解説

Web 2.0 アプリケーションにおける代表的な攻撃手法とその対策

吉濱 佐知子　石田 愛　浦本 直彦
（日本アイ・ビー・エム（株）東京基礎研究所）

Web 2.0 というトレンドの中で，Ajax や Mashup などの技術やユーザ生成コンテンツの増加により，新しい攻撃手法が日々登場する状況となっている．特に Web アプリケーション層の脆弱性は深刻で，90%の Web サイトは何らかのアプリケーションレベルの脆弱性を持っているといわれている．本稿では Web 2.0 の技術的特徴と，それを利用した攻撃手法，代表的な対策手法と今後の展望について解説する．

Web2.0 におけるセキュリティ問題とは

Ajax[☆1] やマッシュアップ（Mashup）に代表される Web 2.0 技術は，単なるトレンドで終わることなく，Web における基盤技術の一角として認知され，企業による使用も始まっている．Web 2.0 技術を用いたビジネスアプリケーションの構築を考えると，セキュリティの問題は避けて通ることができない．Ajax は，非同期通信の利用により，ページ全体を書き換えることなく，効率の良いページの更新と動的で使いやすいインタフェースを実現した．しかし，これによって，従来のように Web サーバが攻撃対象となるだけでなく，リッチになったクライアント側アプリケーションの機能を悪用した新たな種類の攻撃が増加している．

また，Web 2.0 では SNS やブログなどのユーザ参加型のアプリケーションに特徴があるため，悪意を持ったユーザが攻撃者として入力データ中に JavaScript コードを埋め込むクロスサイトスクリプティング（Cross-Site Scripting, XSS）攻撃の問題が深刻さを増している．注入されたコードは，SNS プロバイダなどの信頼されたサーバからダウンロードされるため，従来のブラウザのセキュリティモデルではうまく扱うことができない．

さらに，複数のサービスやウィジットを組み合わせるマッシュアップでは，信用できるサービスと必ずしも信用できない別のサービスを組み合わせる場合があり得るため，悪意を持ったサービスから，他のサービスに対する攻撃が可能となる．現行のブラウザでは Same-Origin Policy というセキュリティポリシーが実装されているが，そもそも異なるドメインのサービスを組み合わせることに妙があるマッシュアップでは，その仮定に基づいたセキュリティモデルが必要である．

もちろんこれらの問題は，従来の Web アプリケーションでもある程度生じていた問題であるが，特に最近の Web 2.0 という技術とビジネスモデルがもたらした変化によって顕在化しつつある．本稿では，Web 2.0 の技術的特徴，Web 2.0 の普及とともに深刻化したセキュリティ上の問題，およびそれらを防ぐための技術について紹介する．

Web 2.0 の技術的特徴

Web 2.0 と従来型の Web の技術的な違いの 1 つは，Ajax と呼ばれるプログラミングモデルで特徴づけられる．Ajax とは Asynchronous JavaScript と XML を組み合わせた造語であり，ブラウザ上で動作するクライアント側 Web アプリケーションにおいて，JavaScript というスクリプト言語を用いて動的に処理を行う点に特徴がある．

図 -1 に従来型 Web と Ajax の比較を示す．従来型の Web アプリケーションでは，ある URL にアクセスしてダウンロードされた HTML 文書が，ほぼそのままの形でブラウザ上に表示される．HTML がフォームを含む場

☆1 Garrett, J. J. : Ajax : A New Approach to Web Applications (Feb. 18, 2005), http://www.adaptivepath.com/publications/essays/archives/000385.php

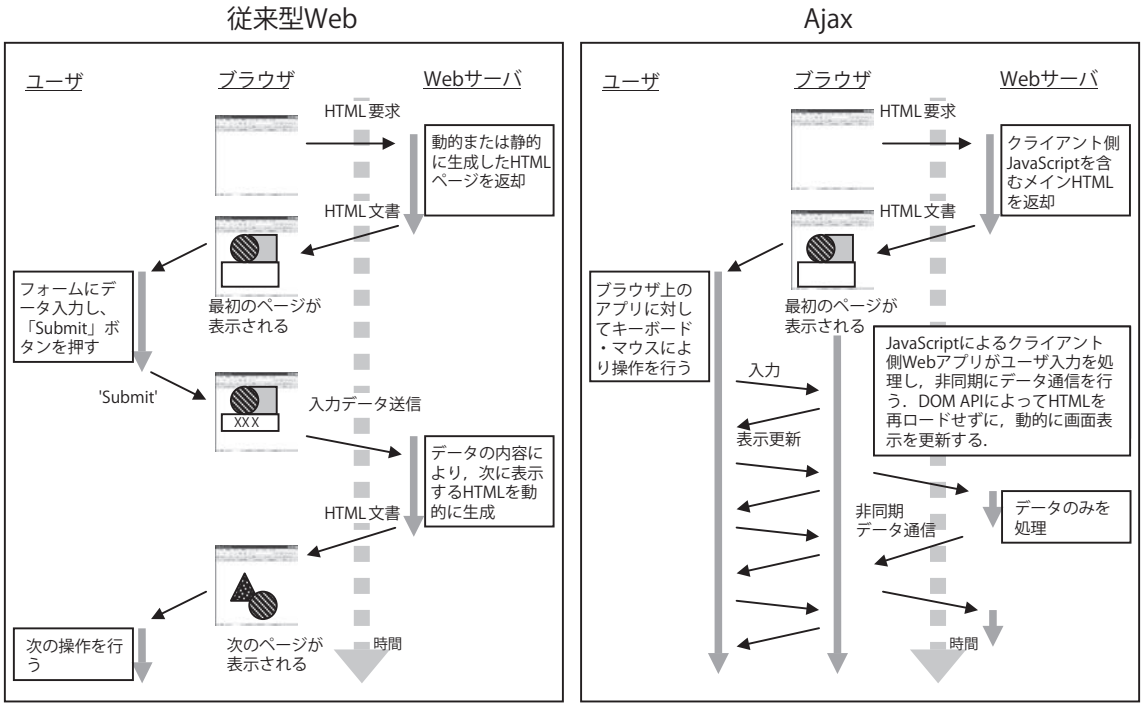

図 -1　従来型 Web と Ajax の比較

合，ユーザがデータを入力して Submit ボタンを押した時点で入力データがサーバ側に送られ，その応答として次の画面を表す HTML 文書がダウンロードされる．このように，ユーザ・Web ブラウザのインタラクションとブラウザ・サーバ間の通信が同期して行われる．

