TLS 1.3

山本和彦

おしながき

- TLS とその歴史
- TLS 1.2 の問題点
  - プロトコルの問題
  - 暗号技術の老朽化
- TLS 1.3
  - 機密性の向上
  - 速度の向上
  - 安全性の向上
- TLS 1.3 の実装

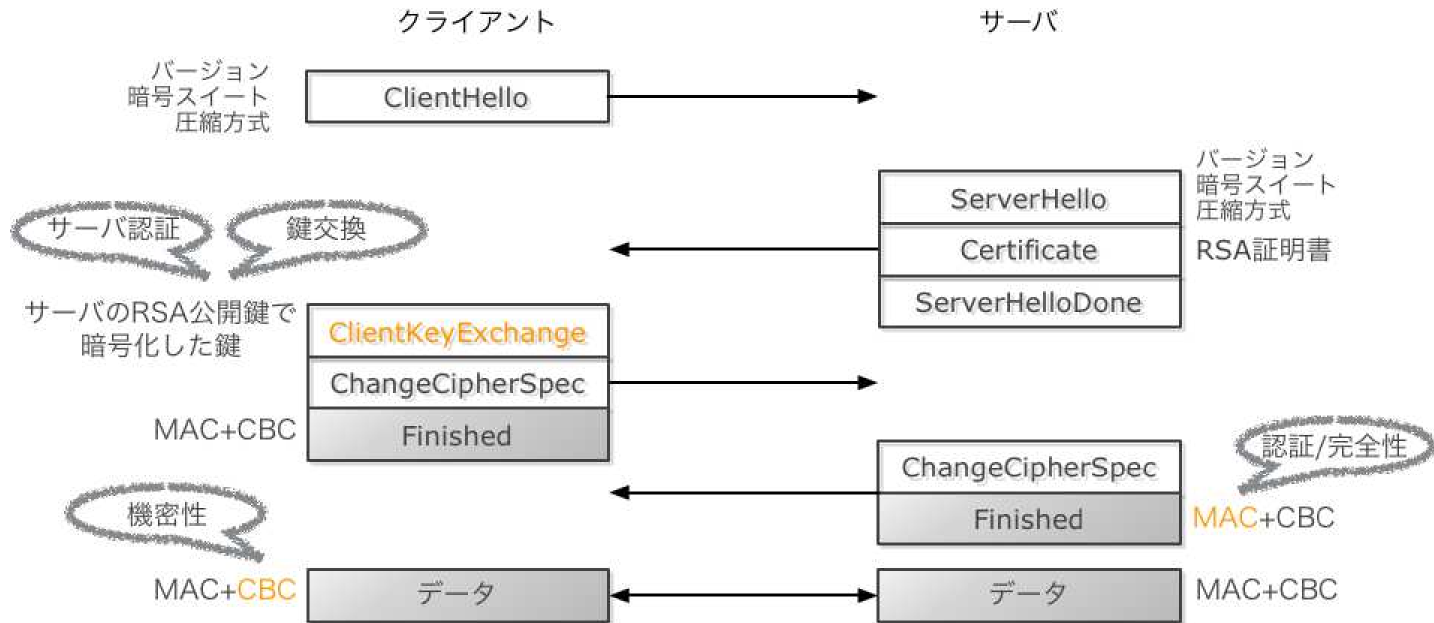
TLSとその歴史

TLS の役割

機密性 confidentiality	意図した相手だけが通信の内容を知り得ること 盗聴されないこと
認証 authentication	ある名前を名乗る人が本当にその人だと確かめること サーバ認証は必須、オプションでクライアント認証
完全性 integrity	通信の内容がそのまま相手に伝わること 改竄を検知できること

TLS 1.2 のフルハンドシェイク

TLS1.2のデフォルトの暗号スイート
TLS_RSA_WITH_AES_128_CBC_SHA



TLS の歴史

	策定年	攻撃	状況
SSL 2.0	1995	DROWN	RFC 6176により禁止
SSL 3.0	1996	POODLE	RFC 7568により禁止
TLS 1.0	1999	BEAST	認証付き暗号がない
TLS 1.1	2006		認証付き暗号がない
TLS 1.2	2008		認証付き暗号がある
TLS 1.3	策定中		プロトコルの大幅な変更

