラスを通じてこの Security Provider を利用できる。一般に、暗号技術関連の実装で脆弱性が明らかになったときは迅速な対応が求められる。なぜなら脆弱性が悪用された際に大きな被害に繋がる可能性があるためである。Security Provider も暗号技術が関係するため、脆弱性への修正がいち早く反映されることが望ましい。

Security Provider の修正を反映する方法としては端末のアップデートが一般的である。端末のアップデートによる修正の反映は、メーカーがアップデートを用意し、ユーザーが用意されたアップデートを端末に適用するといったプロセスを経て行われる。そのため、アプリから、修正を含む最新の状態の Security Provider を使えるかどうかは、メーカーとユーザーの対応に左右される実態がある。それに対して、Google Play 開発者サービスの Provider Installer を利用すると、自動更新される Security Provider を使用できるようになる。

Google Play 開発者サービスの Provider Installer では、アプリから Provider Installer を呼び出すことで、Google Play 開発者サービスが提供する Security Provider を使用できるようになる。Google Play 開発者サービスは、Google Play ストアを通じて自動的にアップデートされるので、Provider Installer で提供される Security Provider も、メーカーとユーザーの対応状況に関わらず、最新の状態に更新されている。

以下に、Provider Installer を呼び出すサンプルコードを示す。

Provider Installer を呼び出す

```
import com.google.android.gms.common.GooglePlayServicesUtil;
import com.google.android.gms.security.ProviderInstaller;

public class MainActivity extends Activity
        implements ProviderInstaller.ProviderInstallListener {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        ProviderInstaller.installIfNeededAsync(this, this);
        setContentView(R.layout.activity_main);
    }

    @Override
    public void onProviderInstalled() {
        // Security Provider が最新、またはインストールが済んだときに呼ばれる
    }

    @Override
    public void onProviderInstallFailed(int errorCode, Intent recoveryIntent) {
        GoogleApiAvailability.getInstance().showErrorNotification(this, errorCode);

    }
}
```

5.7 指紋認証機能を利用する

生体認証の分野では、顔、声紋をはじめ、様々な方式が研究・開発されている。中でも、指紋認証は本人を特定する方法として古くから存在しており、署名（拇印）や犯罪捜査の目的で利用されてきた。コンピューターの世界においても様々な分野で応用が進み、近年ではスマートフォンの所有者識別（主に画面ロック解除）に採用されるなど、（入力の容易さなど）利便性の高い機能としての認識が浸透しつつある。

そのような流れを受けて、Android 6.0(API Level 23) では端末に指紋認証のフレームワークが組み込まれ、アプリから指紋認証（本人確認）機能が利用できるようになった。以下に、指紋認証を使った際のセキュリティ上の注意事項を記す。

5.7.1 サンプルコード

指紋認証機能には、利用者の認証情報に紐付けた鍵を使用する場合と、単純に利用者の認証のみを行う場合の大きく二つのユースケースがある。これらの指紋認証の用途に応じて、 図 5.7.1 に従いサンプルコードを選択すること。

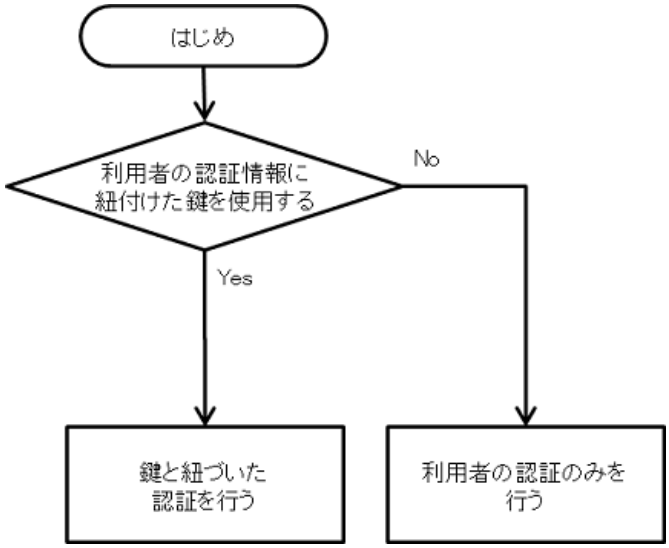

図 5.7.1　指紋認証を利用するサンプルコード選択フローチャート

Android 9.0 (API Level 28) から指紋認証機能を提供する FingerPrintManager は非推奨となり、代わって BiometricPrompt API が導入された。名称からは BiometricPrompt は指紋認証に限らず、将来的に顔や虹彩認証などの生体認証方法もサポートすることを意図したものであると考えられるが、現状のデフォルトでは指紋認証のみのサポートとなっている。また認証時には、これまでアプリが独自で用意していた UI は不要になり、標準的な認証用ダイアログが使用されるようになっている。

本記事を執筆時点 (2018 年 8 月) ではまだ BiometricPrompt の Android Support Library の提供は無い (しかし Android 9.0 の正式リリース後まもなく提供が開始されると思われる[*26]) 。そのため以下に示すサンプルコードでは最初に従来からある FingerPrintManager を利用する場合、続いて BiometricPrompt を利用する場合にわけて、それぞれ使用例を掲載する。

5.7.1.1 鍵と紐づいた認証を行う

以下に利用者の認証情報に紐付けた鍵を使う場合に指紋認証機能をアプリから利用するためのサンプルコードを示す。

[*26] https://android-developers.googleblog.com/2018/06/better-biometrics-in-android-p.html

FingerPrintManager を用いた例

ポイント:

1. USE_FINGERPRINT パーミッションを利用宣言する
2. ”AndroidKeyStore” Provider からインスタンスを取得する
3. 鍵を生成するためには指紋の登録が必要である旨をユーザーに伝える
4. 鍵生成 (登録) 時、暗号アルゴリズムは脆弱でないもの（基準を満たすもの）を使用する
5. 鍵生成 (登録) 時、ユーザー (指紋) 認証の要求を有効にする（認証の有効期限は設定しない）
6. 鍵を作る時点と鍵を使う時点で指紋の登録状況が変わることを前提に設計を行う
7. 暗号化するデータは、指紋認証以外の手段で復元（代替）可能なものに限る

```
MainActivity.java
package authentication.fingerprint.android.jssec.org.fingerprintauthentication;

import android.app.AlertDialog;
import android.hardware.fingerprint.FingerprintManager;
import android.os.Bundle;
import android.support.v7.app.AppCompatActivity;

import android.util.Base64;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;

import java.text.SimpleDateFormat;
import java.util.Date;

import javax.crypto.BadPaddingException;
import javax.crypto.Cipher;
import javax.crypto.IllegalBlockSizeException;

public class MainActivity extends AppCompatActivity {

    private FingerprintAuthentication mFingerprintAuthentication;
    private static final String SENSITIVE_DATA = "sensitive data";

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        mFingerprintAuthentication = new FingerprintAuthentication(this);

        Button button_fingerprint_auth = (Button) findViewById(R.id.button_fingerprint_auth);
        button_fingerprint_auth.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                if (!mFingerprintAuthentication.isAuthenticating()) {
                    if (authenticateByFingerprint()) {
                        showEncryptedData(null);
```

(continues on next page)

(continued from previous page)

```
                    setAuthenticationState(true);
                }
            } else {
                mFingerprintAuthentication.cancel();
            }
        }
    });
}

private boolean authenticateByFingerprint() {

    if (!mFingerprintAuthentication.isFingerprintHardwareDetected()) {
        // 指紋センサーが端末に搭載されていない
        return false;
    }

    if (!mFingerprintAuthentication.isFingerprintAuthAvailable()) {
        // ★ポイント 3★ 鍵を生成するためには指紋の登録が必要である旨をユーザーに伝える
        new AlertDialog.Builder(this)
                .setTitle(R.string.app_name)
                .setMessage("指紋情報が登録されていません。\n" +
                        "設定メニューの「セキュリティ」で指紋を登録してください。\n" +
                        "指紋を登録することにより、簡単に認証することができます。")
                .setPositiveButton("OK", null)
                .show();
        return false;
    }

    // 指紋認証の結果を受け取るコールバック
    FingerprintManager.AuthenticationCallback callback = new FingerprintManager.
↪AuthenticationCallback() {
        @Override
        public void onAuthenticationError(int errorCode, CharSequence errString) {
            showMessage(errString, R.color.colorError);
            reset();
        }

        @Override
        public void onAuthenticationHelp(int helpCode, CharSequence helpString) {
            showMessage(helpString, R.color.colorHelp);
        }

        @Override
        public void onAuthenticationSucceeded(FingerprintManager.AuthenticationResult result) {

            Cipher cipher = result.getCryptoObject().getCipher();
            try {
                // ★ポイント 7★ 暗号化するデータは、指紋認証以外の手段で復元（代替）可能なものに限る
                byte[] encrypted = cipher.doFinal(SENSITIVE_DATA.getBytes());
                showEncryptedData(encrypted);
            } catch (IllegalBlockSizeException | BadPaddingException e) {
            }

            showMessage(getString(R.string.fingerprint_auth_succeeded), R.color.colorAuthenticated);
            reset();
```

(continues on next page)

(continued from previous page)

```
            }


            @Override
            public void onAuthenticationFailed() {
                showMessage(getString(R.string.fingerprint_auth_failed), R.color.colorError);
            }
        };

        if (mFingerprintAuthentication.startAuthentication(callback)) {
            showMessage(getString(R.string.fingerprint_processing), R.color.colorNormal);
            return true;
        }

        return false;
    }


    private void setAuthenticationState(boolean authenticating) {
        Button button = (Button) findViewById(R.id.button_fingerprint_auth);
        button.setText(authenticating ? R.string.cancel : R.string.authenticate);
    }

    private void showEncryptedData(byte[] encrypted) {
        TextView textView = (TextView) findViewById(R.id.encryptedData);
        if (encrypted != null) {
            textView.setText(Base64.encodeToString(encrypted, 0));
        } else {
            textView.setText("");
        }
    }

    private String getCurrentTimeString() {
        long currentTimeMillis = System.currentTimeMillis();
        Date date = new Date(currentTimeMillis);
        SimpleDateFormat simpleDateFormat = new SimpleDateFormat("HH:mm:ss.SSS");

        return simpleDateFormat.format(date);
    }

    private void showMessage(CharSequence msg, int colorId) {
        TextView textView = (TextView) findViewById(R.id.textView);
        textView.setText(getCurrentTimeString() + " :\n" + msg);
        textView.setTextColor(getResources().getColor(colorId, null));
    }

    private void reset() {
        setAuthenticationState(false);
    }
}
```