一方で典型的な Ajax アプリケーションでは，いったんメインの HTML がダウンロードされた後は XMLHttpRequest などのブラウザ組込み API を用いて，XML や後述する JSON などの形式で必要なデータのみをダウンロードする．現行の Web ブラウザの多くは，表示している Web ページの HTML 文書を木構造で表現した DOM（Document Object Model）という内部データモデルで保持している．ブラウザの提供する API を使用して JavaScript から DOM を書き換えることで，ブラウザ上に表示されているデータを更新することができる．また，ブラウザ上で発生するキーボードやマウスイベントは，DOM の提供するイベント機構により，クライアント側 JavaScript が処理する．これらの仕組みによって，ユーザ＝ Web ブラウザのインタラクションとブラウザ＝サーバ間の通信を非同期に行うことができ，従来のデスクトップアプリケーションと遜色ない使用感を Web アプリケーション上で実現することが可能となる .

Web 2.0 の技術的特徴の 2 つ目は，既存のサービスやコンテンツを組み合わせて新しいサービスを作り出す「マッシュアップ」と呼ばれる技術である．たとえば，もともと別の Web サーバで提供されているレストランのリストと地図サービス，それにレストランを訪問した人の感想などのコンテンツを統合して，地図のついたレストランガイドサービスを作る，などの例がある．マッシュアップの詳細については後述する．

Web 2.0 における攻撃の特徴

Web 1.0（従来型 Web）と Ajax に代表される Web 2.0 には，以下のような相違点がある（厳密には Web 1.0 と Web 2.0 にはっきりした技術的境界線があるわけではないが，便宜上以下のように分類する）.

- Web 1.0 ではサーバ側で生成された HTML が，ほぼそのままの形でブラウザ上に表示され，JavaScript は補助的に使用された．Web 2.0 ではクライアント側の JavaScript がリッチになり，複雑な処理を行う．
- Web 1.0 では HTTP は主に HTML ページやイメージなどの可視情報を転送するために使用されたが，Web 2.0 では，クライアントからサーバに対してコマンドを投入したり，データをやりとりしたりというユーザにとって不可視の情報のやりとりにも使われる．

そのため，可能な攻撃のタイプにも違いが出てきている．Web 1.0 の世界では，守るべきリソースはサーバ側にあり，サーバへの不正アクセスを防止するのが Web セキュリティの前提であった．そのために，ユーザ認証を行い，セッション管理を正しく行うことがセキュリテ

ィ上重要であった．

一方で，Web 2.0 の世界では，守るべきリソースはサーバ側だけでなく，クライアント側にも多く存在する．たとえば一般的な XSS の場合，Web 1.0 では XSS によりセッション情報（cookie）を盗み出し，セッションをハイジャックするのが代表的な攻撃であったが，Web 2.0 の場合，セッション情報を盗み出すだけでなく，DOM イベントをハンドルすることにより，ユーザのキー入力を直接盗むということが可能になる．この場合はユーザのキー入力そのものが盗む対象であり，サーバ側のリソースへの不正アクセスを必要としない．

またセッション管理という観点では，Web 1.0 ではセッション情報を推測したり盗んだりすることにより，セッションをハイジャックするのが代表的な攻撃であった．一方，Web 2.0 のセッション管理に対する攻撃である CSRF では，攻撃者は直接セッション情報を入手する必要がない．攻撃者は悪意のあるスクリプトをユーザの Web ブラウザで実行することにより，ユーザのブラウザに記録されている正当なセッション情報にタダ乗りして，サーバに対してコマンドを発行することで，不正な目的を達成する．CSRF については次章で紹介する．

また，次章で同じく紹介する Local Network 攻撃は，ユーザのブラウザを踏み台として，ローカルネットワークにあるルータやプリンタ，他のサーバに対するポートスキャンなどを実行する．Web サーバを攻撃対象とするものではないため，Web サーバ側でのセッション管理では防ぐことのできない攻撃である．

以上のように，リッチなクライアントの機能を悪用した攻撃が Web 2.0 に特徴的であるといえる．

Web 2.0 の代表的攻撃手法

本章では，Web 2.0 に特徴的と思われる，クライアント側 JavaScript に特徴を持つ代表的な攻撃手法について説明する（なおブラウザによって個別に対策をとっており，紹介した攻撃には最新のブラウザでは動作しないものもある）．

■ クロスサイトスクリプティング（XSS）

クロスサイトスクリプティング（XSS）は，Web 1.0 の時代から続く代表的な攻撃手法であり，Web アプリケーション脆弱性の 70%を占めるといわれている☆2．XSS は，悪意を持ったユーザの入力に含まれるスクリプト（多くの場合 JavaScript）が別のユーザの閲覧するコンテンツ上で表示され，実行される攻撃手法である．技術的には古くから存在する攻撃手法ではあるが，Web 2.0 ではブログやソーシャルネットワークサービス（SNS）などのユーザ参加型のサービスが多いことが特徴であるため，XSS の影響は大きくなっている．

さらに，Ajax を利用した新しいタイプの XSS 攻撃が可能になった．XSS は従来，入力されたスクリプトが DB などに格納される「蓄積型 XSS」と，検索キーワードなどの入力データが Web ページ上にそのまま表示されることを利用した「非蓄積型 XSS」の 2 つに分類されてきた．しかし，Ajax の発達により，第三のタイプである「DOM-based XSS」と呼ばれる XSS が最近注目を浴びている☆3．DOM-based XSS は，Ajax アプリケーションにおいて，クライアント側 JavaScript が外部データを DOM 中に挿入するような設計となっている場合に，データ中にスクリプトを挿入することで可能になる攻撃である．**図 -2** は DOM-based XSS の例である．この例では，攻撃者は悪意を持つスクリプト文字列をリクエストパラメータとして含む URL を，スパムなどを通じて被害者に送信する．被害者がこの URL にアクセスすると，DOM-based XSS 脆弱性を持つクライアント側 JavaScript（**図 -3** 参照）を含む HTML 文書がブラウザ上にロードされる．ここで図 -3 の JavaScript は，ブラウザ組み込みの document.URL 変数からその文書自身の URL を読み出し，"name=" というリクエストパラメータの値を取り出して，それをユーザ名として Web ページ上に表示しようとする．しかし，リクエストパラメータの取り出し処理が不十分であるため，フラグメント識別子 # 以降の文字列もすべてユーザ名として取り出し，DOM に書き込んでしまう．結果として，# 以降に含まれるスクリプトが実行される．