TLS 1.1以前のことは忘れましょう

TLS 1.2の問題点

TLS 1.2への脅威

広域監視

圧縮への攻撃

再ネゴシエーション
への攻撃

暗号技術の老朽化

広域監視と前方秘匿性

- 広域監視の存在が明らかに
  - Pervasive monitoring
  - トラフィックはすべて保存されていると考えるべき
  - 静的な秘密鍵が流出すると過去の暗号文を解読される
- 前方秘匿性の重要性が高まる
  - Forward secrecy
  - 鍵交換は前方秘匿性を持つべき
- サーバ認証と鍵交換が密結合
  - 静的なRSA証明書に対する静的なRSA秘密鍵
  - 静的な鍵は前方秘匿性を持たない
- 前方秘匿性を持つ Diffie Hellman の復権
  - 公開鍵と秘密鍵を動的に作って使い捨てる
  - DHE (Diffie Hellman Ephemeral)
  - ECDHE (Elliptic Curve Diffie Hellman Ephemeral)

圧縮

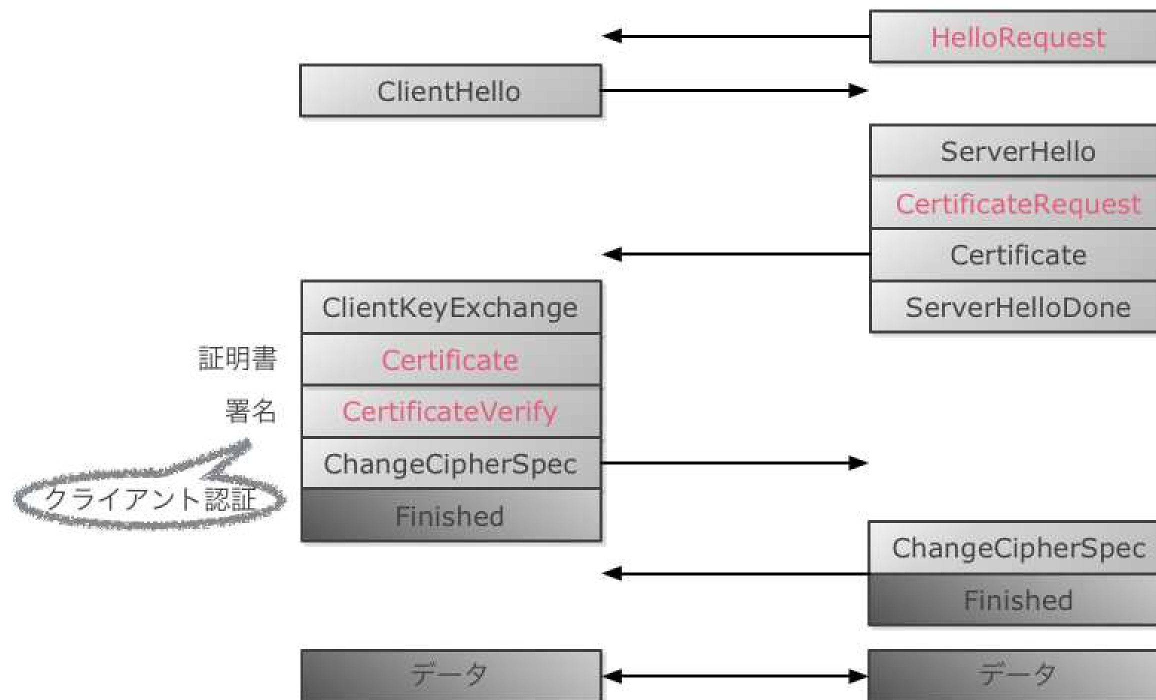
- トランスポートでの圧縮は危険
- CRIME攻撃
  - 暗号を解読しないでクッキーを盗む
- 準備
  - 悪意のサイトへアクセスさせ
    悪意のスクリプトをダウンロードさせる
  - TLSで暗号化されたパケットを見張る
- 攻撃
  - 悪意のスクリプトは、HTTPヘッダを変えながら
    盗みたいクッキーを発行したサーバへアクセスする
  - パケットサイズの変化を見て、クッキーと一致した文字列を推測
- 圧縮は使ってはいけない

再ネゴシエーション

- 2つの目的で利用されている
  - 鍵の更新
  - クライアント認証
- 再ネゴシエーションは危険
  - 中間者攻撃(MitM)を受ける (RFC5746)
  - サーバがDoSを受ける
    - RSA秘密鍵(d は巨大)の利用は RSA公開鍵($e = 2^{2^4}+1 = 65537$)の利用より重い
    - TCP/IPレベルの頻度フィルタでは防げない
- 不自由で安全でない設計
  - クライアント認証はネゴシエーションでしかできない
  - クライアント証明書は平文で流れることもある
- クライアントから再ネゴシエーションさせてはいけない

