HTML5 を利用した Web アプリケーションのセキュリティ問題に関する調査報告書 Rev.2

一般社団法人 JPCERT コーディネーションセンター

2013 年 10 月 30 日

(更新:2014 年 7 月 29 日)

JPCERT CC®

目次

1. はじめに

HTML5 は、狭義には、WHATWG および W3C が HTML4 に代わる次世代の HTML として策定を進めている Web 上の文書を記述するためのマークアップ言語の仕様であり、広義には、通信規格 WebSocket のような関連規格を含む Web アプリケーション記述のための枠組みである。日本を含むアジア太平洋地域において 2012 年 1 月に行われた調査会社 Evans Data 社による調査結果によると、58%のエンジニアが、新規に Web サイトを構築する際に HTML5 を利用すると言っており、急速に普及が進みつつある。こうした強い市場ニーズを受けて HTML5 の実現に、ブラウザ提供者を含む多くのベンダーや組織が取り組んでおり、標準化機関において定義しきれていない仕様の詳細は、ブラウザなどにおける実現が先行し、齟齬も生じている。

HTML5 およびその周辺技術の利用により、Web サイト閲覧者(以下、ユーザ)のブラウザ内でのデータ格納、クライアントとサーバ間での双方向通信、位置情報の取得など、従来の HTML4 よりも柔軟かつ利便性の高い Web サイトの構築が可能となる一方で、それらの新技術が攻撃者に悪用された際にユーザが受ける影響に関して、十分に検証や周知がされているとは言えず、セキュリティ対策がされないまま普及が進むことが危惧されている。

本調査は、HTML5 への移行が Web アプリケーションのセキュリティに及ぼす影響について、現時点における知見を可能な限り体系的に整理し、Web セキュリティ研究者および Web アプリケーション開発者のための基礎資料を提供することを目指して実施された。調査は、Web セキュリティの専門家に委託し、HTML5 で新たに追加された機能に関連して作りこまれる脆弱性や、HTML5 で新たに追加されたセキュリティのための機能を Web アプリケーションの構築時に活用する方法や留意事項を中心に調査を行った。なお、HTML5 の仕様の詳細について実現依存の揺れが存在することを先に述べたが、本報告書は、付録に記載する主要なブラウザについて 2012 年 8 月から 2013 年 2 月にかけて実機で検証した結果に基づいている。

本報告書は次の 6 章で構成されている。

第 1 章の概要に続き、第 2 章では本調査の目的および方法を記述する。

第 3 章では、従来から存在した Web アプリケーションの脆弱性ではあるが、開発において特に注意すべき脆弱性について概要と対策を記述する。この中には、HTML5 で新たに可能になる攻撃手法、従来は不要だったが HTML5 では新たに必要となる Web アプリケーション開発上の考慮事項などが含まれる。

第 4 章では、HTML5 で新たに追加された機能や、HTML5 の Web アプリケーションで使用されることの多い機能の観点から脆弱性が作り込まれる様子と、そうした作り込みを避けるための対策を記述する。

第 5 章では、ブラウザに実装されているセキュリティ上の機能のうち、HTML5 の Web アプリケーションを安全に運用するため特に有効なものについて、これらの機能を使用するための実装方法や設定方法を記述

する。

第 6 章に本報告書全体のまとめを記述する。

本報告書は、Web アプリケーションの開発者に HTML5 を用いた安全な Web アプリケーションの作り方を議論するための出発点となる基本的な資料として利用されることを想定している。HTML5 を用いた Web アプリケーションで使用される機能について焦点を絞っているため、従来からの基本的な Web アプリケーションセキュリティに関する技術に関しては取り扱っていない項目がある。これらの項目に関しては、独立行政法人情報処理推進機構（以下、IPA）が発行する『安全なウェブサイトの作り方』などを参照していただきたい。

また、HTML5 の仕様はなお流動的な部分が少なからず残っており、その動向とそれに伴うセキュリティ問題への影響について今後とも注目していく必要がある。さらに、これまで知られていなかったような HTML5 に固有の攻撃法が新たに見つかる可能性も否定できない。したがって、本報告書は、調査時点での HTML5 において調査時点で発見できたセキュリティ上の問題点と、それを避けるために採用すべき基本的原則をまとめたものであり、読者各位からのフィードバックや新たな知見を加えて定期的に見直していく必要があるものと考えている。

2. 調査の目的と方法

2.1. 調査の目的

本調査の目的は、急速に普及が進み始めようとしている HTML5 を利用した Web アプリケーションの開発時に作り込みが懸念される脆弱性の主なもののうち、HTML5 の Web アプリケーションで使用されやすいものについて、発生の仕組みと発生を回避するために取るべき基本原則を調査し、安全な Web アプリケーション開発のための技術書やガイドラインのベースとなる体系的な資料を提供することである。

2.2. 調査の方法

本調査では、調査時点において仕様の骨格が固まっていた HTML5 の新機能として、次の項目を抽出し、それぞれについて Web アプリケーションセキュリティの専門家による調査を行い、セキュリティ上問題となる事例と対策をまとめた。

- 追加ないし機能が拡張された HTML 要素
  - meta
  - a
  - iframe
  - video
  - audio
  - source
  - canvas
  - input
  - button
- 新たに定義された JavaScript API
  - WebSocket
  - Web Storage
  - Offline Web Application
  - Web Workers
  - Cross Document Messaging

調査では、まず各項目に対して、懸念されるセキュリティ問題を抽出したうえで検討を加え、可能な限り、検証を行ったうえで結果をまとめた。また、上記の項目以外についても、セキュリティ上の問題となる箇所を調査し追記した。

3. HTML5 において特に注意が必要な脆弱性

HTML5 ではブラウザでコンテンツを表現する機能を拡張するため、新しい要素や属性が追加されている。そうした要素や属性の誤った使用のために Web サイトに脆弱性を作り込むばかりでなく、従来はセキュアだった Web アプリケーションが、HTML5 で拡張された機能が攻撃者に利用可能となるために脆弱性を帯びるようになる場合もある。本調査では、そうした脆弱性の中でも特に Web アプリケーション開発者が注意する必要があるものとして、次に挙げる 4 つの脆弱性に焦点を絞って詳細に分析した。

- クロスサイト・スクリプティング
- クロスサイト・リクエスト・フォージェリ
- オープンリダイレクト
- アクセス制御や認可制御の欠落

これらの脆弱性は HTML5 よりも前から注意すべきものとされていたものばかりだが、本章では、これらが HTML5 において特有な発現をする様子と、それを避けるための方法について述べる。HTML5 に依存した脆弱性の全体像を知りたい読者は 0.「
HTML5 における注意が必要な機能」も参照していただきたい。

HTML5 で初めて可能になった重要な仕組みの一つに、クロスオリジン通信がある。オリジンとは、コンテンツの出自を識別するもので、RFC6454 “The Web Origin Concept”で、コンテンツを読み込んだスキームとホストとポートの 3 組であると定義されている。3 組の要素がすべて同じ場合、2 つのコンテンツは同一オリジンであり、3 組の要素の一つでも異なる場合はクロスオリジンであると言う。JavaScript などのコンテンツが、クロスオリジンのコンテンツと行う通信をクロスオリジン通信と呼ぶ。

3.1. クロスサイト・スクリプティング

クロスサイト・スクリプティング(以下、XSS)とは、利用者が信頼している Web サイトに関連して、利用者からは当該サイトが意図的に発行したかのように見えるスクリプトを、当該サイトの意図に反して、ユーザのブラウザに送り込み実行する攻撃法である。XSS 攻撃に対する脆弱性(以下、XSS の脆弱性)は、Web アプリケーションや JavaScript において、種々の入力データを取り込んで、ブラウザによって実行される HTML 文書を動的に生成するための処理の実現方法が適切でない場合に、入力データに紛れ込んだスクリプトを見逃して HTML 文書に組み込んでしまうことにより作り込まれる。XSS の脆弱性を防ぐためには、スクリプトが解釈されるような文脈、あるいは、特定の文字列を挿入すると、スクリプトが解釈される文脈に切り替わるような部分を動的に生成する際に特別な注意を払う必要がある。

HTML5 で HTML の構造が複雑になったことに伴い、従来とは異なった経緯で作り込まれうる XSS の脆弱性として次のものが、調査の結果、抽出された。

(1) 新要素および新属性により引き起こされる XSS
(2) DOM 操作により引き起こされる XSS

(3) XHR Level 2 による XSS
(4) Ajax データによる XSS

ここでは、(1)および(2)の背景事情および対策について簡単に説明する。(3)については 4.3.1.**「意図しないクロスオリジンでのアクセス(クライアント側)」** を、(4)については 4.3.3.**「Ajax データによる XSS」**で解説する。

3.1.1. 新要素および新属性により引き起こされる XSS

従来の HTML でも、onerror イベントハンドラを指定できる要素があり、それらの要素では属性 onerror の値に JavaScript を設定しエラー処理手続きとして実行させることが可能だった。そのため、動的に生成される HTML 文書中の onerror の値に攻撃者が JavaScript を埋め込みエラー状態を引き起こすことができれば、XSS 攻撃が可能だった。HTML5 では、複数の新しい要素が追加されたが、そのうち<video>要素や<source>要素などでは onerror イベントハンドラがサポートされているため、以下ののような HTML においては指定された JavaScript が実行される。

```
<video onerror="javascript:alert(1)">
    <source onerror="javascript:alert(1)">
</video>
```

また、従来の HTML でも、<input>要素などの要素の動的な生成に際しては「<」「>」をエスケープしていても、value 属性の生成で、引用符「"」「'」に対するエスケープ漏れがあるために XSS 攻撃が可能になる。しかし、従来の HTML では、要素中に指定できるイベントハンドラは、イベントを発生させるために、マウスを動かしたりクリックしたりするユーザの操作が必要な onmouseover や onclick といったものが多数を占めた。

```
<input value="" onmouseover="alert(1)">
```

ところが、HTML5 で追加された一部の属性と、それらユーザの操作が必要なものとを組み合わせることで、ユーザの操作なしに XSS 攻撃を実行することが可能となっている。例えば、<input>要素に、Web ページが表示された際に当該入力欄にフォーカスを移動する autofocus 属性が追加されたため、同要素中の onfocus 属性に指定されたイベントハンドラと組み合わせることでユーザの操作なしに実行されるようになった。

```
<input value="" autofocus onfocus="alert(1)">
```

また、従来の HTML では、値にスクリプトを指定することができる属性は on で始まる属性名をもつ属性に限られていた。ところが、HTML5 では、formaction 属性のように on で始まらない名前をもち、値としてスクリプト

を指定できる属性が追加された。さらにブラウザ独自の実装や、仕様の追加などにより on で始まらない属性が追加される可能性もある。このため、on で始まる属性を検出して、その部分だけを無害化するという対策では、XSS の脆弱性を防ぎきれない。

```
<form>
    <button formaction="javascript:alert(1)">
        some text
    </button>
</form>
```