XSS 対策としては Web サーバ側で入力データのフィルタリングを行い危険な文字列を取り除く方法が一般的であるが，スクリプトをエンコードして検出を防ぐ方法が多く発見されており，脆弱性の温床となっている．また DOM-based XSS の場合には，挿入されたスクリプトが完全にクライアント（ブラウザ）側だけで処理されるように構成することが可能であり，サーバ側での検出が難しい．たとえば上記の例ではスクリプト文字列は URL のフラグメント部分に含まれているが，一般的にブラウザは URL のフラグメント識別子以降はサーバに送信せずにブラウザ内でのみ処理するため，このような攻撃を Web サーバや Web Application Firewall（WAF）で検出す

☆2 WhiteHat Website Security Statistics Report, March 2008, Jeremiah Grossman, WhiteHat Security.

☆3 Klein, A. : DOM Based Cross Site Scripting or XSS of the Third Kind - A Look at an Overlooked Flavor of XSS, Web Application Security Consortium Article (July 4, 2005), http://www.webappsec.org/projects/articles/071105.shtml

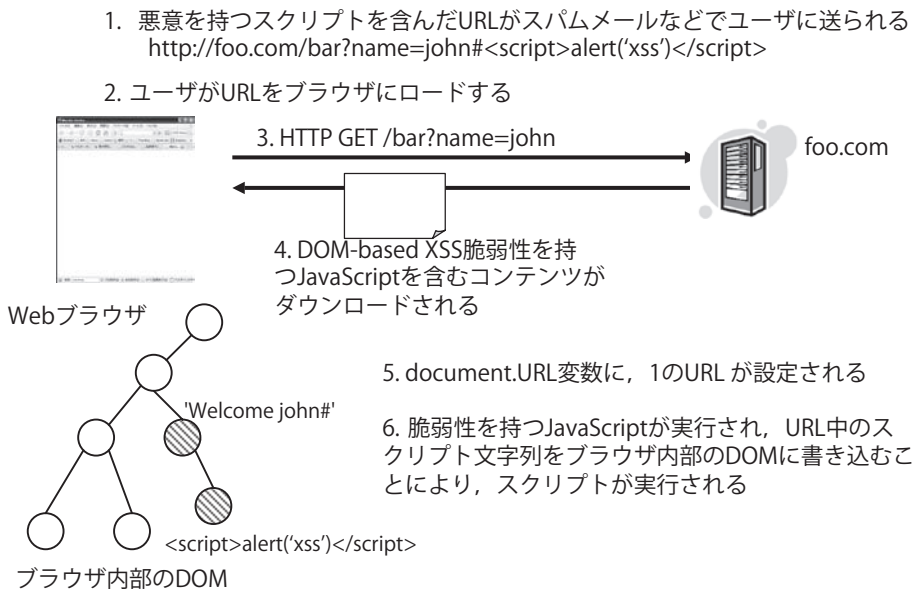

図 -2　DOM-Based XSS

るのは困難である．

悪意のあるスクリプトをブラウザ上で実行することによって，攻撃者はさまざまな方法でユーザに被害をもたらすことが可能になる．代表的な攻撃としては，Web アプリケーションのセッション情報を表す cookie や，ユーザのキー入力を盗んで遠隔地にいる攻撃者に送るというものがある．また，JavaScript を使って動的にコンテンツを書き換えることにより，フィッシングに悪用することも可能となる．一般的なフィッシング対策ツールではサーバの URL や SSL 証明書からフィッシングサイトを検出するが，XSS により書き換えられたコンテンツは正当なサーバからダウンロードされるため，このようなツールでの検出が難しい．

また，URL のリダイレクト機能を持つサーバを悪用して，悪意を持つ URL が直接ユーザの目に触れないようにすることも可能である．たとえば TinyURL.com では，任意の URL に対して http://tinyurl.com/5hmy6j のように短い URL を提供し，実際の URL にリダイレクトするサービスを行っている．前述のようにスパムメールで攻撃を含む URL を直接送る代わりにリダイレクトを利用することでユーザに検知されづらくすることが可能となる．

スタイルシートと Cross-Site Image Overlay

Cascading Style Sheet（CSS）は，HTML の表示を制御するために使われているスタイルシート言語である．CSS によるスタイルシートは HTML 中から参照したり HTML 中に <style> タグで直接埋め込んだりするだけでなく，個々の HTML 要素の style 属性に指定することもできる．Ajax の世界では，JavaScript によって CSS を書き換えることで，動的に表示を制御することができる．

```
<html><body>
…
<script>
  var pos=
    window.location.indexOf('name=')+5;
  var s = window.location.substring(pos,
                window.location.length)
  document.write('Welcome' + s);
</script>
…
</body></html>
```

図 -3　DOM-Based XSS 脆弱性を持つスクリプト

CSS のプロパティには値として URL を指定可能なものがあり，これを悪用した XSS が可能となる．たとえば CSS の background プロパティでは，背景として使うイメージの URL を指定することができる．ここで URL として javascript: で始まる仮想的 URL を指定した場合，続く文字列がスクリプトとして実行される．ほかにも，特定のブラウザでサポートされる expression 関数により，任意の JavaScript を実行することができる．

また，JavaScript を実行しなくても，スタイルシートそのものの操作による Cross-Site Image Overlay（XSIO）攻撃が可能であることも指摘されている[☆4]．たとえばオンラインショップのサイトで，ユーザが入力するレビュー情報の中にイメージを埋め込んだのち，さらにスタイルシートを指定することによってイメージを任意の位置に移動し，本来ユーザ入力の行われるエリア外のコン

☆4 Vetsch, S. : XSIO - Cross Site Image Overlaying (Aug. 7, 2007), http://www.disenchant.ch/blog/wp-content/uploads/2007/09/xsio.pdf