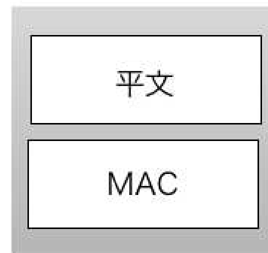
再ネゴシエーション・ハンドシェイク

■ 暗号路の確立後、クライアント認証が必要となった例



暗号モード

- ストリーム暗号
  - 実質 RC4 のみ
  - RC 4 には多くの問題が見つかっていて非推奨 (RFC 7465)
- ブロック暗号のCBCモード
  - CBC モードはもう終わったという雰囲気
  - SSL 3.0 の CBC モードに対する POODLE (パディングオラクル攻撃の一種)



暗号モードとMAC

- MAC 後暗号化
  - TLS 1.2以前のストリーム暗号、CBCモード共にこの方法
  - 安全でないとされる
    - 暗号メールなどでは、MACが存在するか隠せるので機密性が高いと言われた時代もあった
  - TLS 1.0 の CBC モードに対する BEAST
- 暗号化後 MAC
  - 安全だとされる
  - RFC 7366 で定義されているが、ほとんど使われていない

SHA-1 の老朽化

- SHA-1
  - すでに安全でないので利用しない方がよい
  - Chrome や Firefox では SHA-1 を使った証明書を 2017年を目処に受け入れなくなる
- SHA-2
  - SHA224
    - SHA256 の劣化版なので、SHA256を使えという雰囲気
  - SHA256
  - SHA384
  - SHA512
- SHA-3
  - 定義されたが SHA-2 があればいいという雰囲気

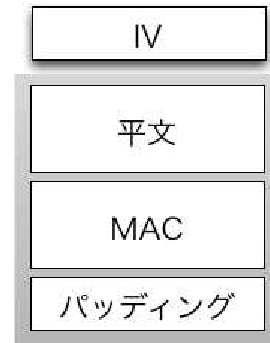
三重苦

RC4は脆弱

CBCモードは脆弱

ストリーム暗号

ブロック暗号のCBCモード



MAC後暗号化は脆弱

SHA1は脆弱

SHA256だと長すぎ？

認証付き暗号

- TLS 1.2 で採用された第三の暗号化手法
  - AEAD: Authenticated Encryption with Associated Data
  - 暗号化と同時に認証
  - ハッシュ関数は使わない
- 2つの認証付き暗号
  - GCM (Galois/Counter Mode)
  - CCM (Counter and CBC MAC Mode)

TLS 1.2 と HTTP/2

- HTTP/2 は TLS 1.2 を安全な暗号技術と共に使う
  - 前方秘匿性を持つECDHE
  - 圧縮禁止
  - 通信途中の再ネゴシエーション禁止
  - 認証付き暗号(AEAD)の利用
  - 128ビットの安全性

	古いデフォルト		
鍵交換	RSA	DHE	ECDHE HTTP/2 楕円曲線はP256
サーバ認証	RSA HTTP/2		
共通鍵暗号	CBC	Stream	AEAD HTTP/2
セキュアハッシュ関数	SHA1	SHA256 HTTP/2	SHA512