以上のことから、XSS 攻撃の対策としてはユーザの入力から危険な要素や属性を検出するといった手法は避け、HTML 生成時にエスケープすることを推奨する。

より具体的な対策方法については、IPA『安全なウェブサイトの作り方』の 1.5 **「クロスサイト・スクリプティング」**の章などを参照していただきたい。

3.1.2. DOM 操作により引き起こされる XSS

DOM (Document Object Model)とは、Web サーバと通信をすることなく、ブラウザ上に表示されている HTML 文書をブラウザ上で動作するスクリプトが生成ないし更新できるようにするために定義された API であり、これにより、手元の PC 上で稼働しているアプリケーションに近い即応性のある Web アプリケーションを実現することができる。DOM Based XSS とは、攻撃者が入力データなどに細工して忍び込ませたスクリプトを、ブラウザ上で動作している JavaScript などのスクリプトが不用意に取り込んで Web ページ中に DOM 操作を通じて組み込み、それがブラウザ上で実行されるという攻撃法である。HTML5 を用いた Web アプリケーションは JavaScript が多く使用される傾向にあることから DOM Based XSS の脆弱性が作り込まれる場合がある。

DOM Based XSS 攻撃の対策を、3.1.2.1. 「HTML 生成時は DOM 経由で操作を行う」および 3.1.2.2. 「URL を取り扱うときは http あるいは https に限定する」で述べる。

3.1.2.1. HTML 生成時は DOM 経由で操作を行う

DOM Based XSS 攻撃に対する対策としても、「HTML 生成時のエスケープ」の原則は有効であるが、テキストノードを生成して DOM 経由で操作する方がより網羅的に XSS 攻撃の発生を防ぐことができる。なお、JavaScript における「HTML 生成時」とは、innerHTML への代入や document.write によるドキュメントへの書き込みなど、JavaScript 内で DOM 構造が変更されるタイミングのことを指す。

```
// 脆弱な例
// 外部からの文字列をそのまま HTML として使用することで XSS 攻撃が可能である
var div = document.getElementById("msg");
div.innerHTML = some_text; // 外部からの文字列
```

```
// 安全な例
// 外部からの文字列をエスケープして HTML を生成
function escape_html( s ){
    return s.replace( /&/g, "&amp;" )
        .replace( /</g, "&lt;" )
        .replace( />/g, "&gt; " )
        .replace( /"/g, "&quot;" )
        .replace( /'/g, "&#39;" );
}

var div = document.getElementById("msg");
var escaped_text = escape_html( some_text );
div.innerHTML = escaped_text; // エスケープ済み文字列
// 安全な例（推奨）
// テキストノードを DOM API を介して操作することで安全に HTML を生成
var div = document.getElementById("msg");
var text = document.createTextNode( some_text ); // テキストノード生成
div.appendChild( text );
```

テキスト部分だけでなく、次のように属性値にも注意をする必要がある。

```
// 脆弱な例
// 外部由来の文字列をそのまま属性値として使用することで XSS 攻撃が可能である
var f = document.getElementById("form");
f.innerHTML = "<input type='text' value='" + some_text + "'>";
```

```
// 安全な例（推奨）
// DOM API を経由して属性値を設定する
var f = document.getElementById("form");
var elm = document.createElement( "input" );
elm.setAttribute( "type", "text" );
elm.setAttribute( "value", some_text ); // 属性値の設定
```

```
f.appendChild( elm );
```

3.1.2.2. URL を取り扱うときは http あるいは https に限定する

<a>要素の href 属性や<iframe>要素などの src 属性、location.href への代入などにおいて URL を取り扱う場合に、javascript スキームなどの http あるいは https 以外の任意のスキームが攻撃者によって指定される場合 XSS 攻撃が行われる可能性がある。

```
// 脆弱な例 1
// 外部からの文字列をそのまま URL として使用することで XSS 攻撃が可能である
// 攻撃者は変数 url に "javascript:..." といった文字列を指定可能とする
location.href = url;
```

```
// 脆弱な例 2
// 外部からの文字列をそのまま URL として使用することで XSS 攻撃が可能である
// 攻撃者は変数 url に "javascript:..." といった文字列を指定可能とする
var elm = document.getElementById("link"); // <a id="link">
var text = document.createTextNode( url );
elm.appendChild( text );
elm.setAttribute( "href", url ); // href に"javascript:..."が設定される
```

外部由来の文字列を URL として使用する場合には、それが http あるいは https であることが明確な場合に限定することで、このような XSS 攻撃を防ぐことができる。

```
// 安全な例
// URL が"http://"または"https://"で始まっている場合のみ処理する
if( url.match( /^https?:¥/¥// ) ){
    var elm = document.getElementById("link"); // <a id="link">
    var text = document.createTextNode( url );
    elm.appendChild( text );
    elm.setAttribute( "href", url );
}
```

ただし、URL が http://または https://で始まっているものに限定した場合でも、location オブジェクトに代入する場合には、XSS 攻撃は発生しないがオープンリダイレクトが発生する可能性がある。詳細は 3.3.「**オープンリダイレクト**」で解説する。

DOM Based XSS の脆弱性に関しては、IPA が発行する『IPA **テクニカルウォッチ『DOM Based XSS』に関するレポート**』も参照していただきたい。

3.2. クロスサイト・リクエスト・フォージェリ

クロスサイト・リクエスト・フォージェリ（以下、CSRF）攻撃は、攻撃者が用意したページに被害者ユーザを誘導することにより、意図しない Web サイトから意図しない操作を被害者ユーザに行わせる攻撃手法である。XHR がクロスオリジンでのリクエストをサポートしたことにより、クロスオリジン通信での CSRF 攻撃が可能となった。そのため、従来は CSRF の脆弱性がなかった Web サイトにも CSRF 攻撃が行われる可能性がある。

例えば、ファイルのアップロードを行う機能に対して攻撃者が攻撃を行うケースについて、HTML4 の機能のみを使用した攻撃手法と、HTML5 の新機能を使用した攻撃手法について考える。このファイルのアップロード機能では、次のような HTML を使用しておりサーバ側ではリファラのチェックが行われていないとする。次のフォームが送信するデータにはトークンのような情報が見当たらないため、この機能には CSRF の脆弱性があると考えられる。

```
<!-- http://target.example.jp/ 上のコードの例 -->
<form method="POST" action="upload" enctype="multipart/form-data">
    <input type="file" name="file">
    <input type="submit">
</form>
```

HTML5 の機能を使用しない場合、JavaScript を使用してクロスオリジンのリクエストを送ることができない。そのため、攻撃者は次のような、ページロード時にクロスオリジンで攻撃対象の正規サイトに対してデータを POST する悪意のある Web サイトを作成し、ユーザの誘導を狙った CSRF 攻撃を仕掛けるとする。

```
<!-- 悪意ある Web サイト上のコードの例 -->
<body onload="document.forms[0].submit();">
<form method="POST" action="http://target.example.jp/upload"
     enctype="multipart/form-data">
    <input type="file" name="file">
    <input type="submit">
</form>
</body>
```

ところが、実際にはこの悪意のある Web サイトをユーザが開いたとしても、次のようにファイルの内容、ファイル名ともに空のまま送信されることになり、有意なデータをサーバへと送信することはできない。

```
POST http://target.example.jp/upload
Host: target.example.jp
...
Content-Type: multipart/form-data; boundary=----abcdefg

------abcdefg
Content-Disposition: form-data; name="file"; filename=""
Content-Type: application/octet-stream



------abcdefg--
```

一方で、HTML5 を使用する場合、XHR がクロスオリジン通信に対応したことにより、XHR によって POST リクエストを送信することが可能になり、アップロードするファイルの内容を次のように JavaScript 内で自由に組み立てることができる。

```
var xhr = new XMLHttpRequest();
var boundary = '----boundary';
var file="abcd";   //送信するファイルの内容
var request;

xhr.open( 'POST', 'http://target.example.jp/upload', 'true' );
xhr.setRequestHeader( 'Content-Type',
    'multipart/form-data; boundary=' + boundary );
xhr.withCredentials = true;       // Cookieを付与
xhr.onreadystatechange = function(){};
request = '--' + boundary + '¥r¥n' +
    'Content-Disposition: form-data; name="file"; ' +
    ' filename="filename.txt"¥r¥n' +
    'Content-Type: application/octet-stream¥r¥n¥r¥n' +
    file +
    '¥r¥n' + '--' + boundary + '--';
xhr.send( request );
```

HTML5 では、このように、ファイルの内容およびファイル名を自由に JavaScript 内で組み立てて攻撃対象となるサーバへ CSRF の脆弱性を使用してアップロードすることが可能となる。なお、XHR でのクロスオリジンの通信では通常 Cookie は送信されないが、withCredentials プロパティを true に設定することで Cookie が送信されるようになる。ファイル名やファイルコンテンツの表示部分に XSS の脆弱性が存在する場合には、これらを組み合わせての攻撃も可能である。

自サイトが XHR のクロスオリジン通信を許可していない場合や、クロスオリジン通信に対応していないという場合であっても、攻撃者の用意した罠ページからのリクエスト自体は送ってこられる可能性がある。そのため、自サイトに対して送られたデータを無条件に処理している場合には攻撃に使用される可能性がある。

CSRF 攻撃の対策に他サイトから送信された場合のリクエストに含まれる Origin ヘッダを確認するという方法は、従来どおり自サイト form から送信されたリクエストによる CSRF 攻撃には対応できないため、根本的な対策とはならない。XHR を利用した CSRF 攻撃への対策としては、従来の CSRF 攻撃の対策と同様、トークンなどの秘密情報を<input type="hidden">要素に含めるなどの手段を用いる。

```
<body onload="document.forms[0].submit();">
<form method="POST" action="http://target.example.jp/upload"
     enctype="multipart/form-data">
    <input type="file" name="file">
    <input type="hidden" name="token" value="9CF89BC43B1B6FEA399A...">
    <input type="submit">
</form>
</body>
```

CSRF 攻撃のより具体的な対策方法については、IPA『安全なウェブサイトの作り方』の 1.6 「CSRF(クロスサイト・リクエスト・フォージェリ)」の章などを参照していただきたい。

3.3. オープンリダイレクト

Web アプリケーションで、入力値などのデータに応じてリダイレクト先を変えたい場合がある。この時、不用意な Web アプリケーションの作り方をすると、リダイレクト先を攻撃者が自在に変更して、被害者を悪意ある Web サイトに誘導することが可能になってしまう。この脆弱性をオープンリダイレクトと呼ぶ。オープンリダイレクトは Web サイトそのものに被害を及ぼすわけではないが、その Web サイトのドメイン名を使用して悪意あるサイトへユーザが誘導されるなど、Web サイトの信用問題に繋がるため対策が必要である。location.hash を使用してリダイレクト先を指定する手法では、リダイレクト先 URL がサーバ側のログに記録されないため、オープンリダイレクトが存在する場合には、アクセス先が把握できず後から状況を確認できない可能性があるため特に注意が必要である。

オープンリダイレクトを発生させないためには、リダイレクト先 URL を外部からの入力によって生成するのではなく、事前に固定のリストとしてプログラム内で持っておくことが最も有効である。

```
#!/usr/bin/perl
# 安全な例
# リダイレクト先（/foo、/bar、/baz）を事前に固定して保持
use URI::Escape;
my $index = uri_unescape( $ENV{QUERY_STRING} || '' );
my $pages = { foo=>'/foo', bar=>'/bar', baz=>'/baz' };
my $url = $pages->{$index} || '/';
print "Status: 302 Found¥n";
print "Location: $url¥n¥n";
```