テンツ部分に表示することにより XSIO 攻撃が発生する．たとえばイメージがクリックされたときにフィッシングサイトへジャンプするようにハイパーリンクを指定しておいた上で，イメージをオンラインショップのトップページへジャンプするアイコン上に重ねて表示することで，別のユーザがそのアイコンをクリックした際にフィッシングサイトに誘導することが可能になる．

■ クロスサイトリクエストフォージェリ（CSRF）

クロスサイトリクエストフォージェリ（Cross-Site Request Forgery, CSRF）は比較的最近になって注目を浴びている新しいタイプの攻撃であり，Web アプリケーションの認証セッション管理の持つ脆弱性を悪用する．一般的にユーザ認証を行う Web サイトでは，ユーザがログインした後に認証クッキーをブラウザに返却する．ブラウザは一定の有効期間中この認証クッキーを保存しておき，期間内に同じサイトにアクセスする場合には，HTTP リクエストに認証クッキーを添付してサイトに送る．この仕組みにより，毎回パスワードを入力しなくても，サーバ側で認証済みのユーザかどうかを判別することが可能になる．

しかしこの方式では，ユーザが意識していなくても，ブラウザが HTTP リクエストを発行するたびに自動的に認証クッキーが送付されるという点に問題がある．たとえば，ユーザがオンラインショップのサイトにログインする．そこで得られたクッキーの有効期限が切れる前に悪意のある Web ページを閲覧したときに，そのページに含まれる JavaScript から，オンラインショップに対して，購入要求のような不正なコマンドを発行することが可能になる．このとき，HTTP で送られるリクエストにはブラウザが自動的に認証クッキーを添付するため，サーバ側ではログインしているユーザの意図した正規のコマンドとの見分けがつかない．

日本で有名な CSRF による攻撃として，2005 年に某大手 Social Networking Site で多くのユーザの日記に同一の文面の書き込みが行われた「はまちちゃん」事件がある☆5．また，CSRF によって，日記の書き込みだけでなく，書き込みの削除や退会など，HTTP によるコマンドで実現されているほとんどの機能を正当なユーザの権限により実行させることが可能となる．

XSS と異なり，CSRF では悪意のあるスクリプトを Web サイトに挿入する必要がない．このため潜在的に CSRF 脆弱性を持つ Web サイトは多く，今後表面化していくと考えられている．

☆5 古田雄介の " 顔の見えるインターネット "，「ぼくはまちちゃん！こんにちはこんにちは !!」，2007 年 9 月 3 日，Ascii.jp, http://ascii.jp/elem/000/000/063/63560/

```
  : // JSON文字列例
01: var jsonstr = "[ {'name': 'Sachiko Yoshihama',
  'destination': 'Barcelona', 'date': 'Aug 25, 2008' },
  {'name': 'Ai Ishida', 'destination': 'New York', 'date':
  'October 28 2008' }]";

  : // スクリプトの含まれる，不正なJSON例
02: var malstr = "[ {'name': 'Naohiko Uramoto',
  'destination': 'Okinawa', 'date': 'December 7, 2008' }];
  alert('xss');";

  : // JSON文字列を評価するとJavaScriptオブジェクトになる
03: var obj1 = eval(jsonstr);

  : // "Sachiko Yoshihama" が出力される
04: document.write(obj[0].name);

 : // malstrをevalすると、スクリプトが実行されてしまう
05: var obj2 = eval(malstr);
```

図 -4　JSON の例

JSON と JavaScript ハイジャッキング

Ajax では，JSON（JavaScript Object Notation）と呼ばれるテキストベースの軽量なデータ記述方式が多く使用されている．JSON では，データを名前・値のペアで表し，配列やオブジェクトなどの構造化されたデータを表現できる．JSON はプログラミング言語に依存しないが，JavaScript のサブセットであるため，特に Web アプリケーションで扱いやすいデータ形式である．文字列を JavaScript として評価する eval 関数によって JSON 文字列を評価すると，オブジェクトに変換される．しかし，この方法では JSON 文字列中に任意のスクリプトが含まれている場合に実行されてしまう危険がある（**図 -4** に例を示す）．そのため，JSON をオブジェクトに変換する場合には，eval 関数を使用する前に RFC4627 で推奨している正規表現によって JSON 構文への適合性をチェックするか，もしくは eval 関数を使用せずに構文解析を行って変換するのが望ましい．

JavaScript ハイジャッキングとは CSRF を応用した攻撃であり，ユーザ認証を回避して秘匿データを盗むことを可能とする[5), ☆6]．この攻撃手法では，オブジェクトの組込みメソッドを上書きすることにより，JSON 文字列がオブジェクトに変換される過程でデータを盗み取る．**図 -5** は JavaScript ハイジャッキングを行う攻撃の例である．攻撃者はまずオブジェクトのコンストラクタやプロパティの setter メソッドを書き換えておき，その中で当該オブジェクトの情報を盗んで攻撃者に送るような処理を埋め込んでおく．その状態で動的に <script> 要素を生成して，その src 属性に JSON を返却する攻撃対象の URL を指定する．するとブラウザから HTTP リクエストが発行されるが，この際，その URL がユーザ

☆6 Chess, B., O'Neil, Y. T. and West, J. : JavaScript Hijacking, Fortify Software (2007).