鍵長

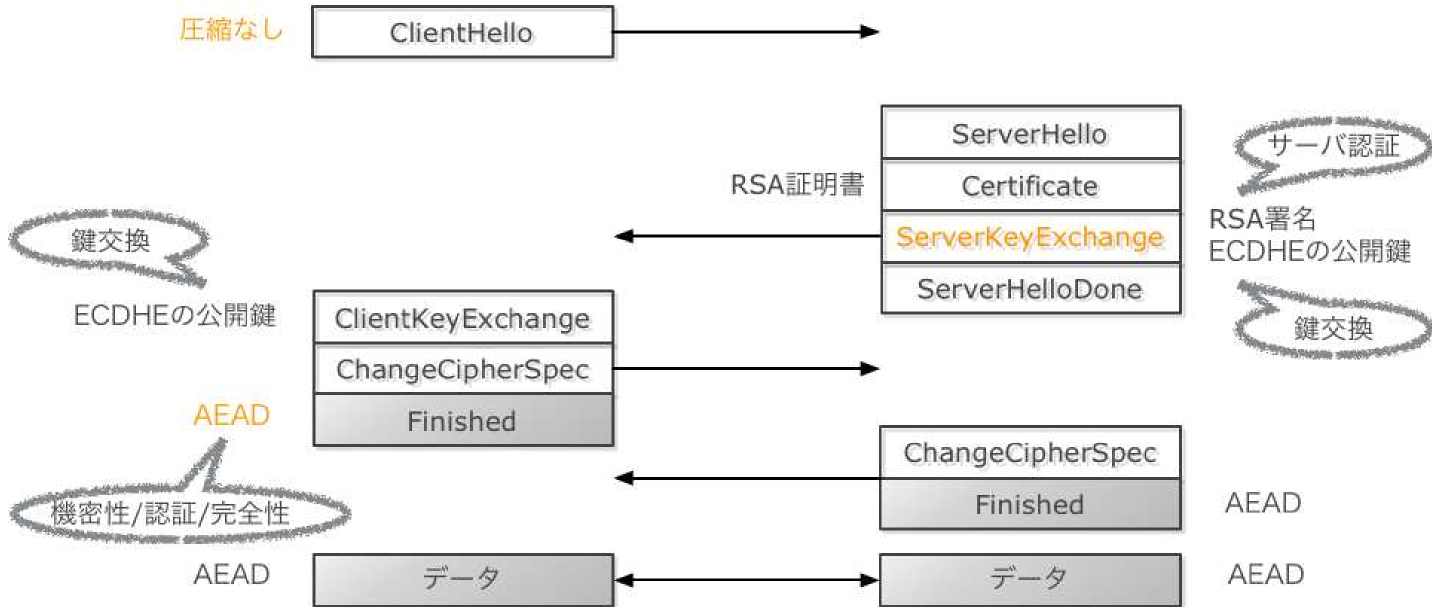
オススメ 2050年ごろまで安全

安全性	80ビット	112ビット	128ビット	192ビット	256ビット
共通鍵暗号	80	112	128	192	256
RSA/DH	1024	2048	3072	7680	15360
楕円曲線暗号	160	224	256	384	512
ハッシュ関数	160	224	256	384	512

「デファクトスタンダード暗号技術の大移行」より

TLS 1.2 のフルハンドシェイク(2)

TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256



TLS 1.2 のまとめ

- TLS 1.2 の問題点
  - プロトコル自体の欠陥
    - 圧縮、再ネゴシエーション
  - 暗号技術の老朽化
    - RC4、CBC、SHA-1
  - 継ぎ接ぎだらけの仕様
    - 拡張の RFC が多過ぎる
- 参考文献
  - RFC 7457: TLS 1.2 以前への攻撃手法のまとめ
  - RFC 7525: TLS 1.2 のオススメの利用方法
- TLS 1.2を安全に使えているかの検査
  - SSL Server Test で "A" を取りましょう
  - https://www.ssllabs.com/ssltest/

TLS 1.3

TLS 1.3 の特徴

安全性の向上	老朽化した暗号技術の排除 再ネゴシエーションの排除 圧縮の排除 前方秘匿性 (サーバ認証と鍵交換の分離)
機密性の向上	クライアント証明書の暗号化 サーバからの拡張の暗号化
速度の向上	1RTT 再開 (前方秘匿性あり) 0RTT 再開 (前方秘匿性なし)