リダイレクト先の URL を事前に固定できず動的に生成せざるをえない場合には、任意サイトへリダイレクトされないように、生成後のリダイレクト先 URL のドメインが想定される範囲に含まれるかどうかチェックする必要がある。

Web アプリケーションで、動的に生成した URL にリダイレクトするための、代表的な 3 種類の方法とそれぞれに関連する注意事項を次に述べる。

3.3.1. HTTP ステータスコードとして 301 または 302 などを返し、リダイレクト先を Location ヘッダで指定

この方法によるリダイレクトでは、オープンリダイレクトだけでなく HTTP ヘッダインジェクションにも注意する必要がある。

```
#!/usr/bin/perl
# 脆弱な例
# オープンリダイレクトだけでなく HTTP ヘッダインジェクションもある
use URI::Escape;
my $url = uri_unescape( $ENV{QUERY_STRING} || '' );
print "Status: 302 Found¥n";
print "Location: /$url¥n¥n";
```

3.3.2. JavaScript による location オブジェクトへの代入

この方法によるリダイレクトでは、スキームを http あるいは https に限定する必要がある。任意の JavaScript を参照するスキームにリダイレクトされて XSS 攻撃が行われる可能性があるからである。詳細は、3.1.2. 「**DOM 操作により引き起こされる XSS**」を参照していただきたい。

```
// 脆弱な例１
// 外部由来の文字列をそのまま URL として使用することで XSS 攻撃が可能である
// 攻撃者は変数 url に "javascript:..." といった文字列を指定可能
var url = decodeURIComponent( location.hash.substring(1) );
location.href = url;
```

```
// 脆弱な例２
// 外部由来の文字列の確認が不十分なため、オープンリダイレクトにつながる。
// 攻撃者が「/¥example.com/」のような形式で URL を指定することで任意の
// サイトへリダイレクト可能
var url = decodeURIComponent( location.hash.substring(1) );
// URL の先頭が/、2 文字目が/以外であることをチェック。
if( url.match( /^¥/[^¥/]/ ) ) {
    location.href = url;
}
```

```
// 安全な例
// リダイレクト先を固定のリストとして保持。
var index = location.hash.substring(1) | 0;
var pages = [ '/foo', '/bar', '/baz' ]; // リダイレクト先のリスト
if( 0 <= index && index < pages.length ){
    location.href = pages[ index ];
}
```

3.3.3. <meta http-equiv="Refresh">を用いたリフレッシュによるリダイレクト

ユーザからの入力をもとに動的に生成した URL を用いて、この方法でリダイレクトする場合には、オープンリダイレクトを防ぐことが困難である。したがって<meta>リフレッシュによるリダイレクトは使用しないことが望ましい。詳細については 4.1.2.「**<meta http-equiv>**」で述べる。

3.4. アクセス制御や認可制御の欠落

一般には公開しない個人情報を扱う Web サイトにアクセス制御や認可制御の不備があると情報漏えいを引き起こす可能性がある。HTML5 で追加された機能を攻撃者が利用すると、従来はアクセスすることができなかった情報がアクセス可能になる場合がある。

例えば、Ajax では、攻撃者の用意した悪意ある Web サイトに<script>要素の src 属性を使用して秘密情報を含む JSON を読み込ませ、JavaScript として解釈させることにより JSON 内の秘密情報にアクセスするなどの、取り扱うデータに含まれる秘密情報を窃取するための様々な攻撃手法が見つかっている。また、従来は許されなかったクロスオリジン通信を利用して、本来はアクセスさせたくない第三者が秘密情報にアクセスする可能性もある。Web サイトの中には、これらの HTML5 の機能を考慮しておらず、不適切に設計し運用しており、開示制限すべき情報を公開しているものがある。一般には公開しない秘密情報を扱う Web サイトを HTML5 を用いて構築する場合には、適切にアクセス制御や認可制御を実装する必要がある。その対策の詳細は、攻撃に使用される機能により大きく異なるため、第 4 章「
HTML5 における注意が必要な機能」で述べる。

4. HTML5 における注意が必要な機能

本章では、次に挙げる HTML5 で新たに追加された機能や、使用されることの多い HTML5 の機能ごとに、脆弱性が作り込まれる様子と、そうした作り込みを避けるための対策を解説する。

- 新しく追加された HTML 要素
- JavaScript API
- XMLHttpRequest

4.1. 新しく追加された HTML 要素

本節では、HTML5 で追加、変更されたものを中心に、HTML 要素、属性ごとに、利用にあたりセキュリティ上注意すべき点について解説する。

4.1.1. <meta charset>

<meta charset>は、HTML ファイル内で文字エンコーディングを指定するために使用される。

```
<head>
    <meta charset="utf-8">
</head>
```

埋め込んだ HTML の特殊文字を文字エンコーディングにより隠すことで、UTF-7 などの文字エンコーディングを使用した XSS 攻撃が可能になる。したがって、文字エンコーディング指定の不正な変更を防ぐことが重要である。そのため、文字エンコーディング名は HTTP レスポンスヘッダにおいて明確に指定することを原則とし、HTML 内での<meta charset>による指定は、コンテンツをローカルに保存した場合の文字化けの防止のためなど補助的な使途だけに限ることを推奨する。文字エンコーディング名の指定は、ブラウザ側が正しく判断できるよう、ハイフンあるいはアンダースコアも含めて誤記のない「utf-8」「Shift_JIS」「EUC-JP」などのような正確な表記によることが求められる。

文字エンコーディングの指定にレスポンスヘッダが使用できず<meta charset>で指定しなければならない場合には、<meta charset>より先行する部分に外部からの入力を表示してはならない。なぜなら、本来の<meta charset>の前に攻撃者が挿入した偽の<meta charset>をブラウザが解釈した場合、文字エンコーディングの認識が Web サイト側とブラウザ側とで食い違いを生じ、XSS 攻撃が行われる可能性がある。

次の例では、攻撃者が、<title>要素として UTF-7 で表記した「</title><meta charset="utf-7">」という文字列を指定している。この例では JavaScript は動かないが、他の手法を組み合わせれば XSS 攻撃が行われる可能性がある。

```
<title>
    +/v8APA-/title+AD4APA-meta charset+AD0AIg-utf-7+ACIAPg-
</title>
<meta charset="utf-8">...
<div>+ADw-script+AD4-alert(1);+ADw-/script+AD4-</div>
```

4.1.2. <meta http-equiv>

<meta http-equiv>は一般的には HTML ファイル内で HTTP レスポンスヘッダを補完するために使用される。すなわち<meta http-equiv="name" content="content">の形式は、HTTP レスポンスヘッダに name: content というフィールドを追加したのと同じ働きをする。本節では、次のような構文により指定される meta リフレッシュと呼ばれるリダイレクト機能に関して述べる。

```
<!-- http://example.jp/ へのリダイレクトの例 -->
<meta http-equiv="Refresh" content="0;url=http://example.jp/">
```

この例では、0 秒後すなわち即刻に「url=」で指定された「http://example.jp/」にリダイレクトされる。リダイレクト先 URL を指定する部分に攻撃者が自由に文字列を挿入できる場合、javascript スキームへのリダイレクトによる XSS 攻撃や任意サイトへのリダイレクト(オープンリダイレクト)の攻撃などにつながる可能性がある。

例えば、次のような javascript スキームへのリダイレクトは、Google Chrome や Opera、Safari のブラウザでは、javascript スキームで指定されたスクリプトが元ページ上で動作するため、XSS 攻撃が発生する可能性がある。

```
<!-- javascript スキーム へのリダイレクトの例 -->
<meta http-equiv="Refresh" content="0;url=javascript:alert(1)">
```

また、次の例のように content 属性の値に「;url=」が複数含まれる場合、Internet Explorer 6 および 7 では末尾の「;url=」以降の文字列が示す URL にリダイレクトされる。<meta http-equiv>によるリダイレクトで、攻撃者がリダイレクト先 URL を自由に指定できる状況では、リダイレクト先ドメインを制限することは難しく、オープンリダイレクトを防ぐことは困難である。したがって、<meta http-equiv>によるリダイレクトを使用する場合は、攻撃者がリダイレクト先 URL を自由に指定できないように固定値とすることが望ましい。

```
<!-- 意図しないサイトへのリダイレクトの例 -->
<meta http-equiv="Refresh"
 content="0;url=http:example.jp/;url=http://evil.example.jp/">
```

4.1.3. <a ping>

<a>要素の ping 属性は、ユーザがリンクをクリックしたことを通知すべき通知先リソースの URL を指定するための属性である。

```
<!--  リンクのクリックで ping の URL へ通知される  -->
<a href="http://example.jp/" ping="http://example.jp/ping">
example.jp へのリンク
</a>
```

ping 属性による通知に対応しているブラウザは Google Chrome および Safari である。通知先リソースとしては、http および https スキームの URL のみが有効で、それ以外の例えば"javascript:alert(1)"のようなスキームは無視される。

4.1.4. <iframe sandbox>

<iframe>要素では、sandbox 属性によって、iframe 内のコンテンツに JavaScript の実行禁止などの条件を指定することができ、信頼できないコンテンツを Web サイト内に埋め込む場合などに利用される。

```
<!-- sandbox 属性をもつ iframe の例  -->
<!-- script の実行や form の submit などが禁止される。  -->
<iframe sandbox src="http://evil.example.com/"></iframe>
```

sandbox 属性が付与された iframe 内のコンテンツに対し、ブラウザは、プラグインの実行、JavaScript の実行、form の実行、トップレベルウィンドウへの干渉を制限し、コンテンツを独自オリジンとして取り扱う。

sandbox 属性を持つ<iframe>要素を用いて Web サイトを埋め込むことで、その Web サイトの JavaScript の実行を禁止できるが、自身のページを<iframe sandbox>で読み込むようにしても XSS 攻撃などを防ぐ対策にはならない。XSS の脆弱性などが存在するページがあれば、攻撃者が iframe を用いず、脆弱性を持つページを直接開くことで攻撃ができるからである。

sandbox 属性では、値として iframe 内のコンテンツに与える条件を指定することもできる。

```
<!-- sandbox 属性をもつ iframe の例  -->
<!-- script の実行は許可される。  -->
<iframe sandbox="allow-scripts" src="http://evil.example.com/">
</iframe>
```

sandbox 属性に複数の値を組み合わせて指定する場合には、各値をスペースで区切って列記する。

```
<!-- sandbox 属性をもつ iframe の例  -->
<!-- script の実行、form の実行は許可される。  -->
<iframe sandbox="allow-scripts allow-forms"
 src="http://evil.example.com/"></iframe>
```

また、<iframe>などによるフレーム内でページが表示されることを避けるために、これまではフレームバスティングと呼ばれる JavaScript コードが利用されることがあった。しかしながら、ページが<iframe sandbox>で読み込まれた場合には JavaScript が動作しないため、フレームバスティングは正常に動作せず、ページが表示されてしまう。したがって、クリックジャッキングを防止するためには、フレームバスティングではなく、X-Frame-Options ヘッダを使用する必要がある。詳細については 5.3.「**X-Frame-Options**」で解説する。