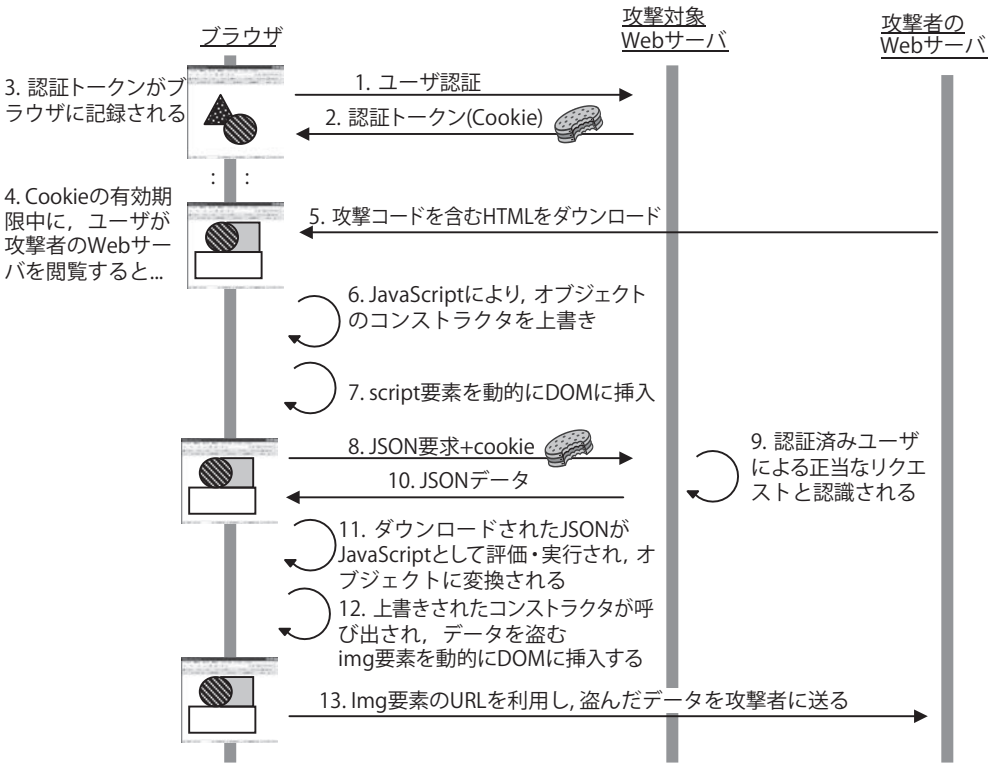

図 -5　JavaScript ハイジャッキングの攻撃例

認証で保護されていたとしても，CSRF によってすでに認証済みのユーザになりすましてリクエストを発行することが可能である．攻撃対象 URL から返却された JSON は <script> タグの機能により JavaScript として評価され，オブジェクトに変換されるが，その際に攻撃者の上書きしたコンストラクタや setter メソッドが呼び出され，その中で攻撃者が JSON 中のデータを盗むことができる．

■ ローカルネットワークに対する攻撃

JavaScript ハイジャッキングの例を見ても明らかなように，JavaScript で動的に HTML を操作することによって任意のサーバへのネットワークアクセスが可能になる．この機能を利用し，ユーザのブラウザを踏み台として，その近隣のサーバやデバイスに対する攻撃を行うことが可能になる．一般に JavaScript を含むクライアント側 Web アプリケーションはユーザの Web ブラウザ上で実行されるため，ファイアウォールの内側で保護されたイントラネット内にあるサーバなどへの攻撃が容易になる．

JavaScript によるポートスキャン

攻撃者にとって，サーバの存在する IP アドレスやポート，動いているサーバソフトウェアの種類を特定することは，実際に攻撃をする前の準備段階となる重要なステップである．JavaScript によるポートスキャンでは，HTML を動的に生成することによってネットワークアクセスを発生させ，アクセスの成否やエラーの種類からサーバの存在やポートの状態，さらには Web サーバの種類を特定する．

ネットワークアクセスを発生させるためには，たとえば JavaScript プログラムで動的に <img> 要素を生成し，src 属性に指定した URL に任意の IP アドレスとポート番号を指定する．属性値を書き込んだタイミングで，ブラウザが URL に対して HTTP リクエストを発行してアクセスする．ポートが開いている場合にはイメージがロードされて onload イベントハンドラが呼び出されるが，閉じている場合にはタイムアウトするという挙動の違いから，ポートの状態を判定できる．また，URL のパスとして，一般的な Web サーバにあらかじめインストールされているイメージファイル等を指定することにより，ファイルの有無からサーバの種類を判定できる．

Cross-Site Printing

ネットワークプリンタで認証を行わないタイプのものは，プリンタポートにコマンド文字列を送るだけで動作させることができる．たとえば HTML フォームのデータとしてプリンタへのコマンドを埋め込み，このフォームを JavaScript から動的に，プリンタポートに対して Submit することで印刷を行うという，Cross-Site

(a) : Flash内のActionScript例

```
getURL(par1); // par1は通常のURL
```

(b) : HTMLへの通常のFlash埋め込み例

```
<embed src='test.swf?par1=http://foo.bar.com/ ' >
```

(c) : HTML内でJavaScriptを挿入した攻撃例

```
<embed src="test.swf?par1=javascript:alert('mal');" >

<embed src="test.swf"
    FlashVars="par1=javascript:alert('mal')">

<object ...>
    <param name="movie" value="test.swf">
    <param name="flashVars"
           value="par1=javascript:alert('mal')" >
```

図 -6　Flash XSS 攻撃例

Printing 攻撃の可能性が報告されている☆7．この攻撃により，たとえば大量の印刷を行って紙を無駄遣いさせたり，広告メッセージを印刷させたりするなどの攻撃が可能になる．

Drive-by Pharming

Pharming（ファーミング）とは，phishing（フィッシング）の進化系であり，狙った獲物を釣り上げるフィッシングに対して，仕掛けの「種」を蒔いておくことで獲物を収穫するという，農業（farming）になぞらえた造語である．具体的には DNS 応答などを改ざんすることにより，ユーザが正規 URL にアクセスしようとしたときに，偽サイトに誘導するようなタイプの攻撃を指す．Drive-by Pharming[5)]，☆8 では JavaScript によるローカルネットワーク攻撃を応用し，家庭用ブロードバンドルータの HTTP インタフェースから DNS 設定を書き換えることにより，Pharming 攻撃を行う．家庭用ブロードバンドルータでは一般的に管理コマンドを使う前にユーザ認証が必要だが，パスワードを出荷時設定から変更していないような場合には容易に推測可能である．また，ブラウザにセッション情報が残っている場合には，CSRF を利用して認証を回避される可能性もある．

■ プラグインを悪用した攻撃

Web2.0 において Web コンテンツは単に文字を表示するだけでなく，音や動画なども一緒に提供するリッチコンテンツが主流になっている．Adobe Flash（以降 Flash）は，そういったインタラクティブなコンテンツを実現する手段の 1 つとして広く使用されている．またインターネットの閲覧が可能なマシンの 98%に Flash コンテンツを再生するための Flash Player がインストールされている．Flash では ActionScript という ECMAScript の拡張言語を利用してプログラムを記述することができる．このような状況の中で，Flash コンテンツとプレイヤーを使用した攻撃も増加している．