FingerprintAuthentication.java

```
package authentication.fingerprint.android.jssec.org.fingerprintauthentication;
```

(continues on next page)

(continued from previous page)

```
import android.app.KeyguardManager;
import android.content.Context;
import android.hardware.fingerprint.FingerprintManager;
import android.os.CancellationSignal;
import android.security.keystore.KeyGenParameterSpec;
import android.security.keystore.KeyInfo;
import android.security.keystore.KeyPermanentlyInvalidatedException;
import android.security.keystore.KeyProperties;

import java.io.IOException;
import java.security.InvalidAlgorithmParameterException;
import java.security.InvalidKeyException;
import java.security.KeyStore;
import java.security.KeyStoreException;
import java.security.NoSuchAlgorithmException;
import java.security.NoSuchProviderException;
import java.security.UnrecoverableKeyException;
import java.security.cert.CertificateException;
import java.security.spec.InvalidKeySpecException;

import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.NoSuchPaddingException;
import javax.crypto.SecretKey;
import javax.crypto.SecretKeyFactory;


public class FingerprintAuthentication {
    private static final String KEY_NAME = "KeyForFingerprintAuthentication";
    private static final String PROVIDER_NAME = "AndroidKeyStore";

    private KeyguardManager mKeyguardManager;
    private FingerprintManager mFingerprintManager;

    private CancellationSignal mCancellationSignal;
    private KeyStore mKeyStore;
    private KeyGenerator mKeyGenerator;
    private Cipher mCipher;

    public FingerprintAuthentication(Context context) {
        mKeyguardManager = (KeyguardManager) context.getSystemService(Context.KEYGUARD_SERVICE);
        mFingerprintManager = (FingerprintManager) context.getSystemService(Context.FINGERPRINT_SERVICE);
        reset();
    }

    public boolean startAuthentication(final FingerprintManager.AuthenticationCallback callback) {
        if (!generateAndStoreKey())
            return false;

        if (!initializeCipherObject())
            return false;

        FingerprintManager.CryptoObject cryptoObject = new FingerprintManager.CryptoObject(mCipher);

        mCancellationSignal = new CancellationSignal();
```

(continues on next page)

(continued from previous page)

```
        // 指紋認証の結果を受け取るコールバック
        FingerprintManager.AuthenticationCallback hook = new FingerprintManager.AuthenticationCallback()
↪{
            @Override
            public void onAuthenticationError(int errorCode, CharSequence errString) {
                if (callback != null) callback.onAuthenticationError(errorCode, errString);
                reset();
            }

            @Override
            public void onAuthenticationHelp(int helpCode, CharSequence helpString) {
                if (callback != null) callback.onAuthenticationHelp(helpCode, helpString);
            }

            @Override
            public void onAuthenticationSucceeded(FingerprintManager.AuthenticationResult result) {
                if (callback != null) callback.onAuthenticationSucceeded(result);
                reset();
            }

            @Override
            public void onAuthenticationFailed() {
                if (callback != null) callback.onAuthenticationFailed();
            }
        };

        // 指紋認証を実行
        mFingerprintManager.authenticate(cryptoObject, mCancellationSignal, 0, hook, null);

        return true;
    }

    public boolean isAuthenticating() {
        return mCancellationSignal != null && !mCancellationSignal.isCanceled();
    }

    public void cancel() {
        if (mCancellationSignal != null) {
            if (!mCancellationSignal.isCanceled())
                mCancellationSignal.cancel();
        }
    }

    private void reset() {
        try {
            // ★ポイント２★ "AndroidKeyStore" Providerからインスタンスを取得する
            mKeyStore = KeyStore.getInstance(PROVIDER_NAME);
            mKeyGenerator = KeyGenerator.getInstance(KeyProperties.KEY_ALGORITHM_AES, PROVIDER_NAME);
            mCipher = Cipher.getInstance(KeyProperties.KEY_ALGORITHM_AES
                    + "/" + KeyProperties.BLOCK_MODE_CBC
                    + "/" + KeyProperties.ENCRYPTION_PADDING_PKCS7);
        } catch (KeyStoreException | NoSuchPaddingException
                | NoSuchAlgorithmException | NoSuchProviderException e) {
            throw new RuntimeException("failed to get cipher instances", e);
```

(continues on next page)

(continued from previous page)

```
        }
        mCancellationSignal = null;
    }

    public boolean isFingerprintAuthAvailable() {
        return (mKeyguardManager.isKeyguardSecure() && mFingerprintManager.hasEnrolledFingerprints()) ?␣
↪true : false;
    }

    public boolean isFingerprintHardwareDetected() {

        return mFingerprintManager.isHardwareDetected();
    }

    private boolean generateAndStoreKey() {
        try {
            mKeyStore.load(null);
            if (mKeyStore.containsAlias(KEY_NAME))
                mKeyStore.deleteEntry(KEY_NAME);
            mKeyGenerator.init(
                    // ★ポイント 4★ 鍵生成（登録）時、暗号アルゴリズムは脆弱でないもの（基準を満たすもの）を使用する
                    new KeyGenParameterSpec.Builder(KEY_NAME, KeyProperties.PURPOSE_ENCRYPT)
                            .setBlockModes(KeyProperties.BLOCK_MODE_CBC)
                            .setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_PKCS7)
                            // ★ポイント 5★ 鍵生成（登録）時、ユーザー（指紋）認証の要求を有効にする（認証の有効期
限は設定しない）
                            .setUserAuthenticationRequired(true)
                            .build());
            // 鍵を生成し、Keystore(AndroidKeyStore) に保存する
            mKeyGenerator.generateKey();
            return true;
        } catch (IllegalStateException e) {
            return false;
        } catch (NoSuchAlgorithmException | InvalidAlgorithmParameterException
                | CertificateException | KeyStoreException | IOException e) {
            throw new RuntimeException("failed to generate a key", e);
        }
    }

    private boolean initializeCipherObject() {
        try {
            mKeyStore.load(null);
            SecretKey key = (SecretKey) mKeyStore.getKey(KEY_NAME, null);
            SecretKeyFactory factory = SecretKeyFactory.getInstance(KeyProperties.KEY_ALGORITHM_AES,␣
↪PROVIDER_NAME);
            KeyInfo info = (KeyInfo) factory.getKeySpec(key, KeyInfo.class);

            mCipher.init(Cipher.ENCRYPT_MODE, key);
            return true;
        } catch (KeyPermanentlyInvalidatedException e) {
            // ★ポイント 6★ 鍵を作る時点と鍵を使う時点で指紋の登録状況が変わることを前提に設計を行う
            return false;
        } catch (KeyStoreException | CertificateException | UnrecoverableKeyException | IOException
                | NoSuchAlgorithmException | InvalidKeySpecException | NoSuchProviderException |␣
↪InvalidKeyException e) {
```

(continues on next page)

(continued from previous page)

```
            throw new RuntimeException("failed to init Cipher", e);
        }
    }

}
```

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="authentication.fingerprint.android.jssec.org.fingerprintauthentication" >

    <!-- ★ポイント 1★ 指紋認証機能を使用するためには、USE_FINGERPRINT パーミッションを利用宣言する -->
    <uses-permission android:name="android.permission.USE_FINGERPRINT" />

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme" >
        <activity
            android:name=".MainActivity"
            android:screenOrientation="portrait" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

BiometricPrompt を用いた例

ポイント:

1. USE_BIOMETRIC パーミッションを利用宣言する
2. ”AndroidKeyStore” Provider からインスタンスを取得する
3. 鍵生成 (登録) 時、暗号アルゴリズムは脆弱でないもの（基準を満たすもの）を使用する
4. 鍵生成 (登録) 時、ユーザー (指紋) 認証の要求を有効にする（認証の有効期限は設定しない）
5. 鍵を作る時点と鍵を使う時点で指紋の登録状況が変わることを前提に設計を行う
6. 暗号化するデータは、指紋認証以外の手段で復元（代替）可能なものに限る

本例は 上で示した FingerPrintManager を用いた例と本質的に同じである。一点、指紋認証機能の有効性に関しては、BiometricPrompt#autehenticate() 内でチェックされ、コールバックでエラーとして扱うことができるため、サンプル内のチェック処理を取り除いてある。

MainActivity.java

```
package org.jssec.android.biometricprompt.cipher;

import android.app.Activity;
import android.hardware.biometrics.BiometricPrompt;
import android.icu.text.SimpleDateFormat;
import android.os.Bundle;
import android.util.Base64;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;
import java.util.Date;
import javax.crypto.BadPaddingException;
import javax.crypto.Cipher;
import javax.crypto.IllegalBlockSizeException;

public class MainActivity extends Activity {
    private BiometricAuthentication mBiometricAuthentication;
    private static final String SENSITIVE_DATA = "sensitive date";

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        mBiometricAuthentication = new BiometricAuthentication(this);

        Button button_biometric_auth = findViewById(R.id.button_biometric_auth);
        button_biometric_auth.setOnClickListener(new View.OnClickListener () {
            @Override
            public void onClick(View v) {
                if (!mBiometricAuthentication.isAuthenticating()) {
                    if (authenticateByBiometric()) {
                        showEncryptedData(null);
                    }
                }
            }
        });
    }

    private boolean authenticateByBiometric () {
        // 指紋認証の結果を受け取るコールバック
        BiometricPrompt.AuthenticationCallback callback = new BiometricPrompt.AuthenticationCallback() {
            @Override
            public void onAuthenticationError(int errorCode, CharSequence errString) {
                showMessage(errString, R.color.colorError);
                reset();
            }

            @Override
            public void onAuthenticationHelp(int helpCode, CharSequence helpString) {
                showMessage(helpString, R.color.colorHelp);
            }

            @Override
            public void onAuthenticationSucceeded(BiometricPrompt.AuthenticationResult result) {
```

(continues on next page)