4.1.5. <video>

<video>要素は、<img>要素によって画像を埋め込むのと同様に、HTML 内に直接再生可能な動画を埋め込むために使用される。Internet Explorer 9 および 10 では、<video>要素の onerror イベントが動作するため、XSS 攻撃が行われる可能性がある。

```
<!-- video 要素による XSS 攻撃の例 -->
<video onerror="javascript:alert(1)">
    <source src="#">
</video>
```

4.1.6. <audio>

<audio>要素は、<img>要素によって画像を埋め込むのと同様に、HTML 内に直接再生可能な音声を埋め込むために使用される。Internet Explorer 9 および 10 では、<audio>要素の onerror イベントが動作するため、XSS 攻撃が行われる可能性がある。

```
<!-- audio 要素による XSS 攻撃の例 -->
<audio onerror="javascript:alert(1)">
    <source src="#">
</audio>
```

4.1.7. <source>

<source>要素は、<video>要素や<audio>要素の中に置いて、提供できるメディアソースを指定するために使用される。Firefox、Google Chrome、Opera、Safari、Android 標準ブラウザの各ブラウザでは、<source>要素の onerror イベントが動作するため、XSS 攻撃が行われる可能性がある。

```
<!-- video 要素と source 要素による XSS 攻撃の例 -->
<video>
    <source onerror="javascript:alert(1)">
</video>
<!-- audio 要素と source 要素による XSS 攻撃の例 -->
<audio>
    <source onerror="alert(1)">
</audio>
```

4.1.8. <canvas>

<canvas>要素は、JavaScript が図形を描画するための描画領域(キャンバス)を定義するために使用される。

Canvas では、JavaScript から自由に画像を読み書きできるが、オリジンが異なるスクリプトが書いた画像の読み取りを制限するために、内部に origin-clean と呼ばれるフラグを持っている。origin-clean フラグは初期状態では true に設定されているが、オリジンの異なる画像が描画された時点で自動的に false に設定される。origin-clean が false になると、toDataURL メソッド、getImageData メソッド、toBlob メソッドを用いて Canvas 内の画像を JavaScript から読み出すことができない。

```
// http://example.jp/上のコード
// 異なるオリジンの画像の読み込みでセキュリティ例外が発生する
<img id="img" src="http://other.example.jp/sample-image.png">
<canvas id="canvas"></canvas>
...
var img = document.getElementById( "img" );
var canvas = document.getElementById( "canvas" );
var ctx = canvas.getContext( "2d" );
ctx.drawImage( img, 0, 0 );
alert( canvas.toDataURL( "image/png" ) );  // 例外が発生
```

この仕組みにより、Canvas の利用においてはセキュリティ上の問題は発生しにくくなっている。

一方で、オリジンを超えて画像を利用したいという要望に対応するため、Firefox および Google Chrome、Safari 6、Opera では、XHR によるクロスオリジン通信と同様、Cross-Origin Resource Sharing (CORS)の仕組みに従い、画像提供側が Access-Control-Allow-Origin レスポンスヘッダを付与することで、アクセスできるようにしている。すなわち、other.example.jp 上で画像のレスポンスヘッダにおいて、リソースの読み取りを許可するオリジンを Access-Control-Allow-Origin レスポンスヘッダによって次のように指定する。

```
Date: Tue, 1 Jan 2013 09:00:00 GMT
Access-Control-Allow-Origin: http://example.jp
Content-Length: 28400
Content-Type: image/png
```

同時に example.jp 上でも、Canvas への転送元となる<img>要素に、crossorigin 属性を与える必要がある。

```
// http://example.jp/上のコード
// 異なるオリジンの画像を読み込む
// セキュリティ例外は発生しない
<img id="img"
  src="http://other.example.jp/sample-image.png" crossorigin>
<canvas id="canvas"></canvas>
...
var img = document.getElementById( "img" );
var canvas = document.getElementById( "canvas" );
var ctx = canvas.getContext( "2d" );
ctx.drawImage( img, 0, 0 );
alert( canvas.toDataURL( "image/png" ) );  // 例外は発生せず
```

crossorigin 属性のついた<img>要素では、どのページからリクエストが発行されたかのオリジンを示す Origin リクエストヘッダが画像のリクエスト時に付与される。サーバ側は、どのオリジンであればコンテンツにアクセス可能かを Access-Control-Allow-Origin レスポンスヘッダで指定する。上の例では、http://example.jp をオリジンとする JavaScript であれば画像にアクセスすることが可能である。画像のレスポンスヘッダに Access-Control-Allow-Origin が存在しない場合や Access-Control-Allow-Origin に自コンテンツのオリジンが含まれていない場合は、Canvas に描画された画像データにアクセスすることはできない。

特定のオリジンからのみ画像コンテンツへのアクセスを許可したい場合には、Origin リクエストヘッダおよび Access-Control-Allow-Origin レスポンスヘッダの指定に加えて、従来どおり Cookie を使用しなければならない。Cookie を使用せず、Origin リクエストヘッダおよび Access-Control-Allow-Origin レスポンスヘッダの指定だけでは、攻撃者が、ブラウザの替わりにツールなどを使用して任意の Origin リクエストヘッダを付与したリクエストを送信することにより、JavaScript を経由せずに直接画像にアクセスすることが可能だからである。

Access-Control-Allow-Origin: * という指定は、全てのオリジンからの JavaScript が画像にアクセス可能なことを意味しており、画像を第三者に見せたくない場合にはこの指定をしてはいけない。攻撃者が crossorigin 属性のついた<img>要素を使用した罠ページを用意すれば、罠ページ内の JavaScript から画

像にアクセスできるからである。また、Cookie を使用した場合 Access-Control-Allow-Origin: * という指定をすることは仕様上禁止されている。

Access-Control-Allow-Origin: * という指定は、アクセス保護などを必要としない完全に公開するコンテンツにのみ使用し、それ以外は Access-Control-Allow-Origin: http://example.jp のように、許可するオリジンを指定する。

4.1.9. <input>

<input>要素は、HTML5 において大幅に機能が強化され、例えば<input type="email">などの属性を指定することで、メールアドレスの形式の入力データだけを受け付けることが簡単に実現できる。また、<input type="text" pattern="^[0-9a-fA-F]$">のように開発者が任意のパターンを指定して、パターンにマッチする入力だけに制限することもできる。従来であれば入力内容を確認する JavaScript が必要とされた場面で JavaScript を使わずに同等のことができる。この機能は、コード量の低減だけでなく、エラーメッセージが統一された表示になるなどの利点をもたらす。

ただし、攻撃者はブラウザ内で HTML を変更して、<input>要素で指定されたパターンに合致しないリクエストを送信することができるので、細工された入力データを<input>要素での入力制限によって完全に排除できると考えてはいけない。

<input>要素では、「<」「>」がエスケープされていても、value 属性の引用符「"」「'」のエスケープが漏れていると XSS 攻撃の可能性がある。そのような XSS 攻撃の多くは、従来は onmouseover や onclick といったイベントハンドラとして指定された JavaScript によって実行されていたので、攻撃が実際に行われるにはユーザ自身の操作が必要だった。

```
<!-- ユーザの操作が必要な攻撃の例 -->
<input value="" onmouseover="alert(1)">
```

HTML5 では<input>要素に自動でフォーカスを移動する autofocus 属性が追加されたため、これを次の例のように onfocus イベントハンドラと組み合わせることで、ユーザの操作なしに JavaScript の実行が可能となった。

```
<!-- ユーザの操作が不要な攻撃の例 -->
<input value="" autofocus onfocus="alert(1)">
```

また、生成する HTML へのイベントハンドラの挿入を禁ずるために、on で始まる属性を検出する対策をとる場合があったが、HTML5 では formaction 属性のように、javascrpt スキームを値にできる on で始まらない要素や属性が追加されたため、このような対策では不十分である。

```
<!-- on で始まらない属性を利用した攻撃の例 -->
<form>
    <input type="button" formaction="javascript:alert(1)">
</form>
```

4.1.10. <button>

<button>要素にも<input>要素同様、formaction 属性のように on で始まらない要素や属性が HTML5 で追加されたため、on で始まる属性を検出するという XSS 攻撃対策は不十分である。

```
<form>
    <button formaction="javascript:alert(1)">
        some_text
    </button>
</form>
```

4.2. JavaScript API

近年の Web ブラウザは、様々な機能を JavaScript から呼び出すための API を実装している。本節ではそれらのうち、セキュリティ上注意すべきものについて述べる。なお、XMLHttpRequest(XHR)については注意すべき事項が多いため、4.3.「**XMLHttpRequest**」にて別途取り上げる。

4.2.1. WebSocket

WebSocket は Web ブラウザとサーバ間での双方向通信を実現するための機能である。WebSocket を利用する場合の典型的な JavaScript コードを次に示す。

```
var ws = new WebSocket( "ws://example.jp/" );
ws.onopen = function(){
    console.log( 'connected' );
};
ws.onmessage = function( evt ){
    console.log( evt.data );
};
ws.onclose = function(){
    console.log( 'closed' );
};
```

```
ws.send( 'data' );
```

ws URI スキームでは、http スキームと同様に通信内容が暗号化されない。TLS で暗号化された通信経路を利用したい場合には、wss URI スキームを使用する。

```
var ws = new WebSocket("wss://example.jp:443/ ");
```

盗聴や改ざんによる影響の大きい、重要な情報の送受信には wss の使用が推奨される。

ws、wss URI スキームとも、接続時のハンドシェイク時に http および https スキームと Cookie を共有することに注意が必要である。すなわち、http://example.jp/ の Web ページへの通常のアクセス時に発行された Cookie は、ws://example.jp:8080/ や wss://example.jp:8081/ などの利用においてもハンドシェイクのリクエスト時にサーバへ送信される。

```
GET / HTTP/1.1
Host: example.jp
Connection: Keep-Alive

HTTP/1.1 200 OK
Date: Tue, 1 Jan 2013 09:00:00 GMT
Content-Length: 2524
Content-Type: text/html; charset=utf-8
Set-Cookie: session=12AFE9BD34E5A202; path=/
....

GET /websocket HTTP/1.1
Upgrade: websocket
Connection: Upgrade
Host: example.jp
Origin: http://example.jp
Sec-WebSocket-Key: mU6OBz5GKwUgZqbj2OtWfQ==
Sec-WebSocket-Version: 13
Sec-WebSocket-Protocol: chat, superchat
Cookie: session=12AFE9BD34E5A202

HTTP/1.1 101 Switching Protocols
Upgrade: WebSocket
Connection: Upgrade
Sec-WebSocket-Accept: IsCRPjZ0Vshy2opkKOsG2UF74eA=
```

```
Sec-WebSocket-Protocol: chat
....
```

secure フラグを付与して発行された Cookie は、https と wss URI スキームの間で共有される。すなわち https://example.jp/ へのアクセス時に secure フラグを付与して発行された Cookie は、wss://example.jp:8081/などの利用においてもハンドシェイクのリクエスト時にサーバへと送信される。

また、JavaScript からのアクセスを禁ずるために httponly フラグを付与して発行された Cookie は、http および https スキームの場合と同様に、ws および wss URI スキームにおいても送信される。