Flash を利用した攻撃は，Flash の脆弱性を利用するものと，Flash コンテンツ自体に攻撃コードが含まれる場合に大別される．前者の代表的な例としては，Flash コンテンツへの URL リクエストパラメータ等にスクリプトを挿入するものがある（**図 -6**）．Flash 内の ActionScript からは，URL リクエストパラメータを変数として参照することができるため，Flash コンテンツによっては，リクエストパラメータに指定された文字列を ActionScript の getURL 関数などを利用して URL としてアクセスしようとするものがある．しかし通常の URL の代わりに javascript: で始まる仮想的 URL が指定された場合，続くスクリプト文字列が実行されてしまう．

Flash コンテンツ自体に悪意がある場合，ActionScript から JavaScript の機能を呼び出したり，Flash コンテンツ内で動的に HTML を生成してその中で JavaScript を実行したりする手法により，ブラウザが持っている情報を盗んだり，HTTP リクエストを発行することで悪意のあるコンテンツを読み込んだり，ユーザをフィッシングサイトにリダイレクトしたりすることが可能になる．さらにこれらの攻撃を実現しているコードは Flash のファイルフォーマットである SWF 形式であるため，従来の HTML と Script を用いた攻撃よりもリアルタイムでの解析が難しい．

ここで説明した Flash を用いた攻撃は，既知の脆弱性が修正された最新の Flash Player で，セキュリティ設定をきちんと行うことにより回避できるものもある．しかし Flash Player のアップデートは強制されるわけではないため，いまだ旧バージョンの Player を使用しているユーザや，セキュリティの設定が正しく行われていない場合もあるため，Web サーバ側でフィルタリングを行い，悪意のあるコンテンツを削除するのが望ましい．

昨今の Web アプリでは，従来の HTML や JavaScript だけでなく，Flash などのコンテンツを統合して，リッチなユーザ体験を実現しようとするものが多い．HTML や JavaScript を用いた攻撃に対する対策が行われている場合でも，Flash を攻撃に用いることによってそれらの対策をすり抜けることが可能になるので注意が必要である．

■ マッシュアップにおけるセキュリティ問題

Web 2.0 では複数のサービスやコンテンツを統合して

☆7 Weaver, A. : Cross Site Printing (2007). http://aaron.weaver2.googlepages.com/CrossSitePrinting.pdf

☆8 Stamm, S., Ramzan, Z. and Jackobsson, M. : Drive-by Pharming, Indiana University Technical Report TR641 (Dec. 2006).

新たなアプリケーションを作り出す「マッシュアップ」が，必要に応じたアプリケーション（Situational application と呼ばれる）を迅速に開発するための手段として注目されている．しかし，一般的なマッシュアップ・アプリケーションでは，異なる信頼度に基づくコンテンツが同一の Web ページ上に存在するように構成されるため，悪意を持ったコンテンツが紛れ込んでいた場合に，XSS 同様に他のコンテンツを攻撃することが可能になる．

現在のブラウザのアーキテクチャは，同じ Web サーバや同一ドメインに属するサーバからダウンロードされたコンテンツは互いに信頼できるという考え方に基づいた Same-Origin Policy（同一起源ポリシー）と呼ばれるセキュリティモデルを取り入れており，ブラウザのウィンドウやフレームの単位でコンテンツの実行環境が分離され，互いにアクセスできないようになっている．

しかし，マッシュアップを行う場合には異なるサーバからダウンロードされたコンテンツ間のインタラクションを許す必要がある．そのために Same-Origin Policy を回避する手法として，大別してサーバ側マッシュアップとクライアント側マッシュアップの 2 つがある．

サーバ側マッシュアップとは，マッシュアップ・アプリケーションを提供する Web サーバ上で，他のプロバイダの提供するコンテンツを統合する手法である．また，ブラウザから他のドメインへのアクセスをプロキシとして仲介することにより，ブラウザが単一の Web サーバにアクセスしているように見せかけて，Same-Origin Policy を回避する．

一方でクライアント側マッシュアップとは，<script> タグ等を利用して，ブラウザ上の JavaScript アプリケーションから直接複数ドメインにアクセスする手法である．Same-Origin Policy はウィンドウやフレームにロードされる HTML 文書の単位で適用され，<script src="…" /> タグで読み込まれた JavaScript は，ダウンロード元 URL にかかわらず，親 HTML 文書と同一ドメインと判断されるので，他ドメインに属する JavaScript プログラムを HTML 中にインポートすることが容易に可能である．また，ネットワークアクセスについては Same-Origin Policy は XMLHttpRequest にも適用されるが，<img> タグや <script> タグは対象外であるため，これらの src 属性を動的に書き換えることによって HTTP リクエストを発行し，実質的に複数ドメインと通信を行うことが可能である．

いったん異なる信頼度のコンテンツがブラウザの同一ウィンドウ上で表示されると，両者の間でアクセス制御を行うことはできない．JavaScript は Java などと異なり，コードのスコープを分離して，変数や関数のアクセスを制限する機能を持たない．そのため，マッシュアップの結果同一ウィンドウ上で表示されるコンテンツに関しては，同一コンテキスト上で JavaScript が実行される．また，すでに宣言された変数や関数があっても，同名の変数や関数によって上書きすることが可能である．さらに，Web ページ内のコンテンツはブラウザの持つ DOM API によって互いにアクセス可能であるため，マッシュアップされたほかのサービスに属するデータを読み取ったり上書きしたりすることも容易に可能である．そのため，XSS と同等の攻撃が可能になる．

現状では，マッシュアップ・アプリケーションのセキュリティは，サービスプロバイダ間の信頼関係を前提にして成立しているといえる．しかし，マッシュアップしているサービスの中に XSS 脆弱性があった場合に，そのサービスを通じてマッシュアップ・アプリケーションが攻撃される危険もある．また，Web ページ上の広告スペースの又貸しによって実際のコンテンツの提供者が不明確になっている場合もあり[☆9]，マッシュアップにおけるセキュリティの確保は重要な課題といえる．

Web 2.0 セキュリティ問題への対策

本章では，前半で取り上げた攻撃手法のうち代表的なものについての対策方法を紹介する．現実には開発者や Web 管理者はこれらを自分の Web アプリケーションに当てはめて，それらに合った実装をしていく必要がある．また網羅的に脆弱性の有無を確認するために Web 脆弱性スキャナなどを利用することも有効である[☆10]．しかし，Web アプリケーションの攻撃手法は変化し続けるものであるため，常に新しい情報に目を配り，対策を取り続けることが非常に重要である．