(continued from previous page)

```
            Cipher cipher = result.getCryptoObject().getCipher();
            try {
                // ★ポイント 6★ 暗号化するデータは、指紋認証以外の手段で復元（代替）可能なものに限る
                byte[] encrypted = cipher.doFinal(SENSITIVE_DATA.getBytes());
                showEncryptedData(encrypted);
            } catch (IllegalBlockSizeException | BadPaddingException e) {
            }

            showMessage(getString(R.string.biometric_auth_succeeded), R.color.colorAuthenticated);
            reset();
        }

        @Override
        public void onAuthenticationFailed() {
            showMessage(getString(R.string.biometric_auth_failed), R.color.colorError);
        }

    };
    if (mBiometricAuthentication.startAuthentication(callback)) {
        showMessage(getString(R.string.biometric_processing), R.color.colorNormal);
        return true;
    }
    return false;
}

private void setAuthenticationState(boolean authenticating) {
    Button button = (Button) findViewById(R.id.button_biometric_auth);
    button.setText(authenticating ? R.string.cancel : R.string.authenticate);
}

private void showEncryptedData(byte[] encrypted) {
    TextView textView = (TextView) findViewById(R.id.encryptedData);
    if (encrypted != null) {
        textView.setText(Base64.encodeToString(encrypted, 0));
    } else {
        textView.setText("");
    }
}

private String getCurrentTimeString() {
    long currentTimeMillis = System.currentTimeMillis();
    Date date = new Date(currentTimeMillis);
    SimpleDateFormat simpleDateFormat = new SimpleDateFormat("HH:mm:ss.SSS");

    return simpleDateFormat.format(date);
}

private void showMessage(CharSequence msg, int colorId) {
    TextView textView = (TextView) findViewById(R.id.textView);
    textView.setText(getCurrentTimeString() + " :\n" + msg);
    textView.setTextColor(getResources().getColor(colorId, null));
}

private void reset() {
    setAuthenticationState(false);
```

(continues on next page)

(continued from previous page)

```
    }
```

BiometricAuthentication.java

```
package org.jssec.android.biometricprompt.cipher;

import android.app.KeyguardManager;
import android.content.Context;
import android.content.DialogInterface;
import android.hardware.biometrics.BiometricPrompt;
import android.os.CancellationSignal;
import android.security.keystore.KeyGenParameterSpec;
import android.security.keystore.KeyInfo;
import android.security.keystore.KeyPermanentlyInvalidatedException;
import android.security.keystore.KeyProperties;

import java.io.IOException;
import java.security.InvalidAlgorithmParameterException;
import java.security.InvalidKeyException;
import java.security.KeyStore;
import java.security.KeyStoreException;
import java.security.NoSuchAlgorithmException;
import java.security.NoSuchProviderException;
import java.security.UnrecoverableKeyException;
import java.security.cert.CertificateException;
import java.security.spec.InvalidKeySpecException;

import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.NoSuchPaddingException;
import javax.crypto.SecretKey;
import javax.crypto.SecretKeyFactory;

public class BiometricAuthentication {
    private static final String TAG = "BioAuth";

    private static final String KEY_NAME = "KeyForFingerprintAuthentication";
    private static final String PROVIDER_NAME = "AndroidKeyStore";
    private KeyguardManager mKeyguardManager;

    private BiometricPrompt mBiometricPrompt;

    private CancellationSignal mCancellationSignal;
    private KeyStore mKeyStore;
    private KeyGenerator mKeyGenerator;
    private Cipher mCipher;
    private Context mContext;

    // ユーザーが「キャンセル」ボタンをタッチした際の処理
    private DialogInterface.OnClickListener cancelListener =
            new DialogInterface.OnClickListener () {
                @Override
                public void onClick(DialogInterface dialog, int which) {
                    android.util.Log.e(TAG, "cancel");
                    if (mCancellationSignal != null) {
```

(continues on next page)

(continued from previous page)

```
                    if (!mCancellationSignal.isCanceled())
                        mCancellationSignal.cancel();
                }
            }
        };

    public BiometricAuthentication(Context context) {
        mContext = context;
        mKeyguardManager = (KeyguardManager) context.getSystemService(Context.KEYGUARD_SERVICE);
        // 認証プロンプトではキャンセル用のボタンも表示される
        // キャンセル処理は setNegativeButton の第 3 引数に与える DialogInterface.OnClickListener で実施する
        BiometricPrompt.Builder builder = new BiometricPrompt.Builder(context);
        mBiometricPrompt = builder
                .setTitle("Please Authenticate")
                .setNegativeButton("Cancel",context.getMainExecutor() ,cancelListener)
                .build();
        reset();
    }

    public boolean startAuthentication(final BiometricPrompt.AuthenticationCallback callback) {
        if (!generateAndStoreKey())
            return false;

        if (!initializeCipherObject())
            return false;

        BiometricPrompt.CryptoObject cryptoObject = new BiometricPrompt.CryptoObject(mCipher);

        mCancellationSignal = new CancellationSignal();

        // 指紋認証の結果を受け取るコールバック
        BiometricPrompt.AuthenticationCallback hook = new BiometricPrompt.AuthenticationCallback() {
            @Override
            public void onAuthenticationError(int errorCode, CharSequence errString) {
                android.util.Log.e(TAG, "onAuthenticationError");
                if (callback != null) callback.onAuthenticationError(errorCode, errString);
                reset();
            }

            @Override
            public void onAuthenticationHelp(int helpCode, CharSequence helpString) {
                android.util.Log.e(TAG, "onAuthenticationHelp");
                if (callback != null) callback.onAuthenticationHelp(helpCode, helpString);
            }

            @Override
            public void onAuthenticationSucceeded(BiometricPrompt.AuthenticationResult result) {
                android.util.Log.e(TAG, "onAuthenticationSuccess");
                if (callback != null) callback.onAuthenticationSucceeded(result);
                reset();
            }

            @Override
            public void onAuthenticationFailed() {
                android.util.Log.e(TAG, "onAuthenticationFailed");
```

(continues on next page)

(continued from previous page)

```
                if (callback != null) callback.onAuthenticationFailed();
            }
        };

        // 指紋認証を実行
        android.util.Log.e(TAG, "Starting authentication");
        mBiometricPrompt.authenticate(cryptoObject, mCancellationSignal, mContext.getMainExecutor(),␣
↪hook);

        return true;
    }

    public boolean isAuthenticating() {
        return mCancellationSignal != null && !mCancellationSignal.isCanceled();
    }



    private void reset() {

        try {
            // ★ポイント 2★ "AndroidKeyStore" Provider からインスタンスを取得する
            mKeyStore = KeyStore.getInstance(PROVIDER_NAME);
            mKeyGenerator = KeyGenerator.getInstance(KeyProperties.KEY_ALGORITHM_AES, PROVIDER_NAME);
            mCipher = Cipher.getInstance(KeyProperties.KEY_ALGORITHM_AES
                    + "/" + KeyProperties.BLOCK_MODE_CBC
                    + "/" + KeyProperties.ENCRYPTION_PADDING_PKCS7);
        } catch (KeyStoreException | NoSuchPaddingException
                | NoSuchAlgorithmException | NoSuchProviderException e) {
            throw new RuntimeException("failed to get cipher instances", e);
        }
        mCancellationSignal = null;
    }

    private boolean generateAndStoreKey() {
        try {
            mKeyStore.load(null);
            if (mKeyStore.containsAlias(KEY_NAME))
                mKeyStore.deleteEntry(KEY_NAME);
            mKeyGenerator.init(
                    // ★ポイント 3★ 鍵生成（登録）時、暗号アルゴリズムは脆弱でないもの（基準を満たすもの）を使用する
                    new KeyGenParameterSpec.Builder(KEY_NAME, KeyProperties.PURPOSE_ENCRYPT)
                            .setBlockModes(KeyProperties.BLOCK_MODE_CBC)
                            .setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_PKCS7)
                            // ★ポイント 4★ 鍵生成（登録）時、ユーザー（指紋）認証の要求を有効にする（認証の有効期
限は設定しない）
                            .setUserAuthenticationRequired(true)
                            .build());
            // 鍵を生成し、Keystore(AndroidKeyStore) に保存する
            mKeyGenerator.generateKey();
            return true;
        } catch (IllegalStateException e) {
            return false;
        } catch (NoSuchAlgorithmException | InvalidAlgorithmParameterException
                | CertificateException | KeyStoreException | IOException e) {
            android.util.Log.e(TAG, "key generation failed: " + e.getMessage());
```

(continues on next page)

(continued from previous page)

```
            throw new RuntimeException("failed to generate a key", e);
        }
    }

    private boolean initializeCipherObject() {
        try {
            mKeyStore.load(null);
            SecretKey key = (SecretKey) mKeyStore.getKey(KEY_NAME, null);
            SecretKeyFactory factory = SecretKeyFactory.getInstance(KeyProperties.KEY_ALGORITHM_AES,␣
↪PROVIDER_NAME);
            KeyInfo info = (KeyInfo) factory.getKeySpec(key, KeyInfo.class);

            mCipher.init(Cipher.ENCRYPT_MODE, key);
            return true;
        } catch (KeyPermanentlyInvalidatedException e) {
            // ★ポイント 5★ 鍵を作る時点と鍵を使う時点で指紋の登録状況が変わることを前提に設計を行う
            return false;
        } catch (KeyStoreException | CertificateException | UnrecoverableKeyException | IOException
                | NoSuchAlgorithmException | InvalidKeySpecException | NoSuchProviderException |␣
↪InvalidKeyException e) {
            android.util.Log.e(TAG, "failed to init Cipher: " + e.getMessage());
            throw new RuntimeException("failed to init Cipher", e);
        }
    }
}
```

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="org.jssec.android.biometricprompt.cipher">

    <!-- ★ポイント 1★ USE_BIOMETRIC パーミッションを利用宣言する -->
    <uses-permission android:name="android.permission.USE_BIOMETRIC" />
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name="org.jssec.android.biometricprompt.cipher.MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

5.7.1.2 利用者の認証のみ行う

以下に、利用者による認証のみ行う場合に指紋認証機能を利用するためのサンプルコードを示す。この場合、特にセキュリティ上注意すべき点はないが、参考のためサンプルコードを以下に掲載する。