4.2.2. Web Storage

Web Storage あるいは DOM Storage は、JavaScript からアクセス可能なデータをクライアント側に保存するための仕組みである。データがセッション終了時に破棄される sessionStorage と、ブラウザ終了後も永続的にデータを保持し続ける localStorage がある。いずれの場合も、読み込みおよび書き込みは同一のオリジンに制限されている。このため、http のページ内から書き込まれた Web Storage の内容は https のページからアクセスできない。

Cookie では httponly フラグを与えることで JavaScript からの読み取りを防ぐことができるが、Web Storage には JavaScript からの読み取りを防ぐための仕組みがない。そのため、Web Storage に秘密情報を保存する場合には、Content Security Policy を併用するなどの対策を行う必要がある。

4.2.2.1. sessionStorage

sessionStorage はブラウザのウィンドウまたはタブが開かれている間データを保持する。sessionStorage の内容は、ページのリロードや他のページなどの表示などがあっても維持され、ブラウザのウィンドウまたはタブを閉じた時に廃棄される。また、他のウィンドウまたはタブの sessionStorage の内容は読み込みも書き込みもできない。sessionStorage は iframe または frame で表示されている同一オリジンの各コンテンツ間で共有される。

Internet Explorer 8 および 9 では、

- iframe または frame 間では sessionStorage が共有されない
- http および https で sessionStorage が分離されずに共有される(Internet Explorer 8 のみ)

という問題が既知となっている。

4.2.2.2. localStorage

localStorage は、ブラウザを終了させても、明示的に削除しない限り永続的にデータを保持し続ける。

ただし、Internet Explorer 8 および 9 には次の問題があることが知られている。

- メニューの「ファイル」「新しいセッション」から立ち上げた Internet Explorer とは localStorage が共有されないことがある
- http と https の間で sessionStorage が分離されずに共有される(Internet Explorer 8 のみ)

また、Safari では Mac 版、iPhone 版とも、プライベートブラウズ時には localStorage への書き込みが一切保存されない。例えば、次のコードをプライベートブラウズ機能が有効になった状態で実行してもデータは取得できない。

```
localStorage.setItem( "foo", "value" );
alert( localStorage.getItem( "foo" ) );
```

Web Storage に保存されたデータの有効期間は、Cookie にて管理される Web アプリケーションとしてのセッションとは一致しない。そのため、Web アプリケーションがログイン機構を持ち、ユーザに固有の情報を Web Storage に格納する場合、明示的にそれを削除しない限りログアウト後もアクセス可能である。例えば、ログイン後に、(a) localStorage にデータを保存したり、(b) localStorage に保存されているデータを使ったりする処理を行う Web アプリケーションがあった場合、ユーザ X がログイン後に(a)の処理で localStorage に保存されたデータを、同じ端末のブラウザを共用しているユーザ Y がログイン後に(b)の処理で参照する可能性がある。こうした問題を避けるため、ログアウト時に Web Storage に格納したデータを削除するなどして、別のユーザからアクセスされないようにしておくことが望ましい。

4.2.3. Offline Web Application

Offline Web Application 機能は、指定したリソースをオフラインキャッシュとしてデバイス上に保存しておくことで、ネットワークがオフラインになった後にも Web アプリケーションが利用できるようにするための機能である。

オフラインキャッシュを利用すると、長期にわたりユーザのデバイス上に保存されたリソースが使用されることになる。そのため、例えば暗号化されていない Wi-Fi のような信頼できないネットワークを利用している端末に対して、攻撃者が中間者攻撃により汚染したリソースをオフラインキャッシュとして保存させた場合、そのネットワークを離れた後も長く汚染されたリソースをユーザが使い続けることになり、結果として、例えば秘密情報が攻撃者のサイトに送信されるなどの被害の可能性も考えられる。そのため、秘密情報などの重要な情報を扱う場合は、当該リソースを持つサイト全体に対して HTTPS を利用することが望ましい。

4.2.4. Web Workers

Web Workers は JavaScript においてバックグラウンドで並列処理を行うための機能である。Web Workers を利用する場合の典型的な JavaScript コードを次に示す。

```
var worker = new Worker( 'worker.js' );
worker.onmessage = function( evt ){
    console.log( evt.data );
};
worker.postMessage( 'ping' );
```

worker.js は、バックグラウンドで動作する JavaScript である。典型的な例を次に示す。

```
onmessage = function( evt ){
    if( evt.data == "ping" ){
        postMessage( 'pong' );
    }
};
```

Web Workers において外部のスクリプトファイルを読み込む方法として、Worker コンストラクタ、SharedWorker コンストラクタ、importScripts メソッドがある。これらで読み込む URL を攻撃者が任意に指定できないように注意しなければならない。

```
// Worker コンストラクタの危険な例
var src = location.hash.substring(1);
var worker = new Worker( src );
```

上のコードが、http://example.jp/#worker.js の URL にアクセスすると、URL 中の#以降の「worker.js」が Web Workers のソースとして使用される。
現在のところ、Worker コンストラクタのソース URI は同一オリジンに限定されているが、将来的には、data スキームもサポートされる可能性がある。すでに Firefox および Opera では Worker コンストラクタでの data スキームの指定がサポートされている。そのため、攻撃者が次のような URI を指定してユーザにアクセスさせた場合、Web Workers として攻撃者の用意した任意のスクリプトが動作することとなる。

```
http://example.jp/#data:text/javascript,onmessage=...
```

SharedWorker コンストラクタも、現在は同一オリジンのリソースに限定されているが、将来的には data スキームをサポートする可能性があるため、Worker コンストラクタと同様、ソースとして任意の URI が指定されないような対策の必要がある。
importScripts は同一オリジンに限定されず、Web Workers において任意のオリジンのスクリプトを読み込むことができる。そのため、下記のようなコードでは、攻撃者の用意した任意のスクリプトが Web Workers として動作することとなる。

```
//脆弱なコード。importScripts に任意の URI がわたる
var src = location.hash.substring(1);
var worker = new Worker( 'worker.js' );
worker.postMessage( src );
```

```
// worker.js
onmessage = function( evt ){
    if( evt.data ) importScripts( evt.data );
};
```

ただし、Web Workers 内で攻撃者のスクリプトが動作しても、Web Workers 内からブラウザの DOM への直接的な操作ができないため、XSS 攻撃に比べて、一般的には脅威が低いと考えられる。

4.2.5. Cross Document Messaging

Cross Document Messaging(以下、XDM)は異なるオリジンのドキュメント間でメッセージをやり取りするための機能である。
XDM を使用する典型的なコードを次に示す。この例では、http://example.jp と http://example2.jp というオリジンの異なるドキュメントの間で通信を行っている。

```
<!-- サイト 1 -->
<!-- http://example.jp/xdm.html -->
<script type="text/javascript">
// メッセージ受信時のハンドラを登録
window.addEventListener( 'message', function( evt ){
    if( evt.origin == 'http://example2.jp' ){
        alert( 'got:' + evt.data );
    }
}, false );
...
//メッセージを送信
var iframe = document.getElementById( "child" ); // child iframe
iframe.contentWindow.postMessage( 'ping', "http://example2.jp" );
</script>
...
<!-- 通信対象-->
<iframe src="http://example2.jp/xdm-child.html" id="child"></iframe>
```

```
<!-- サイト 2 -->
<!-- http://example2.jp/xdm-child.html -->

<script type="text/javascript">
// メッセージ受信時のハンドラを登録
window.addEventListener( 'message', function( evt ){
    if( evt.origin == 'http://example.jp' ){
        evt.source.postMessage( "pong", evt.origin );
    }
}, false );
</script>
```

ドキュメントに対してメッセージを送信する際には、postMessage メソッドを使用する。postMessage メソッドの第 2 引数には対象ドキュメントのオリジンか、"*"を指定する。上のサイト 1 の例では、iframe で指定された URL が http://example2.jp/xdm-child.html なので、送信先オリジンとして http://example2.jp を postMessage メソッドの第 2 引数に指定している。このオリジンの指定が送信対象のドキュメントのオリジンと一致していない場合、対象ドキュメントへメッセージは送信されない。

```
<!-- サイト 1 からの抜粋 -->
<iframe src="http://example2.jp/xdm-child.html" id="child"></iframe>
...
var iframe = document.getElementById( "child" ); // child iframe
iframe.contentWindow.postMessage( 'ping', "http://example2.jp" );
```

対象ドキュメントのオリジンとして"*"を指定して postMessage メソッドを呼び出した場合、任意のオリジンのドキュメントがそのメッセージを受け取ることができる。意図しない通信相手もそのメッセージを受け取ることができるため、送信するメッセージに秘密情報が含まれる場合には、"*"ではなく対象ドキュメントのオリジンを指定する必要がある。

また、外部から送信されたメッセージを受け取るには、window オブジェクトの message イベントハンドラを登録する。message イベントハンドラには、引数としてイベントの詳細が Event オブジェクトとして渡される。

メッセージを受信した側は、引数の Event オブジェクトの origin プロパティを確認して、メッセージの送信元が想定している通信相手かどうかを確認し、任意のドキュメントからのメッセージを受け入れないよう注意する必要がある。

```
<!-- サイト 1 からの抜粋 -->
window.addEventListener( 'message', function( evt ){
    if( evt.origin == 'http://example2.jp' ){
        alert( 'got:' + evt.data );
    }
}, false );
```

4.3. XMLHttpRequest

XMLHttpRequest とは、JavaScript を使用してサーバと HTTP 通信を行うための API である。現在、主要なブラウザでは HTML5 とあわせ Cross-Origin Resource Sharing(CORS)に従ったクロスオリジンでの通信に対応した XHR Level 2 が実装されている。本節では、XHR Level 2 に関連するセキュリティ上の注意点について説明する。

なお、Internet Explorer 8、9 の XHR はクロスオリジンでの通信は対応しておらず、クロスオリジン通信のために XDomainRequest という類似の機能が提供されているが、XDomainRequest に関する詳細については本報告書では割愛する。

4.3.1. 意図しないクロスオリジンでのアクセス(クライアント側)

XHR が同一オリジンとの通信しかできないことを前提に書かれていた既存のコードが、XHR がクロスオリジンでのアクセスをサポートしたことにより脆弱となる場合がある。
例えば、自サイトとの通信を前提とし XHR の通信先を URL 内の#(ハッシュ)以降によって指定し、レスポンスの一部をそのままドキュメント内に挿入して表示する次のようなコードがあったとする。

```
// 同一オリジンでの通信を前提に書かれたコード
// http://example.jp/#/foo のような URL にて
// #以降を通信先 URL として使用する
var url = location.hash.substring(1);
var xhr = new XMLHttpRequest();
xhr.open( "GET", url, true );
xhr.onreadystatechange = function(){
    if( xhr.readyState == 4 && xhr.status == 200 ){
        div.innerHTML = xhr.responseText;
    }
};
xhr.send( null );
```