TLS のハンドシェイク

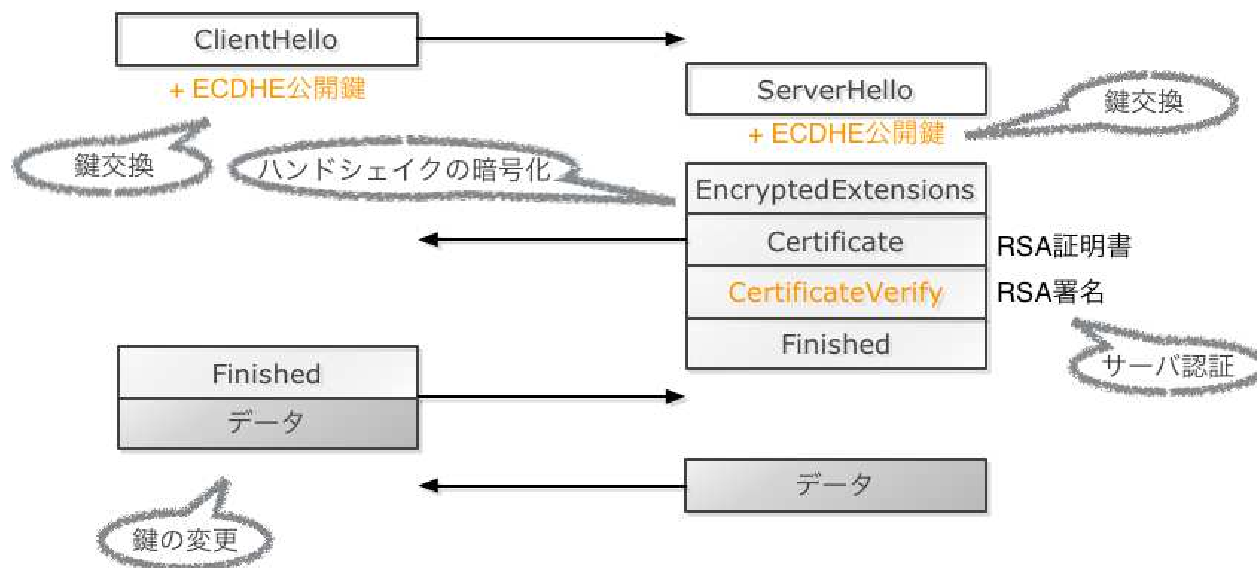
TLS 1.2	TLS 1.3
フルハンドシェイク	フルハンドシェイク
	再トライ
再開	再開 + PSK
	0RTT
再ネゴシエーション	

暗号スイート

- TLS 1.2 の最後の解釈は難しい
  - MAC と擬似乱数の意味
  - 擬似乱数は SHA256 以上 (SHAと書いてSHA256と読む)
  - AEAD だと MAC の意味はない
- TLS 1.3 では鍵交換とサーバ認証が削除された

フルハンドシェイク

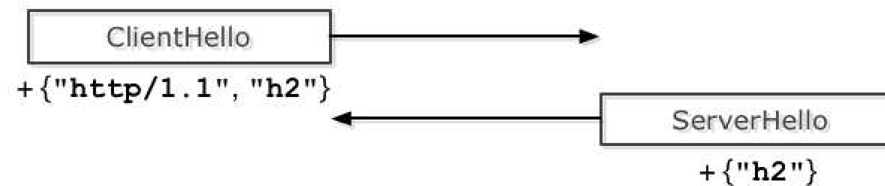
- 前方秘匿性のある 1RTT
  - (EC)DHEを使ったクライアントからの鍵交換
  - サーバ認証との分離
  - TLS 1.2 のフルハンドシェイクでも false start を使えば前方秘匿性のある 1RTT が実現できる