FingerPrintManager を用いた例

ポイント

1. USE_FINGERPRINT パーミッションを利用宣言する

MainActivity.java

```
package org.jssec.android.fingerprint.authentication.nocipher;

import android.hardware.fingerprint.FingerprintManager;
import android.os.Bundle;
import android.support.v7.app.AlertDialog;
import android.support.v7.app.AppCompatActivity;
import android.util.Base64;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;

import java.text.SimpleDateFormat;
import java.util.Date;


public class MainActivity extends AppCompatActivity {

    private FingerprintAuthentication mFingerprintAuthentication;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        mFingerprintAuthentication = new FingerprintAuthentication(this);

        Button button_fingerprint_auth = (Button) findViewById(R.id.button_fingerprint_auth);
        button_fingerprint_auth.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                if (!mFingerprintAuthentication.isAuthenticating()) {
                    if (authenticateByFingerprint()) {
                        showEncryptedData(null);
                        setAuthenticationState(true);
                    }
                } else {
                    mFingerprintAuthentication.cancel();
                }
            }
        });
    }

    private boolean authenticateByFingerprint() {

        if (!mFingerprintAuthentication.isFingerprintHardwareDetected()) {
            // 指紋センサーが端末に搭載されていない
            return false;
        }
```

(continues on next page)

(continued from previous page)

```
        if (!mFingerprintAuthentication.isFingerprintAuthAvailable()) {
            // 指紋の登録が必要である旨をユーザーに伝える
            new AlertDialog.Builder(this)
                    .setTitle(R.string.app_name)
                    .setMessage("指紋情報が登録されていません。\n" +
                            "設定メニューの「セキュリティ」で指紋を登録してください。\n" +
                            "指紋を登録することにより、簡単に認証することができます。")
                    .setPositiveButton("OK", null)
                    .show();
            return false;
        }

        // 指紋認証の結果を受け取るコールバック
        FingerprintManager.AuthenticationCallback callback = new FingerprintManager.
↪AuthenticationCallback() {
            @Override
            public void onAuthenticationError(int errorCode, CharSequence errString) {
                showMessage(errString, R.color.colorError);

                reset();
            }

            @Override
            public void onAuthenticationHelp(int helpCode, CharSequence helpString) {
                showMessage(helpString, R.color.colorHelp);
            }

            @Override
            public void onAuthenticationSucceeded(FingerprintManager.AuthenticationResult result) {
                showMessage(getString(R.string.fingerprint_auth_succeeded), R.color.colorAuthenticated);
                reset();
            }

            @Override
            public void onAuthenticationFailed() {
                showMessage(getString(R.string.fingerprint_auth_failed), R.color.colorError);
            }
        };

        if (mFingerprintAuthentication.startAuthentication(callback)) {
            showMessage(getString(R.string.fingerprint_processing), R.color.colorNormal);
            return true;
        }

        return false;
    }


    private void setAuthenticationState(boolean authenticating) {
        Button button = (Button) findViewById(R.id.button_fingerprint_auth);
        button.setText(authenticating ? R.string.cancel : R.string.authenticate);
    }

    private void showEncryptedData(byte[] encrypted) {
        TextView textView = (TextView) findViewById(R.id.encryptedData);
```

(continues on next page)

(continued from previous page)

```
        if (encrypted != null) {
            textView.setText(Base64.encodeToString(encrypted, 0));
        } else {
            textView.setText("");
        }
    }

    private String getCurrentTimeString() {
        long currentTimeMillis = System.currentTimeMillis();
        Date date = new Date(currentTimeMillis);
        SimpleDateFormat simpleDateFormat = new SimpleDateFormat("HH:mm:ss.SSS");

        return simpleDateFormat.format(date);
    }

    private void showMessage(CharSequence msg, int colorId) {
        TextView textView = (TextView) findViewById(R.id.textView);
        textView.setText(getCurrentTimeString() + " :\n" + msg);
        textView.setTextColor(getResources().getColor(colorId, null));
    }

    private void reset() {
        setAuthenticationState(false);
    }
}
```

FingerprintAuthentication.java

```
package org.jssec.android.fingerprint.authentication.nocipher;

import android.content.Context;
import android.hardware.fingerprint.FingerprintManager;
import android.os.CancellationSignal;

public class FingerprintAuthentication {

    private FingerprintManager mFingerprintManager;
    private CancellationSignal mCancellationSignal;


    public FingerprintAuthentication(Context context) {
        mFingerprintManager = (FingerprintManager) context.getSystemService(Context.FINGERPRINT_SERVICE);
        reset();
    }

    public boolean startAuthentication(final FingerprintManager.AuthenticationCallback callback) {

        mCancellationSignal = new CancellationSignal();

        // 指紋認証の結果を受け取るコールバック
        FingerprintManager.AuthenticationCallback hook = new FingerprintManager.AuthenticationCallback()
↪{
            @Override
            public void onAuthenticationError(int errorCode, CharSequence errString) {
                if (callback != null) callback.onAuthenticationError(errorCode, errString);
```

(continues on next page)

(continued from previous page)

```
                reset();
            }

            @Override
            public void onAuthenticationHelp(int helpCode, CharSequence helpString) {
                if (callback != null) callback.onAuthenticationHelp(helpCode, helpString);
            }

            @Override
            public void onAuthenticationSucceeded(FingerprintManager.AuthenticationResult result) {
                if (callback != null) callback.onAuthenticationSucceeded(result);
                reset();
            }

            @Override
            public void onAuthenticationFailed() {
                if (callback != null) callback.onAuthenticationFailed();
            }
        };

        // 指紋認証を実行
        // 第一引数の CryptoObject に null を与えることにより暗号鍵とは紐づかない認証となる
        mFingerprintManager.authenticate(null, mCancellationSignal, 0, hook, null);

        return true;
    }

    public boolean isAuthenticating() {
        return mCancellationSignal != null && !mCancellationSignal.isCanceled();
    }

    public void cancel() {
        if (mCancellationSignal != null) {
            if (!mCancellationSignal.isCanceled())
                mCancellationSignal.cancel();
        }
    }

    private void reset() {
        mCancellationSignal = null;
    }

    public boolean isFingerprintAuthAvailable() {
        return (mFingerprintManager.hasEnrolledFingerprints()) ? true : false;
    }

    public boolean isFingerprintHardwareDetected() {
        return mFingerprintManager.isHardwareDetected();
    }
}
```

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
```

(continues on next page)

(continued from previous page)

```
    package="authentication.fingerprint.android.jssec.org.fingerprintauthentication" >

    <!-- ★ポイント 1★ 指紋認証機能を使用するためには、USE_FINGERPRINT パーミッションを利用宣言する -->
    <uses-permission android:name="android.permission.USE_FINGERPRINT" />

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme" >
        <activity
            android:name=".MainActivity"
            android:screenOrientation="portrait" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

BiometricPrompt を用いた例

ポイント:

1. USE_BIOMETRIC_PROMPT パーミッションを利用宣言する

MainActivity.java

```
package org.jssec.android.biometricprompt.nocipher;

import android.hardware.biometrics.BiometricPrompt;
import android.icu.text.SimpleDateFormat;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;
import org.jssec.android.biometric.authentication.nocipher.R;
import java.util.Date;

public class MainActivity extends AppCompatActivity {
    private BiometricAuthentication mBiometricAuthentication;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        mBiometricAuthentication = new BiometricAuthentication(this);

        Button button_biometric_auth = findViewById(R.id.button_biometric_auth);
```

(continues on next page)

(continued from previous page)

```
        button_biometric_auth.setOnClickListener(new View.OnClickListener () {
            @Override
            public void onClick(View v) {
                if (!mBiometricAuthentication.isAuthenticating()) {
                    authenticateByBiometric();
                }
            }
        });
    }

    private boolean authenticateByBiometric () {

        BiometricPrompt.AuthenticationCallback callback = new BiometricPrompt.AuthenticationCallback() {
            @Override
            public void onAuthenticationError(int errorCode, CharSequence errString) {
                showMessage(errString, R.color.colorError);
            }

            @Override
            public void onAuthenticationHelp(int helpCode, CharSequence helpString) {
                showMessage(helpString, R.color.colorHelp);
            }

            @Override
            public void onAuthenticationSucceeded(BiometricPrompt.AuthenticationResult result) {
                showMessage(getString(R.string.biometric_auth_succeeded), R.color.colorAuthenticated);
            }

            @Override
            public void onAuthenticationFailed() {
                showMessage(getString(R.string.biometric_auth_failed), R.color.colorError);
            }

        };
        if (mBiometricAuthentication.startAuthentication(callback)) {
            showMessage(getString(R.string.biometric_processing), R.color.colorNormal);
            return true;
        }
        return false;
    }

    private String getCurrentTimeString() {
        long currentTimeMillis = System.currentTimeMillis();
        Date date = new Date(currentTimeMillis);
        SimpleDateFormat simpleDateFormat = new SimpleDateFormat("HH:mm:ss.SSS");

        return simpleDateFormat.format(date);
    }

    private void showMessage(CharSequence msg, int colorId) {
        TextView textView = (TextView) findViewById(R.id.textView);
        textView.setText(getCurrentTimeString() + " :\n" + msg);
        textView.setTextColor(getResources().getColor(colorId, null));
    }
}
```

BiometricAuthentication.java

```
package org.jssec.android.biometricprompt.nocipher;

import android.content.Context;
import android.content.DialogInterface;
import android.hardware.biometrics.BiometricPrompt;
import android.os.CancellationSignal;

public class BiometricAuthentication {
    private static final String TAG = "BioAuth";

    private BiometricPrompt mBiometricPrompt;
    private CancellationSignal mCancellationSignal;
    private Context mContext;

    // ユーザーが「キャンセル」ボタンをタッチした際の処理
    private DialogInterface.OnClickListener cancelListener =
            new DialogInterface.OnClickListener () {
                @Override
                public void onClick(DialogInterface dialog, int which) {
                    android.util.Log.d(TAG, "cancel");
                    if (mCancellationSignal != null) {
                        if (!mCancellationSignal.isCanceled())
                            mCancellationSignal.cancel();
                    }
                }
            };

    public BiometricAuthentication(Context context) {
        mContext = context;
        BiometricPrompt.Builder builder = new BiometricPrompt.Builder(context);
        // 認証プロンプトではキャンセル用のボタンも表示される
        // キャンセル処理は setNegativeButton の第 3 引数に与える DialogInterface.OnClickListener で実施する
        mBiometricPrompt = builder
                .setTitle("Please Authenticate")
                .setNegativeButton("Cancel",context.getMainExecutor() ,cancelListener)
                .build();
        reset();
    }

    public boolean startAuthentication(final BiometricPrompt.AuthenticationCallback callback) {

        mCancellationSignal = new CancellationSignal();

        // 指紋認証の結果を受け取るコールバック
        BiometricPrompt.AuthenticationCallback hook = new BiometricPrompt.AuthenticationCallback() {
            @Override
            public void onAuthenticationError(int errorCode, CharSequence errString) {
                android.util.Log.d(TAG, "onAuthenticationError");
                if (callback != null) callback.onAuthenticationError(errorCode, errString);
                reset();
            }

            @Override
            public void onAuthenticationHelp(int helpCode, CharSequence helpString) {
                android.util.Log.d(TAG, "onAuthenticationHelp");
```