このようなコードは、XHR がクロスオリジンでのリクエストをサポートしていなかったときには問題なかったが、XHR によるクロスオリジン通信をサポートした現在のブラウザでは問題となる。攻撃者が http://example.jp/#//evil.example.com/のような URL へユーザを誘導した場合、XHR の通信先として example.jp 上のリソースではなく、攻撃者が用意した evil.example.com 上のリソースが使用され、XSS 攻撃や意図しないデータを表示させる攻撃が行われる可能性がある。上記の例では innerHTML を使用しているが、仮にこれがテキストノードへの代入やエスケープしたうえでの表示であったとしても、スクリプトは動作しないものの攻撃者が用意した偽コンテンツが表示される可能性がある。

対策としては、XHR が任意サイトのデータを読まないようリクエスト先を制限する必要がある。具体的には、次のようにリクエスト先を固定のリストで保持しておき、攻撃者の指定したリクエスト先が入り込まないようにしておくなどの方法がある。

```
// 安全な例。通信先を固定リストで保持
// http://example.jp/#1 のような URL にて通信先の index を指定する
var pages = [ "/", "/foo", "/bar", "/baz" ];
var index = location.hash.substring(1) | 0;

var xhr = new XMLHttpRequest();
xhr.open( "GET", pages[ index ] || '/', true );
xhr.onreadystatechange = function(){
    if( xhr.readyState == 4 && xhr.status == 200 ){
        div.innerHTML = xhr.responseText;
    }
};
xhr.send( null );
```

リクエスト先のドメインが想定された範囲のものかどうか JavaScript にて確認する方法では、そのサイト内にオープンリダイレクトがひとつでも存在すると、任意のドメインへとリクエストを転送できるため、攻撃者の用意したコンテンツを表示させられることとなる。

```
// 脆弱となる可能性のある例。
// サイト内にオープンリダイレクトが存在すると任意のサイトと通信可能
var url = url_for_request; // リクエスト先 URL
if( url.indexOf( "http://example2.jp/" ) == 0 ||
    url.indexOf( "http://example3.jp/" ) == 0 )
{
    // example2.jp または example3.jp のときのみ通信
```

```
    var xhr = new XMLHttpRequest();
    xhr.open( "GET", url, true );
    ....
}
```

このコードでは、XHR のリクエスト先を example2.jp または example3.jp だけに限定するようにしているが、example2.jp または example3.jp 上にオープンリダイレクトが存在している場合には、結果的に任意のサイトとの通信が可能となってしまう。XHR ではリダイレクトが発生したことを検知する手段や、最終的な URL を知る手段は存在しない。そのため、XHR のリクエスト先オリジンを確認するという方法では問題が発生する可能性があり、先に示したとおり事前にリクエスト先を固定のリストで保持しておくなどの方法で制限する必要がある。

4.3.2. 意図しないクロスオリジンでのアクセス(サーバ側)

XHR によるクロスオリジン通信におけるクライアント側の JavaScript を次に示す。

```
var xhr = new XMLHttpRequest();
xhr.open( "GET", "http://other.example.jp", true );
xhr.onreadystatechange = function(){
    if( xhr.readyState == 4 && xhr.status == 200 ){
        div.appendChild( document.createTextNode( xhr.responseText ) );
    }
};
xhr.send( null );
```

JavaScript のコード上は、同一オリジン内での通信の場合と同様である。
このときの HTTP リクエストおよびレスポンスは例えば次のようになる。

```
GET http://other.example.jp/ HTTP/1.1
Host: other.example.jp
Origin: http://example.jp
User-Agent: Mozilla/5.0(Windows NT 6.0; rv:18.0)
Connection: keep-alive

HTTP/1.1 200 OK
Date: Tue, 1 Jan 2013 09:00:00 GMT
Content-Length: 1512
Content-Type: text/plain; charset=utf-8
```

```
Access-Control-Allow-Origin: http://example.jp


...
```

リクエストには、どのページからリクエストが発行されたかのオリジンを示す Origin リクエストヘッダが付与される。サーバ側は、Access-Control-Allow-Origin レスポンスヘッダにてどのオリジンであればコンテンツにアクセス可能かを指定する。上の例では、http://example.jp をオリジンとする JavaScript であればコンテンツにアクセスすることが可能である。

サーバからの応答に Access-Control-Allow-Origin レスポンスヘッダが存在しない場合や Access-Control-Allow-Origin レスポンスヘッダに自身のオリジンが含まれていない場合は、XHR を利用した JavaScript 内で responseText プロパティなどを通じてコンテンツにアクセスすることはできない。

コンテンツに秘密情報が含まれ、特定のオリジンからのみ、そのコンテンツへのアクセスを許可するという場合は、Origin リクエストヘッダおよび Access-Control-Allow-Origin レスポンスヘッダだけでなく、従来どおりの Cookie を使用したアクセス制限を行う必要がある。Cookie を使用せずに Origin リクエストヘッダおよび Access-Control-Allow-Origin レスポンスヘッダのみを使用した場合、攻撃者はブラウザを使用せずに telnet クライアントなどのツールによって任意の Origin リクエストヘッダを付与したリクエストを送信可能であり、JavaScript を経由せずに直接コンテンツを読むことが可能だからである。

Access-Control-Allow-Origin: *という指定は、全てのオリジンからの JavaScript がコンテンツにアクセス可能なことを意味しているが、秘密情報を含むコンテンツに対してはこの指定をしてはいけない。攻撃者が罠ページから XHR によるリクエストを発行した場合、罠ページ内の JavaScript から秘密情報にアクセス可能となるからである。Access-Control-Allow-Origin: * という指定は、アクセス保護などを必要としない完全に公開状態のコンテンツにのみ使用し、それ以外については Access-Control-Allow-Origin: http://example.jp のように許可するオリジンを指定する。

XHR を用いてクロスオリジンのリクエストを発行する際、デフォルトでは Cookie は送信されない。クロスオリジンで Cookie を送信するには、下記の例のように withCredentials プロパティを true に設定する必要がある。Cookie を使用した場合 Access-Control-Allow-Origin: * という指定をすることは仕様上禁止されており、XHR からも読み込むことができない。

```
var xhr = new XMLhttpRequest();
xhr.open( "GET", "http://other.example.jp", true );
xhr.withCredentials = true; // Cookie が送信されるようになる
xhr.onreadystatechange = function(){
    if( xhr.readyState == 4 && xhr.status == 200 ){
        div.appendChild( document.createTextNode( xhr.responseText ) );
```

```
    }
};
xhr.send( null );
```

```
GET http://other.example.jp/ HTTP/1.1
Host: other.example.jp
Origin: http://example.jp
User-Agent: Mozilla/5.0(Windows NT 6.0; rv:18.0)
Cookie: session=12AFE9BD34E5A202
Connection: keep-alive

HTTP/1.1 200 OK
Date: Tue, 1 Jan 2013 09:00:00 GMT
Content-Length: 1512
Content-Type: text/plain; charset=utf-8
Access-Control-Allow-Origin: http://example.jp

...
```

また、これら以外にも、XHR を経由してクロスオリジンで CSRF を発生させることも可能になっているので、従来通りトークンなどを利用しての CSRF 対策が必要である。詳細については、3.2.「**クロスサイト・リクエスト・フォージェリ**」を参照していただきたい。

4.3.3. Ajax データによる XSS

XHR でやり取りされるデータ(Ajax データ)に HTML として解釈可能な文字列が含まれる場合、これをブラウザで直接開くと XSS 攻撃が成立する可能性がある。

```
Date: Tue, 1 Jan 2013 09:00:00 GMT
Content-Length: 86
Content-Type: application/json; charset=utf-8

{
    "name" : "foo",
    "value" : "<html><script>alert(1);</script></html>"
}
```

上記のような JSON 形式のデータを、Internet Explorer で直接開くと、データの先頭部分に含まれる文字列

を調べて HTML コンテンツであると判断して解釈しようとする結果 JavaScript が動作する可能性がある。JSON 以外のデータであっても、text/plain や text/csv など様々な種類の Ajax データで XSS 攻撃が行われる可能性がある。

また、XSS 攻撃を使用しなくても、攻撃者が罠ページを用意し、攻撃対象となる Ajax データを<script>要素のソースとして読み込ませるなどの手法により、秘密情報を含む Ajax データを攻撃者が盗み見る可能性がある。この手法では、JavaScript として解釈可能な JSON や CSV などが攻撃対象として狙われやすい。

```
攻撃者の罠サイトでは、秘密情報を含む Ajax データを JavaScript として読み込む
<script src="http://example.jp/target.json"></script>
<script src="http://example.jp/target.csv"></script>
```

これらの問題に対して、4.3.3.1.「X-Content-Type-Options レスポンスヘッダを指定する」、4.3.3.2.「XHR からのリクエストにのみ対応する」で述べる両方の対策を実施しておくことを推奨する。これは、Internet Explorer 6 および 7 は、X-Content-Type-Options レスポンスヘッダに対応していないためである。

4.3.3.1. X-Content-Type-Options レスポンスヘッダを指定する

Internet Explorer 8 以降においては、レスポンスヘッダで X-Content-Type-Options: nosniff を指定することにより、与えられたデータを分析して Content-Type をブラウザが判断する動作を抑制できるので、HTML として扱って欲しくないコンテンツが HTML として処理されることによって生じる XSS 攻撃を防ぐことができる。詳細については 5.2.「**X-Content-Type-Options**」を参照していただきたい。

4.3.3.2. XHR からのリクエストにのみ対応する

XHR からのリクエストにのみ Ajax データを応答するために、クライアント側から XHR のリクエストを送る際のリクエストヘッダ内に特定文字列を含めておき、サーバ側ではその文字列を含むリクエスト以外はエラーなどを返すことにより、ブラウザから直接 Ajax データを参照できないようにすることが可能である。
リクエスト時には、次のように setRequestHeader メソッドによりカスタムヘッダを付与する。

```
var XHR = new XMLHttpRequest();
XHR.open( "GET", "http://example.jp/foo.json", true );
XHR.onreadystatechange = function(){ ... };
XHR.setRequestHeader( "X-Request-With", "XMLHttpRequest" );
XHR.send( null );
```

```
GET http://example.jp/foo.json HTTP/1.1
Host: example.jp
```

```
X-Request-With: XMLHttpRequest
User-Agent: Mozilla/5.0(Windows NT 6.0; rv:18.0)
Cookie: session=12AFE9BD34E5A202
Connection: keep-alive
```

サーバ側では、X-Request-With リクエストヘッダなどの独自のリクエストヘッダの存在および内容を確認することにより、ブラウザで直接コンテンツを開いたのではなく、XHR から発行されたリクエストであることが確認でき、XHR からの場合には正規の応答を返し、そうでない場合にはステータスコード 403 などを返すなどの対応をとることができる。なお、jQuery や prototype.js のように、上記のようなリクエストヘッダを自動的に付与してくれるライブラリも存在する。

クロスオリジンでの通信において setRequestHeader メソッドを使用する場合は、GET や POST に先立ち、preflight リクエストと呼ばれる OPTIONS メソッドのリクエストが発行される。このとき、本来のリクエストで使用するメソッドが Access-Control-Request-Method リクエストヘッダで送信され、setRequestHeader メソッドで設定されたカスタムヘッダが Access-Control-Request-Headers リクエストヘッダで送信される。preflight リクエストの処理は XHR が自動的に行うため、クライアント側では JavaScript で preflight リクエストに関する処理を実装する必要はないが、サーバ側での対応が必要となる可能性がある。

```
// クロスオリジンで setRequestHeader する例
var XHR = new XMLHttpRequest();
XHR.open( "GET", "http://other.example.jp/foo.json", true );
XHR.withCredentials = true; // Cookie を送信
XHR.onreadystatechange = function(){ ... };
XHR.setRequestHeader( "X-Request-With", "XMLHttpRequest" );
XHR.send( null );
```