■ XSS 攻撃検知とフィルタリング技術

残念ながら現状では，XSS の対策において「これだけやっておけば安心」という切り札はない．それは，Web アプリケーションが非常に多様であるために，1 つのアプリケーションで有効な対策が，常に別のアプリケーションでも有効とは限らないためである．また，Web 技術が 10 年以上前から後方互換性を保ちつつ発展し続けてきた結果，さまざまな本質的な弱点をも継承しており，多様なブラウザやプラグインの実装それぞれの脆弱性を

☆9 Provos, N., McNamee, D., Mavrommatis, P., Wang, K. and Modadugu, N. : The Ghost in the Browser : Analysis of Web-based Malware, First Workshop on Hot Topics in Understanding Botnets (HotBots'07) (Apr. 10, 2007).

☆10 ただし，現行の Web 脆弱性スキャナでは一般的に Ajax を多用した Web アプリケーションの自動解析がうまくいかない場合もあるため，注意深い使用が求められる．

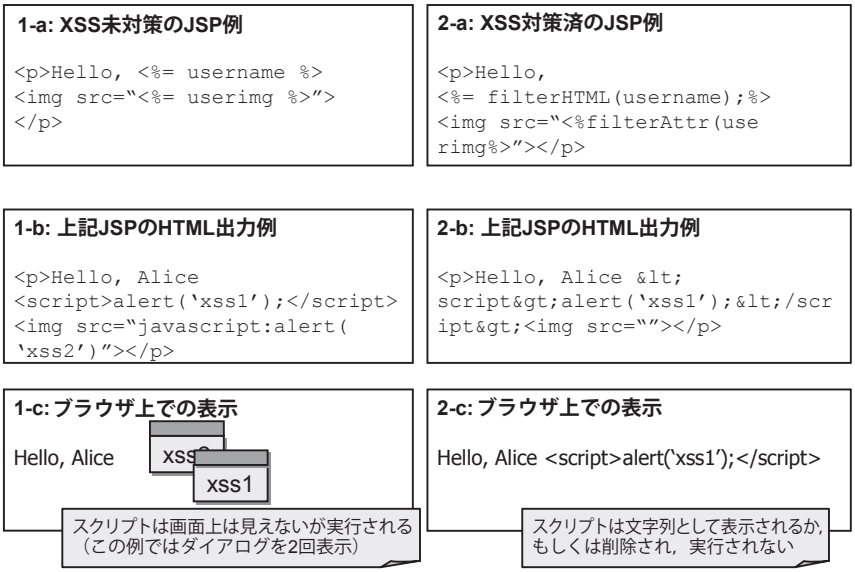

図 -7　JSP における出力サニタイズ例

悪用した攻撃が次々と発見されるためである．

現時点での現実的な対策は，単純であるが，**図 -7** のように Web アプリケーションの中で，ユーザの入力したデータが出力される前に必ずサニタイズを行い，実行可能なコードが出力されるのを防ぐことである．たとえば JSP であれば，動的にデータを出力する部分で必ず出力をサニタイズし，HTML タグなどをエスケープするメソッドを呼ぶようにしておく．また属性値として出力される文字列が，"javascript:" で始まる場合に削除する．ただしこれは最もシンプルな例であり，さまざまな難読化に対応する必要がある [5)]，☆11 また，前述のようにスタイルシートや Flash を悪用した攻撃にも留意する必要がある．また，サニタイズの見落としは脆弱性につながるため，汚染解析（taint analysis）ツールを利用して検証を行うことも有効である．また，フィルタリング方式としては，AntiSamy[5)]，☆12 のようにあらかじめ安全なパターンを登録しておきそれ以外を削除するホワイトリスト方式は，メンテナンスのコストはかかるが，未知の攻撃にも対応できるためにより安全である．

■ CSRF 対策

CSRF については，文献 1）で推奨されているように，URL のリクエストパラメータなどを利用して，第 2 の認証トークンを HTML ページ内から直接送ってサーバ側でチェックするのが現実的な対策として有効である．

URL リクエストパラメータにトークンを挿入するには，

☆11 より網羅的な難読化パターンについては http://ha.ckers.org/xss.html などを参照のこと．

☆12 OWASP AntiSamyProject : http://www.owasp.org/index.php/Category:OWASP_AntiSamy_Project

```
<form method="post" action="http://goodsite/action">
  <input type="text" name="abc" />
  <input type="hidden" name="auth"
         value="<%= token %>" />
</form>

<img
 src="http://goodsite/imgs/myimg.png?auth="<%= token%> />

<script type="text/javascript">
  var xhr = new XMLHttpRequest();
  xhr.open("http://goodsite/webapi/dosomething?auth="
           + token);
  xhr.send();
</script>
```

図 -8　CSRF 対応例

図 -8 のように HTML フォームの hidden フィールドを利用したり，Web ページを動的に生成する際に直接 URL 内にトークンを挿入したりする．ただし HTML 内に静的に埋め込まれた URL だけでなく，XMLHttpRequest などを使って JavaScript から発行される HTTP 要求についても漏れなく対応する必要がある．また，URL リクエストパラメータにトークンを挿入した場合，トークンを含む URL を攻撃者に観察されないように注意する必要がある．たとえば非 SSL 接続上で HTML 文書がダウンロードされる場合，経路上の攻撃者が通信を盗聴して HTML 内のトークンを盗むことが可能になる．このような攻撃を防ぐため，セキュリティを必要とする Web アプリケーションでは常に SSL による接続を必須とし，それ以外のページとトークンを共有しないようにする必要がある．

さらに，購入受付などの重要な場面では，直前でもう一度ユーザにパスワードを入力させることにより，認証

情報を知っている人間が介在していることを確認することで安全性を向上できる．

JavaScript ハイジャッキングは本質的には CSRF 攻撃の応用であるため，CSRF への対策を行うことで防ぐことができる．

■ ローカルネットワーク攻撃対策

残念ながら，現在の Web ブラウザの設計では，ローカルネットワーク攻撃そのものを防ぐのは困難である．Cross Site Printing や Drive-by Pharming については，プリンタやルータでユーザに認証を行うように設定し，パスワードを推測困難なものに変更しておくことが重要である．また，ユーザが意図的にブラウザからプリンタやルータにアクセスした後に，手動で Cookie を削除しておくことで，CSRF を併用した攻撃を防ぐことができる．