(continues on next page)

(continued from previous page)

```
                if (callback != null) callback.onAuthenticationHelp(helpCode, helpString);
            }

            @Override
            public void onAuthenticationSucceeded(BiometricPrompt.AuthenticationResult result) {
                android.util.Log.d(TAG, "onAuthenticationSuccess");
                if (callback != null) callback.onAuthenticationSucceeded(result);
                reset();
            }

            @Override
            public void onAuthenticationFailed() {
                android.util.Log.d(TAG, "onAuthenticationFailed");
                if (callback != null) callback.onAuthenticationFailed();
            }
        };

        // 指紋認証を実行
        // BiometricPrompt の場合、単なる認証用の API が別にある
        android.util.Log.d(TAG, "Starting authentication");
        mBiometricPrompt.authenticate(mCancellationSignal, mContext.getMainExecutor(), hook);

        return true;
    }

    public boolean isAuthenticating() {
        return mCancellationSignal != null && !mCancellationSignal.isCanceled();
    }

    private void reset() {
        mCancellationSignal = null;
    }

}
```

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="org.jssec.android.biometricprompt.cipher">

    <!-- ★ポイント 1★ USE_BIOMETRIC パーミッションを利用宣言する -->
    <uses-permission android:name="android.permission.USE_BIOMETRIC" />
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name="org.jssec.android.biometricprompt.cipher.MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
```

(continues on next page)

(continued from previous page)

```
        </activity>
    </application>
</manifest>
```

5.7.2 ルールブック

暗号鍵と紐づいた認証を行う際に指紋認証機能を利用する場合には以下のルールを守ること。それ以外の用途で指紋認証機能を利用する場合は、特にルールはない。

1. 鍵生成 (登録) 時、暗号アルゴリズムは脆弱でないもの（基準を満たすもの）を使用する （必須）
2. 暗号化するデータは、指紋認証以外の手段で復元（代替）可能なものに限る （必須）
3. 鍵を生成するためには指紋の登録が必要であり、ユーザーにその旨を伝える （推奨）

5.7.2.1 鍵生成 (登録) 時、暗号アルゴリズムは脆弱でないもの（基準を満たすもの）を使用する （必須）

「5.6. 暗号技術を利用する」で解説したパスワード鍵や公開鍵等と同様に、指紋認証機能を用いて鍵を生成する場合においても、第三者による盗聴を防ぐため一定の基準を満たした脆弱でない暗号アルゴリズムを設定すること。また、暗号アルゴリズムだけではなく、暗号モードやパディングについても脆弱でない安全な方式を選択すること。

暗号アルゴリズムの選択については「5.6.2.2. 脆弱でないアルゴリズム（基準を満たすもの）を使用する （必須）」を参照すること。

5.7.2.2 暗号化するデータは、指紋認証以外の手段で復元（代替）可能なものに限る （必須）

アプリ内データの暗号化に際して指紋認証機能を利用する場合は、そのデータが指紋認証機能以外の手段でも復元（代替）できるようにアプリを設計すること。一般に、生体情報には秘匿や変更の困難性、誤認識といった問題が付随しているため、認証を生体情報のみに頼ることは避けるべきである。

例えば、指紋認証機能を利用して生成した鍵を用いてアプリ内のデータを暗号化した後、端末に登録されていた指紋情報がユーザーによって削除されると、データの暗号化に用いていた鍵が使用できなくなり、データを復号することもできなくなる。データの復元が指紋認証機能以外の手段でも行えなければ、可用性が損なわれる可能性が生じるのである。

また、指紋認証機能を利用して生成した鍵が使用できなくなる状況は指紋情報の削除以外でも生じうる。Nexus5X においては、指紋認証機能を利用して鍵を生成した後に新たに指紋情報を追加登録すると、それ以前に生成した鍵が使用できなくなることが確認されている[*27]。また、本来正しく使用できるべき鍵が指紋センサーの誤認識により使用できなくなる可能性も否定できない。

5.7.2.3 鍵を生成するためには指紋の登録が必要であり、ユーザーにその旨を伝える （推奨）

指紋認証機能を利用して鍵を生成するためには端末にユーザーの指紋が登録されている必要がある。指紋登録ユーザーに促すため、設定メニューへ誘導する際には、指紋が重要な個人情報であることに留意し、ユーザーに対してアプリが指紋を利用する必要性や利便性について説明することが望ましい。

指紋認証の登録が必要である旨、ユーザーに伝える

[*27] 2016 年 9 月 1 日版執筆時点の情報。後日、修正される可能性がある。

```
        if (!mFingerprintAuthentication.isFingerprintAuthAvailable()) {
            // ★ポイント★ 鍵を生成するためには指紋の登録が必要である旨をユーザーに伝える
            new AlertDialog.Builder(this)
                    .setTitle(R.string.app_name)
                    .setMessage("指紋情報が登録されていません。\n" +
                            "設定メニューの「セキュリティ」で指紋を登録してください。\n" +
                            "指紋を登録することにより、簡単に認証することができます。")
                    .setPositiveButton("OK", null)
                    .show();
            return false;
        }
```

5.7.3 アドバンスト

5.7.3.1 Android アプリにおける指紋認証機能利用の前提条件

アプリで指紋認証機能を利用するためには以下の 2 点が必要である。

- 端末にユーザーの指紋が登録されていること
- 登録された指紋に（アプリ固有の）鍵が紐づいていること

ユーザーの指紋登録

ユーザーの指紋情報は設定メニューの「セキュリティ」からしか登録することができず、一般のアプリが指紋登録処理を行うことはできない。そのため、アプリが指紋認証機能を利用する時点で端末に指紋が登録されていなければ、ユーザーを設定メニューに誘導し指紋の登録を促す必要がある。その際、アプリが指紋を利用する必要性や利便性についての説明がユーザーに対して行われることが望ましい。

なお、指紋登録が可能となる前提として、端末に予備の画面ロック方式が設定されていることが必要である。端末に指紋が登録されている状態で画面ロックを無効にすると、登録済みの指紋情報も削除される。

鍵の生成・登録

端末に登録された指紋と鍵を紐付けるためには、”AndroidKeyStore” Provider が提供する KeyGenerator や KeyStore インスタンス等を使用する際に、ユーザー（指紋）認証の要求を有効にして、新しい鍵の生成と登録、あるいは既存の鍵の登録を行う。

指紋情報と紐づいた鍵を生成する場合は、KeyGenerator を生成する際のパラメータとして、ユーザー認証の要求を有効にするよう設定する。

指紋情報と紐づいた鍵の生成と登録

```
    try {
        // "AndroidKeyStore" Provider から KeyGenerator インスタンスを取得する
        KeyGenerator keyGenerator = KeyGenerator.getInstance(KeyProperties.KEY_ALGORITHM_AES,
↪"AndroidKeyStore");
        keyGenerator.init(
            new KeyGenParameterSpec.Builder(KEY_NAME, KeyProperties.PURPOSE_ENCRYPT)
            .setBlockModes(KeyProperties.BLOCK_MODE_CBC)
            .setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_PKCS7)
```

(continues on next page)

(continued from previous page)

```
            .setUserAuthenticationRequired(true)  // ユーザー（指紋）認証の要求を有効にする
            .build());
        keyGenerator.generateKey();
    } catch (IllegalStateException e) {
        // 端末に指紋が登録されていない
      throw new RuntimeException("No fingerprint registered", e);
    } catch (NoSuchAlgorithmException | InvalidAlgorithmParameterException
                | CertificateException | KeyStoreException | IOException e) {
        // 鍵の生成に失敗
        throw new RuntimeException("Failed to generate a key", e);
    }
```

既存の鍵に指紋情報を紐づける場合は、ユーザー認証の要求を有効にする設定を加えた KeyStore エントリーに鍵を登録する。

既存の鍵に指紋情報を紐付ける