次の例では、foo.json は HTML として解釈可能な文字列を含んでいるが、XHR からのリクエストにのみ対応することで、ブラウザから直接 Ajax データを参照できないようにしている。

```
OPTIONS http://other.example.jp/foo.json HTTP/1.1
Host: other.example.jp
Origin: http://example.jp
User-Agent: Mozilla/5.0(Windows NT 6.0; rv:18.0)
Cookie: session=12AFE9BD34E5A202
Connection: keep-alive
Access-Control-Request-Method: GET
Access-Control-Request-Headers: X-Request-With

HTTP/1.1 200 OK
Date: Tue, 1 Jan 2013 09:00:00 GMT
Access-Control-Allow-Origin: http://example.jp
Access-Control-Allow-Methods: GET, OPTIONS
Access-Control-Allow-Headers: X-Request-With
Access-Control-Max-Age: 1728000
Content-Length: 0
Connection: Keep-Alive
Content-Type: application/json

GET http://other.example.jp/foo.json HTTP/1.1
Host: other.example.jp
X-Request-With: XMLHttpRequest
Origin: http://example.jp
User-Agent: Mozilla/5.0(Windows NT 6.0; rv:18.0)
Cookie: session=12AFE9BD34E5A202
Connection: keep-alive

HTTP/1.1 200 OK
Date: Tue, 1 Jan 2013 09:00:00 GMT
Access-Control-Allow-Origin: http://example.jp
Connection: Keep-Alive
Content-Length: 86
Content-Type: application/json; charset=utf-8

{
    "name" : "foo",
    "value" : "<html><script>alert(1);</script></html>"
}
```

5. HTML5 におけるセキュリティ機能

本章では、次に挙げる一部のブラウザに実装されているユーザを保護するためのセキュリティ機能の概要と、それを有効に活用するための留意事項を述べる。これらの機能を有効にするかどうかは、HTTP のレスポンスヘッダで指定することができる。これらの機能は、まだ標準化されていないものが大部分で、機能の有無を含めてブラウザによる相違があることに注意されたい。

- X- XSS-Protection
- X-Content-Type-Options
- X-Frame-Options
- Content-Security-Policy
- Content-Disposition
- Strict-Transport-Security

5.1. X-XSS-Protection

ブラウザによっては XSS 攻撃から保護する機能が付与されていることがある。この機能の名前は次のようにブラウザにより異なっている。

・Internet Explorer 8 以降 ：XSS フィルター
・Google Chrome : XSS Auditor
・Safari : XSS Auditor

ブラウザにより差はあるが、リクエストに含まれる<script>などの要素がレスポンスにも同様に含まれていた場合に XSS 攻撃と見なして、それらをブロック・無害化することにより反射型 XSS 攻撃を防ぐというのが共通する原理である。原理上、リクエストとレスポンスの両方に類似する文字列が含まれる場合に XSS 攻撃を誤検出することがある。

Internet Explorer の XSS フィルターでは、図 5-1 に示すように、XSS 攻撃を検知した場合にアドレスバーの下に通知が表示されるが、Google Chrome および Safari では特にそのような通知は表示されない。

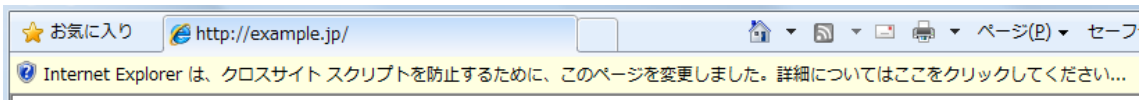

図 5-1 Internet Explorer における XSS 保護フィルタ動作画面

この機能は、表 5-1 に設定例を示したような形式で、X-XSS-Protection レスポンスヘッダにより制御することができる。

表 5-1 X-XSS-Protection の設定例

X-XSS-Protection: 0	XSS 保護フィルタ機能を無効にする。
X-XSS-Protection: 1	XSS 保護フィルタ機能を有効にする。
X-XSS-Protection: 1;mode=block	XSS 保護フィルタ機能を有効にする。XSS 攻撃検出時にブラウザでの表示を空白にする。

「0」を指定した場合、ブラウザの XSS 保護フィルタは一時的に無効となる。「1」を指定した場合、ブラウザの XSS 保護フィルターは有効となる。「1;mode=block」を指定した場合、XSS 攻撃検出時に検出された要素だけを削除するのではなく、空白のドキュメントが表示される(Internet Explorer では「#」のみが表示され、Google Chrome、Safari では about:blank ページが表示される)。X-XSS-Protection が指定されていない場合は、「1」を指定した場合と同様に XSS 保護フィルタが有効となる。

X-XSS-Protection ヘッダはレスポンスヘッダ内で指定された場合のみ有効で、<meta http-equiv>により HTML 内で指定しても機能しない。

X-XSS-Protection: 0 を指定して XSS 保護フィルタを無効にすることは、誤検出が多いなど特別な理由があるページのみに限定すべきである。

5.2. X-Content-Type-Options

レスポンスヘッダにて X-Content-Type-Options: nosniff を指定することにより、Internet Explorer 8 以降では Content-Type レスポンスヘッダに従ってコンテンツを取り扱う。

```
Date: Tue, 1 Jan 2013 09:00:00 GMT
Content-Length: 42
X-Content-Type-Options: nosniff
Content-Type: text/plain; charset=utf-8

<html>
<script>alert(1)</script>
</html>
```

Internet Explorer 6 および 7 でこのようなコンテンツを開いた場合、Content-Type は text/plain であるにも関わらず、MIME スニッフィングにより HTML と判断され、JavaScript が動作してしまうが、Internet Explorer 8 以降では X-Content-Type-Options: nosniff が指定されている場合には Content-Type の指定どおりにコンテンツを text/plain として取り扱う。

X-Content-Type-Options を使用することにより、Internet Explorer 8 以降に対しては、4.3.3.「**Ajax データ**

による XSS」で説明したような、HTML 以外のものが HTML として処理されることによる XSS 攻撃を防ぐことができる。一方、Internet Explorer 6 および 7 では、X-Content-Type-Options レスポンスヘッダは利用できないため、XHR で応答するデータに関しては、4.3.3.2. 「XHR からのリクエストにのみ対応する」で解説した対策が必要である。また、XHR 以外からもアクセスが想定されるコンテンツに関しては

- HTML として解釈されても問題が出ないようにエスケープする
- Content-Disposition: attachment レスポンスヘッダを付与しブラウザ内で直接開かせない
- 異なるドメインにコンテンツを配置する

などの対策で影響を軽減することができる。

また、X-Content-Type-Options: nosniff は Internet Explorer 9 および 10 に対しては、コンテンツを直接開く場合だけでなくスタイルシートや<script>要素での外部スクリプトの読み込みにも適用され、次の Content-Type のみを使用することができる。

・スタイルシート

text/css

・スクリプト

application/ecmascript	text/ecmascript	text/x-javascript
application/javascript	text/javascript	text/vbs
application/x-javascript	text/jscript	text/vbscript

なお、X-Content-Type-Options はレスポンスヘッダ内で指定された場合のみ有効で、<meta http-equiv> により HTML 内で指定された場合には機能しない。

HTTP レスポンスヘッダにて X-Content-Type-Options: nosniff を指定することで、読み込み可能なスタイルシートやスクリプトソースの Content-Type を制限することができるため、動的に生成される全てのコンテンツに対して X-Content-Type-Options: nosniff を付与することを推奨する。

5.3. X-Frame-Options

Internet Explorer 6 および 7 を除く現在の主要なブラウザでは、クリックジャッキング対策としてレスポンスヘッダにて X-Frame-Options を指定することにより、iframe、frame 内での該当コンテンツの埋め込みを禁止することができる。

例えば、次のような HTTP レスポンスヘッダを付与することにより、コンテンツが他ページ内の iframe などで表示されることを禁止できる。

```
Date: Tue, 1 Jan 2013 09:00:00 GMT
Content-Length: 28400
X-Frame-Options: DENY
Content-Type: text/html; charset=utf-8

<!doctype html>
<html>....</html>
```

X-Frame-Options として指定可能な値は次の 3 種類である。

表 5-2: X-Frame-Options で使用される値

X-Frame-Options: DENY	Iframe などでの表示を禁止する
X-Frame-Options: SAMEORIGIN	ページ自身のオリジンと同一オリジンの場合のみ iframe などでの表示が可能。オリジンが一致しない場合は表示を禁止する。
X-Frame-Options: ALLOW-FROM uri	ページ自身のオリジンが uri で指定されたオリジンと一致する場合のみ iframe などでの表示が可能。一致しない場合は表示を禁止する。

「ALLOW-FROM uri」の指定は Internet Explorer 8 以降および PC 版、Android 版の Firefox(ESR を除く)でサポートされている。uri 部にはプロトコルスキームも必要である。次は、http://example.jp/内のページから iframe で読み込まれた場合にのみ表示を許可する場合の例である。

```
X-Frame-Options: ALLOW-FROM http://example.jp
```

また、X-Frame-Options はいくつかのブラウザではレスポンスヘッダ内で指定された場合のみ機能し、<meta http-equiv>で指定された場合には機能しない。

クリックジャッキング対策として、X-Frame-Options レスポンスヘッダではなく「フレームバスティング」と呼ばれるコードによってコンテンツの iframe 内への埋め込みを防ぐ方法が知られているが、この方法は多数の欠点があるため、使用するべきではない。

```
// フレームバスティングコードの一例
if( top != self ){
    top.location = self.location;
}
```

5.4. Content-Security-Policy

Content Security Policy(CSP)は、読み込み可能なリソースのオリジンをレスポンスヘッダにて指定することで、機能を制限し、XSS 攻撃の可能性を低減する機能である。現在 CSP に対応しているブラウザは、Firefox および Google Chrome、Safari である。

CSP が有効になると、ブラウザ内の次の機能が制限される。

- <script>や<img>、<iframe>といった各要素でのリソースの読込み
- や<script>alert(1)</script>といったインラインでの JavaScript の実行
- JavaScript 内での eval 関数や Function コンストラクタといった、文字列からのコードの生成
- javascript スキームや data スキーム

CSP は Firefox では X-Content-Security-Policy レスポンスヘッダで、Google Chrome では Content-Security-Policy レスポンスヘッダまたは X-WebKit-CSP レスポンスヘッダ、Safari では X-WebKit-CSP レスポンスヘッダで指定する。将来的には、Content-Security-Policy レスポンスヘッダに統一される見込みである。すべてに対応するためには、次のように 3 種類のレスポンスヘッダを出力する。

```
Date: Tue, 1 Jan 2013 09:00:00 GMT
Content-Type: text/html; charset=utf-8
X-Content-Security-Policy: default-src 'self'; img-src img.example.jp
X-WebKit-CSP: default-src 'self'; img-src img.example.jp
Content-Security-Policy: default-src 'self'; img-src img.example.jp
```