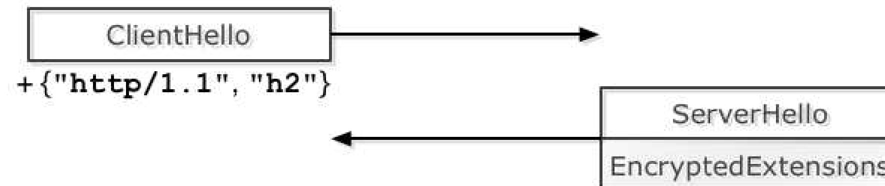
サーバからの拡張の暗号化

- ALPNの例
  - Application Layer Protocol Negotiation

TLS 1.2

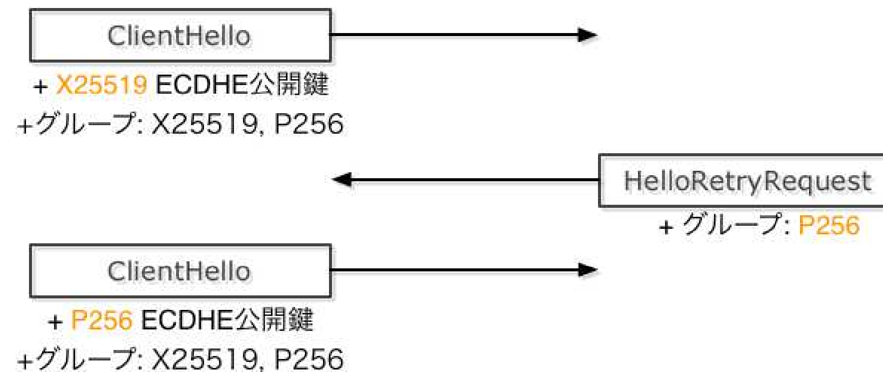


TLS 1.3



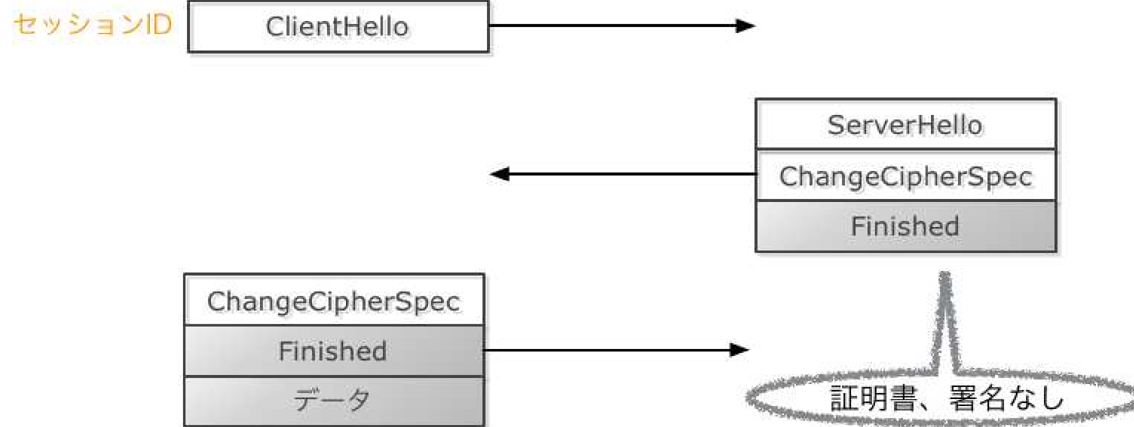
再トライ

- 鍵交換の拡張
  - 複数の公開鍵を送ってもよい
  - 帯域を節約するために1つの公開鍵だけを送ってもよい
- HRR: HelloRetryRequest
  - サーバが送られて来た公開鍵を受けいられれないが、NegotiatedGroup拡張に受け入れられる楕円曲線があれば、再トライを促せる



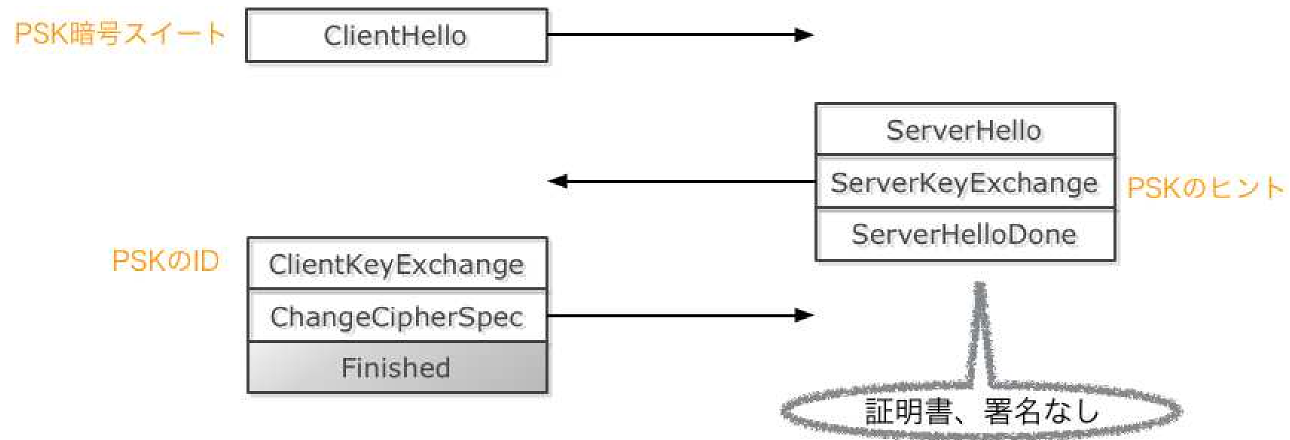
TLS 1.2のセッションの再開

- セッションの状態を再開できる
  - Resumption
  - 公開鍵暗号を使わずに前のセッションの通信相手を認証
  - セッションID: サーバが状態を保つ
  - セッションチケット: サーバが状態を持たない