```
    SecretKey key = existingKey;  // 既存の鍵

    KeyStore keyStore = KeyStore.getInstance("AndroidKeyStore");
    keyStore.load(null);
    keyStore.setEntry(
        "alias_for_the_key",
        new KeyStore.SecretKeyEntry(key),
        new KeyProtection.Builder(KeyProperties.PURPOSE_ENCRYPT)
                .setUserAuthenticationRequired(true)  // ユーザー（指紋）認証の要求を有効にする
                .build());
```

6 難しい問題

Android には OS の仕様や OS が提供する機能の仕様上、アプリの実装でセキュリティを担保するのが困難な問題が存在する。これらの機能は悪意を持った第三者に悪用されたり、ユーザーが注意せずに利用したりすることで、情報漏洩を始めセキュリティ上の問題に繋がってしまう危険性を常に抱えている。この章ではそのような機能に対して、開発者が取りうるリスク削減策などを提示しながら注意喚起が必要な話題を記事として取り上げる。

6.1 Clipboard から情報漏洩する危険性

コピー＆ペーストはユーザーが普段から何気なく使っている機能であろう。例えば、この機能を使って、メールや Web ページで気になった情報や忘れたら困る情報をメモ帳に残しておいたり、設定したパスワードを忘れないようにメモ帳に保存しておき、必要な時にコピー＆ペーストして使うというユーザーは少なからず存在する。これらは一見何気ない行為であるが、実はユーザーの扱う情報が盗まれるという危険が潜んでいる。

これには Android のコピー＆ペーストの仕組みが関係している。ユーザーやアプリによってコピーされた情報は、一旦 Clipboard と呼ばれるバッファに格納される。ユーザーやアプリによってペーストされたときに、この Clipboard の内容が各アプリに再配布されるわけである。この Clipboard に情報漏洩に結び付く危険性がある。Android 端末の仕様では、Clipboard の実体は端末に 1 つであり、ClipboardManager を利用することで、どのアプリからでも常時 Clipboard の中身が取得できるようになっているからである。このことは、ユーザーがコピー・カットした情報は全て悪意あるアプリに対して筒抜けになることを意味している。

よって、アプリ開発者は、この Android の仕様を考慮しながら情報漏洩の可能性を最小限に抑える対策を講じなくてはならない。

6.1.1 サンプルコード

Clipboard から情報漏洩する可能性を抑える対策には、大きく分けて次の 2 つが考えられる。

(1) 他アプリから自アプリへコピーする際の対策

(2) 自アプリから他アプリへコピーする際の対策

最初に、1. について説明する。ここでは、ユーザーがメモ帳や Web ブラウザ、メーラーアプリなど他アプリから文字列をコピーし、それを自アプリの EditText に貼り付けるシナリオを想定している。結論だけを言ってしまうと、このシナリオでコピー・カットによってセンシティブな情報が漏洩してしまうことを防ぐ根本的な対策は存在しない。第三者アプリのコピー機能を制御するような機能が Android にはないからだ。

よって、1. についてはセンシティブな情報をコピー・カットする危険性をユーザーに説明し、行為自体を減らしていく啓発活動を継続的に行っていくしか対策はない。

次に、2. を説明する。ここでは、自アプリが表示している情報がユーザーによってコピーされるシナリオを想定する。この場合、漏洩に対する確実な対策は、View(TextView, EditText など) からのコピー・カットを禁止にすることである。個人情報などセンシティブな情報が入力あるいは出力される View にコピー・カット機能がなければ、自アプリからの Clipboard を介した情報の漏洩もないからだ。

コピー・カットを禁止する方法はいくつか考えられるが、ここでは、実装が簡単でかつ効果のある方法として、View の長押し無効化の方法と文字列選択時のメニューからコピー・カットの項目を削除する方法を扱う。

対策要否は、図 6.1.1 の判定フローによって判定することができる。図 6.1.1 において、入力タイプ (Input Type) が Password 属性に固定されているとは、入力タイプ (Input Type) がアプリの実行時に常に下記のいずれかであることを指す。この場合は、デフォルトでコピー・カットが禁止されているので、特に対策する必要はない。

- InputType.TYPE_CLASS_TEXT | InputType.TYPE_TEXT_VARIATION_PASSWORD
- InputType.TYPE_CLASS_TEXT | InputType.TYPE_TEXT_VARIATION_WEB_PASSWORD
- InputType.TYPE_CLASS_NUMBER | InputType.TYPE_NUMBER_VARIATION_PASSWORD

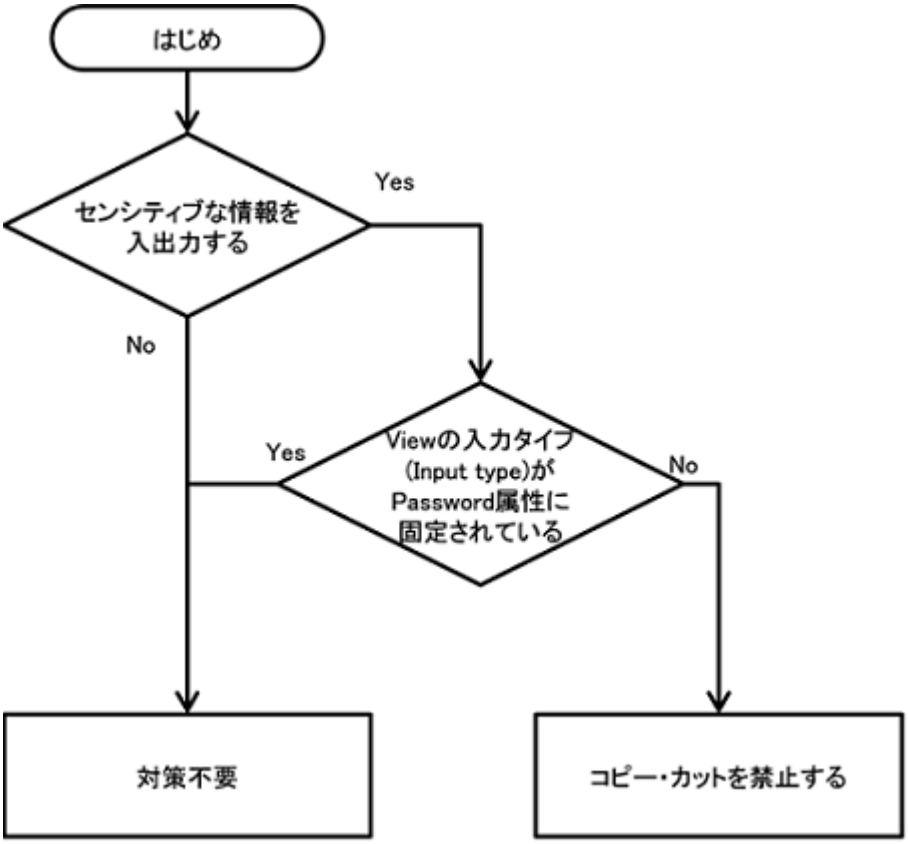

図 6.1.1　対策要否の判定フロー

以下で、それぞれの対策の詳細を説明し、サンプルコードを示す。

6.1.1.1 文字列選択時のメニューからコピー・カットを削除する

TextView.setCustomSelectionActionModeCallback() メソッドによって、文字列選択時のメニューをカスタマイズできる。これを用いて、文字列選択時のメニューからコピー・カットのアイテムを削除すれば、ユーザーが文字列をコピー・カッ

トすることはできなくなる。

以下、EditText の文字列選択時のメニューからコピー・カットの項目を削除するサンプルコードを示す。

ポイント：

1. 文字列選択時のメニューから android.R.id.copy を削除する。
2. 文字列選択時のメニューから android.R.id.cut を削除する。

UncopyableActivity.java

```
package org.jssec.android.clipboard.leakage;

import android.app.Activity;
import android.os.Bundle;
import android.support.v4.app.NavUtils;
import android.view.ActionMode;
import android.view.Menu;
import android.view.MenuItem;
import android.widget.EditText;

public class UncopyableActivity extends Activity {
    private EditText copyableEdit;
    private EditText uncopyableEdit;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.uncopyable);

        copyableEdit = (EditText) findViewById(R.id.copyable_edit);
        uncopyableEdit = (EditText) findViewById(R.id.uncopyable_edit);
        // setCustomSelectionActionModeCallback メソッドにより、
        // 文字列選択時のメニューをカスタマイズすることができる。
        uncopyableEdit.setCustomSelectionActionModeCallback(actionModeCallback);
    }

    private ActionMode.Callback actionModeCallback = new ActionMode.Callback() {
        public boolean onPrepareActionMode(ActionMode mode, Menu menu) {
            return false;
        }

        public void onDestroyActionMode(ActionMode mode) {
        }

        public boolean onCreateActionMode(ActionMode mode, Menu menu) {
            // ★ポイント 1★ 文字列選択時のメニューから android.R.id.copy を削除する。
            MenuItem itemCopy = menu.findItem(android.R.id.copy);
            if (itemCopy != null) {
                menu.removeItem(android.R.id.copy);
            }
            // ★ポイント 2★ 文字列選択時のメニューから android.R.id.cut 削除する。
            MenuItem itemCut = menu.findItem(android.R.id.cut);
            if (itemCut != null) {
                menu.removeItem(android.R.id.cut);
            }
            return true;
```

(continues on next page)

(continued from previous page)

```
        }

        public boolean onActionItemClicked(ActionMode mode, MenuItem item) {
            return false;
        }
    };

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.uncopyable, menu);
        return true;
    }


    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        switch (item.getItemId()) {
            case android.R.id.home:
                NavUtils.navigateUpFromSameTask(this);
                return true;
        }
        return super.onOptionsItemSelected(item);
    }

}
```

6.1.1.2 View の長押し (Long Click) を無効にする

コピー・カットを禁止する方法は、View の長押し (Long Click) を無効にすることでも実現できる。View の長押し無効化はレイアウトの xml ファイルで指定することができる。

ポイント：

1. コピー・カットを禁止する View は android:longClickable を false にする。

```
unlongclickable.xml
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/unlongclickable_description" />

    <!-- コピー・カットを禁止する EditText -->
    <!-- ★ポイント 1★ コピー・カットを禁止する View は android:longClickable を false にする。 -->
    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:longClickable="false"
        android:hint="@string/unlongclickable_hint" />
```

(continues on next page)

(continued from previous page)