この例では、画像は img.example.jp 上のものを、それ以外の全てのリソースについてドキュメントと同一オリジンからのみ読み込みを許可している。CSP では、この例の default-src や img-src をディレクティブ、'self' や img.example.jp をディレクティブソースと呼んでおり、ディレクティブ、ディレクティブソースとして W3C では様々な種類が定義されているが、ブラウザによって実装状況は異なる。ディレクティブとして代表的なものを次に示す。

表 5-3: ディレクティブの例

default-src	他のディレクティブで指定されていない、デフォルトで許可されるディレクティブソースを列挙する
script-src	スクリプトとして許可するディレクティブソースを列挙する
style-src	スタイルシートとして許可するディレクティブソースを列挙する
img-src	画像の読込みを許可するディレクティブソースを列挙する
frame-src	フレームとして表示可能なディレクティブソースを列挙する

ディレクティブソースとしては、先に示した「img.example.jp」のようなホスト名での指定の他に、「*.example.jp」といったワイルドカードや「https:」のようなスキーム名、次に示す予約語での指定が可能で

ある。

表 5-4: ディレクティブソースの例

'self'	ドキュメント自身と同一オリジンの場合にのみ許可する
'none'	どのオリジンも許可しない
'unsafe-inline'	script-src、style-src においてインラインでのスクリプト記述、スタイル記述を許可する
'unsafe-eval'	JavaScript 内での eval、Function、setTimeout、setInterval といった文字列からコードを生成する機能を許可する

CSP では、ポリシー違反を検出した場合にブラウザから自動的にサーバに対してレポートを送信することもできる。レポートの送信先は report-uri ディレクティブによって指定する(Content-Security-Policy 以外のレスポンスヘッダは省略している)。

```
Date: Tue, 1 Jan 2013 09:00:00 GMT
Content-Type: text/html; charset=utf-8
Content-Security-Policy: default-src 'self'; img-src img.example.jp;
    report-uri http://example.jp/report.cgi
```

ポリシー違反のレポートを受け取ることで、サイト運営者は XSS 攻撃を早い段階で察知することが可能となる。

また、レスポンスヘッダにて Content-Security-Policy の代わりに Content-Security-Policy-Report- Only(または X-WebKit-CSP-Report-Only あるいは X-Content-Security-Policy-Report-Only)を指定した場合、ポリシーに違反した場合でもリソースはブロックされることなく読み込まれ、違反のレポートのみが送信されることになる。

```
Date: Tue, 1 Jan 2013 09:00:00 GMT
Content-Type: text/html; charset=utf-8
Content-Security-Policy-Report-Only: default-src 'self';
     img-src img.example.jp; report-uri http://example.jp/report.cgi
```

CSP は現在も活発に仕様策定と実装が進められており、ブラウザ間およびバージョン間での実装差異が大きいという段階である。

5.5. Content-Disposition

レスポンスヘッダにおいて、Content-Disposition: attachment を指定することでコンテンツをブラウザ内で表示するのではなく、ファイルとして保存するように指定できる。この機能は Web メールや掲示板などにおいて添付ファイル機能として広く利用されている。Internet Explorer ではこのときに表示されるダイアログにおいて「開く」を選択することで XSS 攻撃が行われる可能性がある。

```
Date: Tue, 1 Jan 2013 09:00:00 GMT
Content-Length: 42
Content-Disposition: attachment; filename="index.html"
Content-Type: text/html; charset=utf-8

<html>
<script>alert(1)</script>
</html>
```

例えば、Internet Explorer 8 でこのようなレスポンスを返すサーバにアクセスした場合、次のようなダイアログが表示される。

図 5-2: ダウンロード確認ダイアログ

ダイアログが表示された時点でユーザが「開く」を選択すると、サーバをオリジンとしてコンテンツがブラウザ上で表示されるため蓄積型の XSS 攻撃が行われる可能性がある。Internet Explorer 8 以降では、レスポンスヘッダに X-Download-Options: noopen を指定することでダイアログの「開く」ボタンを非表示にすることができる。

```
Content-Type: text/html; charset=utf-8
Date: Tue, 1 Jan 2013 09:00:00 GMT
Content-Length: 42
Content-Disposition: attachment; filename="index.html"
X-Download-Options: noopen

<html>...
<script>alert(1)</script>
</html>
```

図 5-3: ダウンロード確認ダイアログ

5.6. Strict-Transport-Security

HTTP Strict Transport Security (以下、HSTS)と呼ばれる機能を利用すると、一定の期間のアクセスを HTTPS 経由に強制できる。

この機能は、HTTPS の応答において次のような Strict-Transport-Security レスポンスヘッダを追加すると有効になる。このヘッダは HTTP の応答の中に追加しても無視されるだけである。

```
Strict-Transport-Security:max-age=1800
```

max-age で示される値には、HSTS を有効にする期間を秒単位で指定する。上の例では、ブラウザは応答を受け取ってから 30 分間は Strict-Transport-Security レスポンスヘッダが送られたサイトへ HTTPS でアクセスする。

max-age で示された期間内に再度 Strict-Transport-Security レスポンスヘッダを受け取った場合には、HSTS の有効期間は新しい値に更新される。

オプションとして includeSubdomains パラメータを指定した場合には、サブドメインも含め HSTS が有効となる。includeSubdomains を指定しない場合は、サブドメインを含まない。

```
Strict-Transport-Security:max-age=1800; includeSubdomains
```

現在、HSTS をサポートしているブラウザは Firefox と Google Chrome、Opera である。

参考 ：Do not Track

本章では、開発者がサーバ側に設定・実装などを行うことにより、クライアント側のブラウザに処理を要求するものを扱ったが、ユーザがクライアントに設定することにより、リクエストヘッダが付加され、サーバ側に処理を要求するものもある。

Do Not Track(以下 DNT)は、「ユーザの行動履歴の追跡をしないで欲しい」とのユーザの意思を Web サイトに対して表明するために使用されるリクエストヘッダである。Firefox、Google Chrome(PC 版のみ)、Safari、Opera、Internet Explorer 9 以降の各ブラウザでは、ユーザが明示的に設定を変更することで機能が有効になり(Internet Explorer 10 では簡単設定を選択してインストールされた場合はデフォルトで有効)、リクエストヘッダとして DNT: 1 が各リクエストに含まれるようになる。

アプリケーションが DNT: 1 というリクエストヘッダを受け取った場合の具体的な対応方法はサービス提供側に任されているが、サービスを提供する組織やサービスの性質に応じてユーザのプライバシに配慮した実装とするべきである。一般社団法人 Mozilla Japan「Do Not Track **実装ガイド**」にはサービス提供側のとるべき方針として複数のケーススタディが掲載されている。

6. まとめ

本調査レポートでは、HTML5 の Web アプリケーションを開発する際に、セキュリティ上留意すべき機構に関して調査を行い、分散しているセキュリティ情報について体系的にまとめている。本調査レポートを開発の際に参照していただければ幸いである。

今後は、HTML5 の仕様の変化などに対して見直し、必要に応じて修正や追加を行う予定である。

参考文献

- HTML5
  http://www.w3.org/TR/html5/
  (最終アクセス 2013 年 10 月 30 日)
- HTML Standard
  http://www.whatwg.org/specs/web-apps/current-work/multipage/
  (最終アクセス 2013 年 10 月 30 日)
- 「安全なウェブサイトの作り方」改訂第 6 版
  https://www.ipa.go.jp/security/vuln/websecurity.html
  (最終アクセス 2013 年 10 月 30 日)
- 情報処理推進機構：IPA テクニカルウォッチ：『DOM Based XSS』に関するレポート
  https://www.ipa.go.jp/about/technicalwatch/20130129.html
  (最終アクセス 2013 年 10 月 30 日)
- Cross-Origin Resource Sharing
  http://www.w3.org/TR/cors/
  (最終アクセス 2013 年 10 月 30 日)
- RFC 6454 - The Web Origin Concept
  https://tools.ietf.org/html/rfc6454
  (最終アクセス 2013 年 10 月 30 日)
- XMLHttpRequest
  http://www.w3.org/TR/XMLHttpRequest/
  (最終アクセス 2013 年 10 月 30 日)
- The WebSocket API
  http://www.w3.org/TR/websockets/
  (最終アクセス 2013 年 10 月 30 日)
- Web Workers
  http://www.w3.org/TR/workers/
  (最終アクセス 2013 年 10 月 30 日)
- HTTP Header X-Frame-Options
  http://tools.ietf.org/html/draft-ietf-websec-x-frame-options
  (最終アクセス 2013 年 10 月 30 日)
- MIME-Handling Change: X-Content-Type-Options: nosniff (Windows)
  http://msdn.microsoft.com/en-us/library/ie/gg622941(v=vs.85).aspx
  (最終アクセス 2013 年 10 月 30 日)
- Controlling the XSS Filter - IEInternals - Site Home - MSDN Blogs
  http://blogs.msdn.com/b/ieinternals/archive/2011/01/31/controlling-the-internet-explorer-xss-filter-with-the-x-xss-protection-http-header.aspx
  (最終アクセス 2013 年 10 月 30 日)

- Tracking Preference Expression (DNT)
  http://www.w3.org/TR/tracking-dnt/
  (最終アクセス 2013 年 10 月 30 日)
- HTTP Strict Transport Security (HSTS)
  https://tools.ietf.org/html/draft-ietf-websec-strict-transport-sec
  (最終アクセス 2013 年 10 月 30 日)
- Content Security Policy 1.0
  http://www.w3.org/TR/CSP/
  (最終アクセス 2013 年 10 月 30 日)
- HTML5 Security Cheatsheet
  http://html5sec.org/
  (最終アクセス 2013 年 10 月 30 日)
- HTTP access control (CORS) - HTTP | MDN
  https://developer.mozilla.org/en-US/docs/HTTP/Access_control_CORS
  (最終アクセス 2013 年 10 月 30 日)
- Do Not Track 実装ガイド
  http://www.mozilla.jp/static/docs/firefox/dnt-guide.pdf
  (最終アクセス 2013 年 10 月 30 日)
- HTML5 Implementation Issues in IE8 (and IE9) - IEInternals - Site Home - MSDN Blogs
  http://blogs.msdn.com/b/ieinternals/archive/2009/09/16/bugs-in-ie8-support-for-html5-postmessage-sessionstorage-and-localstorage.aspx
  (最終アクセス 2013 年 10 月 30 日)

JPCERT CC®

付録

動作確認ブラウザ

本書における対象ブラウザ、および動作を確認しているブラウザは原則として次の通りである。次の対象から外れるものについてはバージョンを都度明記している。パッチについては 2013 年 2 月時点での最新のものを全て適用している。

動作確認ブラウザ(PC 用)

- Internet Explorer 6～10
- Google Chrome バージョン 25.0.1364.97 m (リリース版)
- Mozilla Firefox 18.0.2 (release channel)
- Opera 12.14
- Safari 6.0.2 (Mac 版)

動作確認ブラウザ(スマートフォン用)

- Mozilla Firefox for Android 19.0
- Android 2.3 標準ブラウザ
- Opera Mobile 12.10.ADR
- Safari 5.1 for iOS 5.1.1