TLS 1.2の Pre-Shared Key

- 手で設定した秘密を使う
  - 公開鍵暗号を使わずに定められた通信相手を認証

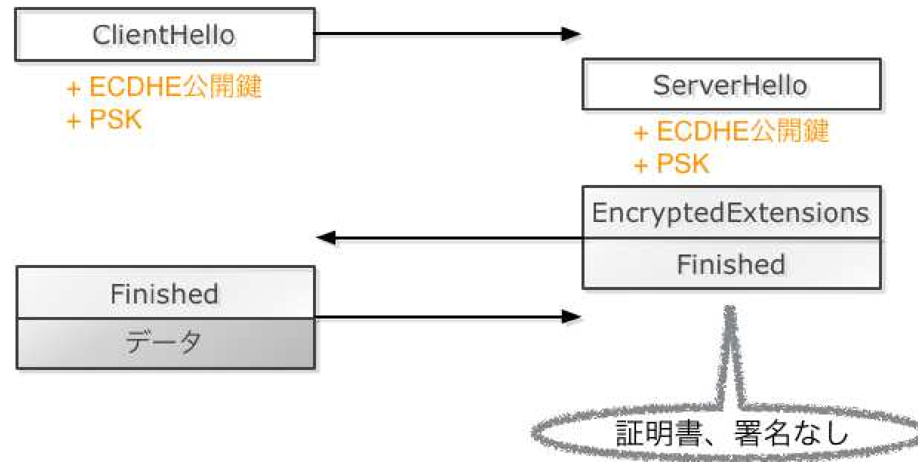


TLS 1.3 での試み

- TLS 1.2 のセッション再開
  - 公開鍵暗号を使わずに前のセッションの通信相手を認証
  - 秘密は前のセッションで共有
  - セッションID、セッションチケット
- TLS 1.2 の Pre-Shared Key
  - 公開鍵暗号を使わずに定められた通信相手を認証
  - 秘密は手で設定
- 統合できるのでは？

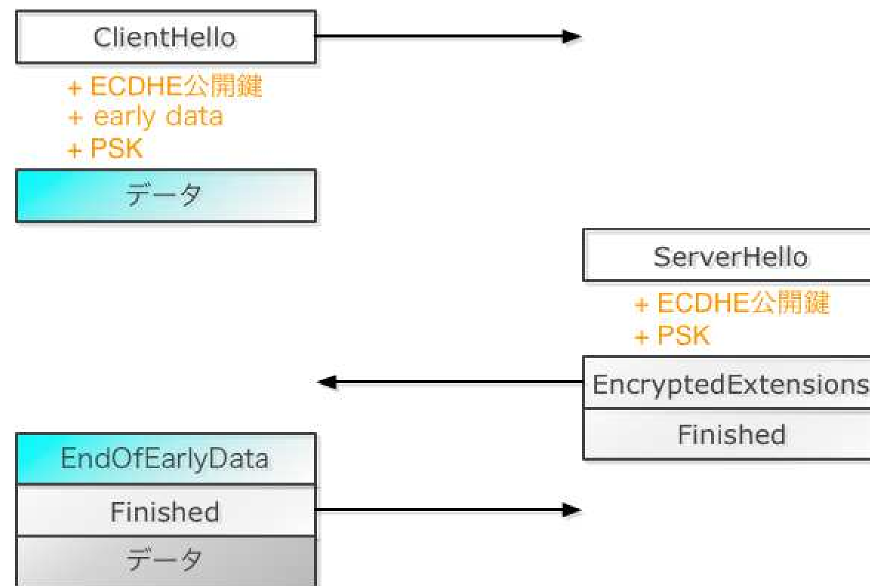
TLS 1.3 のPSK

- 2つの用途
  - 外部 PSK: TLS 1.2 でいう PSK (Pre-Shared Key)
  - 再開 PSK: TLS 1.3 でいう再開 (resumption)
- 前方秘匿性のある1RTT 再開
  - TLS 1.2 の再開は 1RTT だが、前方秘匿性はない



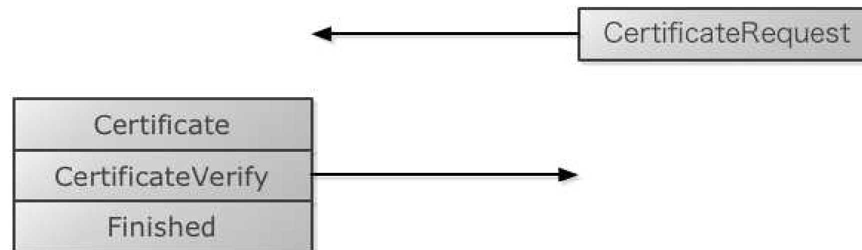
0RTT

- PSK を元に0RTTデータを暗号化
  - (EC)DHEによる共有鍵が得られる前なので前方秘匿性はない
  - リプレイ攻撃される可能性もある



クライアント認証

- サーバは、ハンドシェイク後いつでも要求できる
  - ハンドシェイク時にも要求できる
- クライアントの証明書は暗号化される
  - ハンドシェイク時もハンドシェイク後も