```
</LinearLayout>
```

6.1.2 ルールブック

自アプリから他アプリへのセンシティブな情報のコピーが発生する可能性がある場合は、以下のルールを守ること。

1. *View* に表示されている文字列のコピー・カットを無効にする *(必須)*

6.1.2.1 View に表示されている文字列のコピー・カットを無効にする (必須)

アプリがセンシティブな情報を表示する View を持っている場合、それが EditText のようにコピー・カットが可能な View ならば、Clipboard を介してその情報が漏洩してしまう可能性がある。そのため、センシティブな情報を表示する View はコピー・カットを無効にしておかなければならない。

コピー・カットを無効にする方法には、文字列選択時のメニューからコピー・カットの項目を削除する方法と、View の長押しを無効化する方法がある。

「6.1.3.1. ルール適用の際の注意」も参照のこと。

6.1.3 アドバンスト

6.1.3.1 ルール適用の際の注意

TextView はデフォルトでは文字列選択不可であるため、通常は対策不要であるが、アプリの仕様によってはコピーを可能にする場合もある。TextView.setTextIsSelectable() メソッドを使うことで、文字列の選択可否とコピー可否を動的に設定することができる。TextView をコピー可能とする場合は、その TextView にセンシティブな情報が表示される可能性がないかよく検討し、その可能性があるのであれば、コピー可にすべきでない。

また、「6.1.1. サンプルコード」の判定フローにも記載されているように、パスワードの入力を想定した入力タイプ (InputType.TYPE_CLASS_TEXT|InputType.TYPE_TEXT_VARIATION_PASSWORD など) の EditText については、デフォルトで文字列のコピーが禁止されているため通常は対策不要である。しかし、「5.1.2.2. パスワードを平文表示するオプションを用意する （必須)」に記載したように「パスワードを平文表示する」オプションを用意している場合は、パスワード平文表示の際に入力タイプが変化し、コピー・カットが有効になってしまうので、同様の対策が必要である。

なお、ルールを適用する際には、ユーザビリティの面も考慮する必要があるだろう。例えば、ユーザーが自由にテキストを入力できる View の場合、センシティブな情報が入力される「可能性がゼロでない」からといってコピー・カットを無効にしてしまったら、ユーザーの使い勝手が悪くなるだろう。もちろん、重要度の高い情報を入出力する View やセンシティブな情報を単独で入力するような View にはルールを無条件で適用するべきであるが、それ以外の View を扱う場合は、次のことを考慮しながら対応を考えると良い。

- センシティブな情報の入力や表示を行う専用のコンポーネントを用意できないか
- 連携先 (ペースト先) アプリが分かっている場合は、他の方法で情報を送信できないか
- アプリでユーザーに入出力に関する注意喚起ができないか
- 本当にその View が必要か

Android OS の Clipboard と ClipboardManager の仕様にセキュリティに対する考慮がされていないことが情報漏洩の可能性を生む根本的な要因ではあるが、アプリ開発者は、ユーザー保護やユーザビリティ、提供する機能など様々な観点からこうした Clipboard の仕様に対して対応し、質の高いアプリを作成する必要がある。

6.1.3.2 Clipboard に格納されている情報の操作

「6.1. *Clipboard* から情報漏洩する危険性」で述べたように、ClipboardManager を利用することでアプリから Clipboard に格納された情報を操作することができる。また、ClipboardManager の利用には特別な Permission を設定する必要が無いため、アプリはユーザーに知られることなく ClipboardManager を利用できる。

Clipboard に格納されている情報 (ClipData と呼ぶ) は、ClipboardManager.getPrimaryClip() メソッドによって取得できる。タイミングに関しても、OnPrimaryClipChangedListener を実装して ClipboardManager.addPrimaryClipChangedListener() メソッドで ClipboardManager に登録すれば、ユーザーの操作などにより発生するコピー・カットの度に Listener が呼び出されるので、タイミングを逃すことなく ClipData を取得することができる。ここで Listener の呼び出しは、どのアプリでコピー・カットが発生したかに関係なく行われる。

以下、端末内でコピー・カットが発生する度に ClipData を取得し、Toast で表示する Service のソースコードを示す。下記のような簡単なコードにより Clipboard に格納された情報が筒抜けになってしまうことを実感していただきたい。アプリを実装する際は、少なくとも下記のコードによってセンシティブな情報が取得されてしまうことのないように注意する必要がある。

ClipboardListeningService.java

```
package org.jssec.android.clipboard;

import android.app.Service;
import android.content.ClipData;
import android.content.ClipboardManager;
import android.content.ClipboardManager.OnPrimaryClipChangedListener;
import android.content.Context;
import android.content.Intent;
import android.os.IBinder;
import android.util.Log;
import android.widget.Toast;

public class ClipboardListeningService extends Service {
    private static final String TAG = "ClipboardListeningService";
    private ClipboardManager mClipboardManager;

    @Override
    public IBinder onBind(Intent arg0) {
        return null;
    }

    @Override
    public void onCreate() {
        super.onCreate();
        mClipboardManager = (ClipboardManager) getSystemService(Context.CLIPBOARD_SERVICE);
        if (mClipboardManager != null) {
            mClipboardManager.addPrimaryClipChangedListener(clipListener);
        } else {
            Log.e(TAG, "ClipboardService の取得に失敗しました。サービスを終了します。");
            this.stopSelf();
        }
```

(continues on next page)

(continued from previous page)

```
    }

    @Override
    public void onDestroy() {
        super.onDestroy();
        if (mClipboardManager != null) {
            mClipboardManager.removePrimaryClipChangedListener(clipListener);
        }
    }

    private OnPrimaryClipChangedListener clipListener = new OnPrimaryClipChangedListener() {
        public void onPrimaryClipChanged() {
            if (mClipboardManager != null && mClipboardManager.hasPrimaryClip()) {
                ClipData data = mClipboardManager.getPrimaryClip();
                ClipData.Item item = data.getItemAt(0);
                Toast
                    .makeText(
                        getApplicationContext(),
                        "コピーあるいはカットされた文字列:\n"
                            + item.coerceToText(getApplicationContext()),
                        Toast.LENGTH_SHORT)
                    .show();
            }
        }
    };
}
```

次に、上記 ClipboardListeningService を利用する Activity のソースコードの例を示す。

ClipboardListeningActivity.java

```
package org.jssec.android.clipboard;

import android.app.Activity;
import android.content.ComponentName;
import android.content.Intent;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.widget.Toast;

public class ClipboardListeningActivity extends Activity {
    private static final String TAG = "ClipboardListeningActivity";

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_clipboard_listening);
    }

    public void onClickStartService(View view) {
        if (view.getId() != R.id.start_service_button) {
            Log.w(TAG, "View ID が不正です");
        } else {
            ComponentName cn = startService(
                    new Intent(ClipboardListeningActivity.this, ClipboardListeningService.class));
```

(continues on next page)

(continued from previous page)

```
            if (cn == null) {
                Log.e(TAG, "サービスの起動に失敗しました");
                Toast.makeText(this, "サービスの起動に失敗しました", Toast.LENGTH_SHORT).show();
            }
        }
    }

    public void onClickStopService(View view) {
        if (view.getId() != R.id.stop_service_button) {
            Log.w(TAG, "View ID が不正です");
        } else {
            stopService(new Intent(ClipboardListeningActivity.this, ClipboardListeningService.class));
        }
    }
}
```

ここまでは、Clipboard に格納された情報を取得する方法について述べたが、ClipboardManager.setPrimaryClip() メソッドによって、Clipboard に新しく情報を格納することも可能である。

ただし、setPrimaryClip() は Clipboard に格納されていた情報を上書きするので、ユーザーが予めコピー・カット操作により格納しておいた情報が失われる可能性がある点に注意が必要である。これらのメソッドを使用して独自のコピー機能あるいはカット機能を提供する場合は、必要に応じて、内容が改変される旨を警告するダイアログを表示するなど、Clipboard に格納されている内容がユーザーの意図しない内容に変更されることのないように設計・実装する必要がある。

更新履歴

2012-06-01　初版

2012-11-01

下記の構成・内容を見直し拡充致しました

- 4.1. *Activity* を作る・利用する
- 4.2. *Broadcast* を受信する・送信する
- 4.3. *Content Provider* を作る・利用する
- 4.4. *Service* を作る・利用する
- 5.2. *Permission* と *Protection Level*

下記の新しい記事を追加致しました

- 2.5. サンプルコードの *Android Studio* への取り込み手順
- 3.1. *Android* アプリのセキュリティ
- 4.7. *Browsable Intent* を利用する
- 5.3. *Account Manager* に独自アカウントを追加する
- 6.1. *Clipboard* から情報漏洩する危険性

2013-04-01

下記の記事の内容を見直し書き直しました

- 5.3. *Account Manager* に独自アカウントを追加する

下記の新しい記事を追加致しました

- 4.8. *LogCat* にログ出力する
- 5.4. *HTTPS* で通信する
- 4.9. *WebView* を使う

2014-07-01

下記の新しい記事を追加致しました

- 5.5. プライバシー情報を扱う
- 5.6. 暗号技術を利用する

2015-06-01

下記の方針で本書全体の内容を見直し書き直しました

- 開発環境の変更 (Eclipse -> Android Studio)
- Android 最新版 Lollipop への対応
- 対応する API Level の見直し (8 以降 -> 15 以降)

2016-02-01

下記の新しい記事を追加致しました

- 4.10. *Notification* を使用する
- 5.7. 指紋認証機能を利用する

下記の構成・内容を見直し拡充致しました

- 5.2. *Permission* と *Protection Level*

2016-09-01

下記の構成・内容を見直し拡充致しました

- 2.5. サンプルコードの *Android Studio* への取り込み手順
- 5.4. *HTTPS* で通信する
- 5.6. 暗号技術を利用する

2017-02-01

下記の新しい記事を追加致しました

- 4.6.3.5. *Android 7.0*（*API Level 24*）における外部ストレージの特定ディレクトリへのアクセスに関する仕様変更について
- 5.4.3.7. *Network Security Configuration*

下記の構成・内容を見直し拡充致しました

- 4.1. *Activity* を作る・利用する
- 4.2. *Broadcast* を受信する・送信する
- 4.4. *Service* を作る・利用する
- 4.5. *SQLite* を使う
- 4.6. ファイルを扱う

下記の記事を削除致しました

- 4.8.3.4 BuildConfig.DEBUG は ADT 21 以降で使う

下記の方針で本書全体の内容を見直し書き直しました

- Android 4.0.3（API Level 15）未満に関する本文中の記述を削除または脚注へ移動

2018-02-01

下記の新しい記事を追加致しました

- 4.1.3.7. *Autofill* フレームワークについて
- 5.3.3.3. *Android 8.0*（*API Level 26*）以降で署名の一致しない *Authenticator* のアカウントを読めるケース
- 5.4.3.8. （コラム） セキュア接続の *TLS1.2* への移行について
- 5.5.3.3. *Android ID* のバージョンによる違いについて

下記の構成・内容を見直し拡充致しました

- 4.2. *Broadcast* を受信する・送信する
- 5.2. *Permission* と *Protection Level*
- 5.3. *Account Manager* に独自アカウントを追加する
- 5.4. *HTTPS* で通信する
- 5.5. プライバシー情報を扱う

2018-09-01

下記の新しい記事を追加いたしました

- 4.9.3.4. *WebView* の *Safe Browsing* について
- 4.11. 共有メモリを使用する

下記の構成・内容を見直し拡充いたしました

- 2.5. サンプルコードの *Android Studio* への取り込み手順
- 4.1.3.7. *Autofill* フレームワークについて
- 4.5.3.6. *[参考]SQLite* データベースを暗号化する *(SQLCipher for Android)*
- 5.2.1.2. 独自定義の *Signature Permission* で自社アプリ連携する方法
- 5.4.1.2. *HTTPS* 通信する
- 5.4.3.7. *Network Security Configuration*
- 5.7. 指紋認証機能を利用する
- 5.4.3.2. *Android OS* の証明書ストアにプライベート認証局のルート証明書をインストールする
- 5.4.3.8. （コラム） セキュア接続の *TLS1.2* への移行について

※ 改訂内容の詳細については「1.4. *2018* 年 *2* 月 *1* 日版からの訂正記事について」を参照して下さい。

新版の公開にあたり、皆様から頂いたご意見・コメントを元に本ガイドの内容を更新しました。

制作

一般社団法人 日本スマートフォンセキュリティ協会 技術部会 セキュアコーディング WG

リーダー	安藤 彰	ソニーデジタルネットワークアプリケーションズ株式会社
メンバー	澤田 壽實	株式会社 SRA
	鈴木 康平	株式会社 SRA
	本間 輝彰	KDDI 株式会社
	小木曽 純	ソニーデジタルネットワークアプリケーションズ株式会社
	久本 純樹	ソニーデジタルネットワークアプリケーションズ株式会社
	山口 信明	ソニーデジタルネットワークアプリケーションズ株式会社
	谷田部 茂	株式会社フォーマルハウト・テクノ・ソリューションズ

（執筆関係者、社名五十音順）

2018 年 2 月 1 日版製作者

リーダー

安藤彰 ソニーデジタルネットワークアプリケーションズ株式会社

メンバー

奥山 謙	Android セキュリティ部
星本 英史	株式会社 SRA
武井 滋紀	エヌ・ティ・ティ・ソフトウェア株式会社
塩田 明弘	株式会社エヌ・ティ・ティ・データ
福本 郁哉	一般社団法人 JPCERT コーディネーションセンター (JPCERT/CC)
小木曽 純、久本 純樹、山口 信明、吉田 万里子	ソニーデジタルネットワークアプリケーションズ株式会社
笠原 正弘	ソフトバンクモバイル株式会社
伊藤 健文	日本システム開発株式会社
谷田部 茂	タオソフトウェア株式会社

(執筆関係者、社名五十音順)

2017 年 2 月 1 日版制作者

リーダー

奥山謙 ソニーデジタルネットワークアプリケーションズ株式会社

メンバー

荒木 成治、島野 英司	Android セキュリティ部
大内 智美、福本 郁哉、山野井 陽一	株式会社 SRA
武井 滋紀	エヌ・ティ・ティ・ソフトウェア株式会社
塩田 明弘	株式会社エヌ・ティ・ティ・データ
高橋 哲也	株式会社スクウェア・エニックス
山地 秀典	ソニー株式会社
安藤 彰、小木曽 純、松並 勝	ソニーデジタルネットワークアプリケーションズ株式会社
谷口 岳	タオソフトウェア株式会社

(執筆関係者、社名五十音順)

2016 年 9 月 1 日版制作者

リーダー

松並勝 ソニーデジタルネットワークアプリケーションズ株式会社

メンバー

荒木 成治	Android セキュリティ部
大内 智美、福本 郁哉	株式会社 SRA
武井 滋紀	エヌ・ティ・ティ・ソフトウェア株式会社
大園 通	シスコシステムズ合同会社
山地 秀典	ソニー株式会社
安藤 彰、大谷 三岳、小木曽 純、奥山 謙	ソニーデジタルネットワークアプリケーションズ株式会社
島野 英司、谷口 岳	タオソフトウェア株式会社
満園 大祐	日本システム開発株式会社

(執筆関係者、社名五十音順)

2016 年 2 月 1 日版制作者

リーダー

松並勝 ソニーデジタルネットワークアプリケーションズ株式会社

メンバー

安達 正臣	Android セキュリティ部
福本 郁哉、星本 英史	株式会社 SRA
武井 滋紀	エヌ・ティ・ティ・ソフトウェア株式会社
大園 通	シスコシステムズ合同会社
安藤 彰、伊藤 妙子、大谷 三岳、奥山 謙、楫節子、西村 宗晃	ソニーデジタルネットワークアプリケーションズ株式会社
山地 秀典	ソニーモバイルコミュニケーションズ株式会社
笠原 正弘	ソフトバンクモバイル株式会社
島野 英司、谷口 岳	タオソフトウェア株式会社

(執筆関係者、社名五十音順)

2015 年 6 月 1 日版制作者

リーダー

松並勝 ソニーデジタルネットワークアプリケーションズ株式会社

メンバー

星本 英史	株式会社 SRA
武井 滋紀	エヌ・ティ・ティ・ソフトウェア株式会社
大園 通	シスコシステムズ合同会社
安藤 彰、奥山 謙、西村 宗晃	ソニーデジタルネットワークアプリケーションズ株式会社
笠原 正弘	ソフトバンクモバイル株式会社
島野 英司、谷口 岳	タオソフトウェア株式会社
八津川 直伸	日本ユニシス株式会社
谷田部 茂	株式会社フォーマルハウト・テクノ・ソリューションズ
今西 杏丞、河原 豊、近藤 昭雄、志村 直彦、新谷 正人、原 昇平、藤澤 智之、藤田 竜史、三竹 一馬	株式会社ブリリアントサービス

(執筆関係者、社名五十音順)

2014 年 7 月 1 日版制作者

リーダー

松並勝 ソニーデジタルネットワークアプリケーションズ株式会社

メンバー

熊澤 努、星本 英史	株式会社 SRA
武井 滋紀	エヌ・ティ・ティ・ソフトウェア株式会社
竹森 敬祐、磯原 隆将	KDDI 株式会社
大園 通	シスコシステムズ合同会社
安藤 彰、伊藤 妙子、奥山 謙、椙 節子、片岡 良典	ソニーデジタルネットワークアプリケーションズ株式会社
笠原 正弘	ソフトバンクモバイル株式会社
島野 英司、谷口 岳	タオソフトウェア株式会社
佐藤 導吉	東京システムハウス株式会社
八津川 直伸	日本ユニシス株式会社
谷田部 茂	株式会社フォーマルハウト・テクノ・ソリューションズ

(執筆関係者、社名五十音順)

2013 年 4 月 1 日版制作者

リーダー

松並勝 ソニーデジタルネットワークアプリケーションズ株式会社

メンバー

安達 正臣、長谷川 智之	Android セキュリティ部
安部 勇気、大内 智美、熊澤 努、澤田 寿実、畑 清志、比嘉 陽一、福井 悠、福本 郁哉、星本 英史、横井 俊、吉澤 孝和	株式会社 SRA
藤原 健	NRI セキュアテクノロジーズ株式会社
武井 滋紀	エヌ・ティ・ティ・ソフトウェア株式会社
竹森 敬祐	KDDI 株式会社
久保 正樹、熊谷 裕志、戸田 洋三	一般社団法人 JPCERT コーディネーションセンター (JPCERT/CC)
大園 通	シスコシステムズ合同会社
新井 幹也、坂本 昌彦	株式会社セキュアスカイ・テクノロジー
浅野 徹、安藤 彰、池邉 亮志、小木曽 純、奥山 謙、片岡 良典、西村 宗晃、古澤 浩司、山岡 研二	ソニーデジタルネットワークアプリケーションズ株式会社
谷口 岳	タオソフトウェア株式会社
八津川 直伸	日本ユニシス株式会社
谷田部 茂	株式会社フォーマルハウト・テクノ・ソリューションズ

(執筆関係者、社名五十音順)

2012 年 11 月 1 日版制作者

リーダー

松並勝 ソニーデジタルネットワークアプリケーションズ株式会社

メンバー

佐藤 勝彦、中口 明彦	Android セキュリティ部
大内 智美、大平 直之、熊澤 努、関川 未来、中野 正剛、比嘉 陽一、福本 郁哉、星本 英史、安田 章一、八尋 唯行、吉澤 孝和	株式会社 SRA
武井 滋紀	エヌ・ティ・ティ・ソフトウェア株式会社
竹森 敬祐	KDDI 株式会社
久保 正樹、熊谷 裕志、戸田 洋三	一般社団法人 JPCERT コーディネーションセンター (JPCERT/CC)
大園 通	シスコシステムズ合同会社
浅野 徹、安藤 彰、池邉 亮志、市川 茂、大谷 三岳、小木曽 純、奥山 謙、片岡 良典、佐藤 郁恵、西村 宗晃、山岡 一夫、吉川 岳流	ソニーデジタルネットワークアプリケーションズ株式会社
谷口 岳、島野 英司、北村 久雄	タオソフトウェア株式会社
山川 隆郎	一般社団法人日本オンラインゲーム協会
石原 正樹、森 靖晃	日本システム開発株式会社
八津川 直伸	日本ユニシス株式会社
谷田部 茂	株式会社フォーマルハウト・テクノ・ソリューションズ
藤井 茂樹	ユニアデックス株式会社

(執筆関係者、社名五十音順)

2012 年 6 月 1 日版制作者

リーダー

松並勝 ソニーデジタルネットワークアプリケーションズ株式会社

メンバー

佐藤 勝彦	Android セキュリティ部
大内 智美、比嘉 陽一、星本 英史	株式会社 SRA
武井 滋紀	エヌ・ティ・ティ・ソフトウェア株式会社
千田 雅明	グリー株式会社
久保 正樹、熊谷 裕志、戸田 洋三	一般社団法人 JPCERT コーディネーションセンター (JPCERT/CC)
大園 通、谷田部 茂	シスコシステムズ合同会社
田口 陽一	株式会社システムハウス．アイエヌジー
坂本 昌彦	株式会社セキュアスカイ・テクノロジー
安藤 彰、市川 茂、奥山 謙、佐藤 郁恵、西村 宗晃、山岡 一夫	ソニーデジタルネットワークアプリケーションズ株式会社
谷口 岳、島野 英司、北村 久雄	タオソフトウェア株式会社
佐藤 導吉	東京システムハウス株式会社
服部 正和	トレンドマイクロ株式会社
八津川 直伸	日本ユニシス株式会社
谷田部 茂	株式会社フォーマルハウト・テクノ・ソリューションズ
藤井 茂樹	ユニアデックス株式会社

(執筆関係者、社名五十音順)