- その他いつでも送れるメッセージ
  - 鍵更新
  - 新しいセッションチケット

TLS 1.3 の実装

存在する実装

- クライアント側
  - picotls: フル、HRR、再開 PSK、0RTT
    - X25519 がないのが唯一の欠点？
  - Firefox Nightly: フル、再開 PSK
  - Chrome Canary: フル、HHR
    - Firefox Nightly よりもちょっと遅れている感じ
- サーバ側
  - picotls
  - tris: Cloudflare
  - NSS: Mozilla
  - BoringSSL: Google
- キャプチャ
  - Wireshark は対応していると言っているが対応してない
- まとめサイト

  https://github.com/tlswg/tls13-spec/wiki/Implementations

Haskell での実装

- Haskell "tls" ライブラリ
  - Haskell で書かれている
    - OpenSSL へのバインディングを提供する他のライブラリも存在している
  - Haskell のフルスタック ウェブ アプリケーションで使われている
- サーバ側
  - 4つのハンドシェイクを実装
  - DHE w/ 5つのFFDHE
  - ECDHE w/ P256、P384、P521、X25519、X448
  - 総合接続性試験
    - picotls: フル、再開 PSK、0RTT
    - Firefox Nightly: フル、再開 PSK
    - Chrome Canary: フル、HRR
- クライアント側
  - 0RTT 以外のハンドシェイクを実装済み

TLS 1.2 だと足らない部品

- RSA PSS
  - RFC 3447
  - Probabilistic Signature Scheme
  - PKCS#1 と PSS は、パディングが違うだけ
  - PSS は安全性が証明されている
  - PKCS#1 な証明書から取り出した公開鍵が PSS に利用できる
- HKDF
  - RFC 5869
  - HMAC-based Extract-and-Expand Key Derivation Function
  - 作るのは簡単
- X25519、X448
  - RFC 7748
  - X25519: 128ビット安全性の楕円曲線暗号
  - X448: 224ビット安全性の楕円曲線暗号
  - 速いかは別として、作るのは簡単

TLS 1.2 との共存問題 (1)

- 再利用されている拡張
  - SignatureScheme vs SignatureAndHashAlgorithm
  - NamedGroup vs NamedCurve
  - 一方にあって他方にない値
  - カスタマイズできるようにしているときの古い値はどうする？
  - たまたま拡張IDが同じ別の拡張と考える
- TLS 1.3 ではメッセージによって構造が違う拡張がある
  - パーサにはハンドシェイクメッセージの型を渡す必要がある

```
struct {
  select (Handshake.msg_type) {
    case client_hello: KeyShareEntry client_shares<0..2^16-1>;
    case hello_retry_request: NamedGroup selected_group;
    case server_hello: KeyShareEntry server_share;
  };
} KeyShare;
```

TLS 1.2 との共存問題 (2)

- ClientHello
  - TLS 1.2 も 1.3 も構造は実質同じ
  - サーバ側は問題なし
- ServerHello
  - TLS 1.2 と 1.3 では構造が違う
  - クライアント側の実装が難しい場合もある
    - TLS 1.2 と 1.3 で型を分けたいが、Server Hello を処理する関数はパーサに TLS 1.2 の値を要求している

0RTT の実装

- サーバがPSKを受け入れずフルにフォールバック
  - サーバは 0RTT のデータを読み飛ばす必要がある
  - クライアントは 0RTT のデータを 1RTT で再送する必要がある

Google grease

- TLS のバージョンアップでは拡張が生命線
  - 拡張パーサは、知らない値を無視しないといけない
    - エラーにしてはいけない
- 日頃から変な値を織り交ぜてテスト
  - Grease と呼ばれる
  - Chrome Canary では実装済み
- Haskell TLS は見事にハマった
  - 後で無視するように変更しようと思って、
    とりあえずエラーにしていた
  - Chrome Canary がどうやっても TLS 1.3 を喋らない
  - Haskell TLS がパースに失敗し拡張がないように見えていた

TLS 1.3 開発日記

- HTTP/2 advent calendar の一環
  - その1 実装状況
  - その2 暗号スイート
  - その3 バージョン
  - その4 フルハンドシェイク
  - その5 Hello Retry
  - その6 Pre Shared Key
  - その7 0RTT
  - その8 開発メモ