■ 安全なマッシュアップのためのフレームワーク

近年になり，以下に紹介するように，既存の Web ブラウザ上でマッシュアップを安全に行うためのフレームワークがいくつか提案されている．これらを利用することで，異なるドメインからダウンロードされたコンテンツの実行環境を分離し，安全に実行することができる．

Subspace

Subspace[2] はマッシュアップするコンテンツをブラウザの iframe（インラインフレーム）要素によって分離する方式を提案している．フレームは Same-origin policy によって制御されるので，異なるサーバからダウンロードされたコンテンツが iframe 内にロードされた場合，通常はそのままではマッシュアップ・アプリケーションとデータのやりとりをすることができない．ブラウザには document.domain 変数を親ドメイン名に書き換えることで Same-Origin Policy を回避できるという機能がある．Subspace ではこれを利用し，マッシュアップ・アプリケーションとコンテンツの間で限定的な通信を可能にする方式を提案している．ただし，ソースコードなどは公開されていない．

SMash

SMash[3] も，既存のブラウザ上で iframe によってコンテンツを分離し，安全なマッシュアップを可能にするフレームワークである．SMash ではマッシュアップされるコンテンツを iframe によりコンポーネント化して別のサーバからダウンロードすることでコンポーネントの実行環境を分離した上で，コンポーネント間の通信を可能にする下位レイヤ，イベント通信，上位のイベントハブの 3 層のソフトウェアスタックを提供する．また，下位レイヤの通信としてはフラグメント通信を採用している．これは iframe 内のコンテンツ URL を表す window.location は他ウィンドウからも書き込みが可能であるという特性を利用して，既存のブラウザ上でクロスドメイン通信を行う方式である．さらに，セキュリティトークンを使用してメッセージの改ざんやなりすましを検知する機構も備えている．上位通信レイヤのイベント通信やイベントハブは原則的に下位レイヤの実装に依存しないので，後述のようにブラウザがネイティブのクロスドメイン通信をサポートするようになった場合にも，プログラミングモデルを変更せずにアプリケーションを移行することが可能である．SMash はオープンソースとして OpenAjax Alliance によりソースコードが公開されている☆13．

今後の課題と展望

本稿で述べたような Web アプリケーションに対する攻撃は，Web のありかたが変化するに従い，現在のブラウザのセキュリティアーキテクチャが現実にそぐわないものになってきているのが本質な問題であるといえる．

1）Same-Origin Policy の問題：ユーザ生成コンテンツやマッシュアップにより，同一ウィンドウ内のコンテンツは互いに信頼できるという Same-Origin Policy における安全性の前提が崩れている．一方で，マッシュアップを実現するためにはドメインをまたがった通信機能が必要とされているが，現在のブラウザには明示的にはそのような機能がない．

2）ブラウザが HTML だけでなく，Flash や PDF，Java アプレットなどを実行するプラグインを統合するプラットフォームに変化している．その一方で，個々のプラグインはブラウザのサンドボックス外の実行権限を持つため，その脆弱性により Web アプリケーションやコンピュータ自体が危険にさらされる．

3）JavaScript が非常に柔軟な構造と機能を持つ言語であり，またブラウザが誤りのある HTML や JavaScript コンテンツをベストエフォートで実行しようとすることにより，脆弱性を生み出しやすい．

1）に関しては明示的なコンテンツ分離とクロスドメイン通信をブラウザの機能として導入する動きが広がっている．研究レベルでは，OS の実行空間分離をマッシュアップに適用した MashupOS[4] の提案などがある．

W3C では次世代の HTML として HTML5 の仕様を策定中であるが，ここでも異なるドメインからダウンロードされたコンテンツ間の明示的な通信を可能

☆13 http://sourceforge.net/projects/openajaxallianc/

にする postMessage 機能が提案されている[5), ☆14]．また，XMLHttpRequest Level 2 仕様[5), ☆15] では XMLHttpRequest の機能を拡張し，明示的なクロスドメイン通信をサポートする予定であり，その一部であるアクセスコントロール仕様[☆16] は Firefox3 ですでに実装されている[☆17]．また，Microsoft は Internet Explorer 8 でコンテンツの分離や明示的なクロスドメイン通信をサポートすることを発表している[☆18]．

2）についてはブラウザの各機能を分離して実行することにより，各機能の脆弱性が他に影響を及ぼさないようにする研究が進められている．たとえば文献 5）ではブラウザカーネルや JavaScript エンジン，各プラグインを OS レベルのサンドボックス内で実行することで各コンポーネントの脆弱性の影響を最小限に抑えるブラウザ実装を提案している．Google の発表した Chrome ブラウザ[☆19] も同様の構成をとっているが，動作の互換性を保つためにプラグインによるファイルやネットワークなどへのアクセスは制限されていない．

3）に関しては，JavaScript の元である ECMAScript Ver.4（ES4）で言語機能を大幅に拡張し，名前空間分離や型システム，Private スコープの導入が検討されていた．しかし先ごろ ES4 の多くの新機能は Web に適合しないという理由で標準化を中止し，現在の ver.3 に近い機能を次のバージョンとすることが発表された[☆20]．

一方で Caja[☆21] や ADSafe[☆22] のように，JavaScript の安全なサブセットを定義しようという動きもある．JavaScript プログラムがこれらのサブセットに適合することを実行前に静的に検証しておくことで，既存ブラウザの JavaScript 環境の上で安全に実行できるようにするというものである．

Web セキュリティ上の問題点の多くは，インターネット初期の閉鎖的な環境で，セキュリティ上の脅威が認知される以前に設計された，ユーザやコンテンツに対して寛容な基本技術を，根本的な変更なしに使い続けている点にある．上記 2），3）に見られるように，セキュリティのために Web 技術を大幅に改造しようとする動きはあるが，既存のコンテンツとの後方互換性のために採用に至らないケースも多い．後方互換性を保ちつつ，セキュリティを向上させていくのは今後の課題である．

☆14 HTML5, http://www.w3.org/html/wg/html5/
☆15 XMLHttpRequest Level 2, http://www.w3.org/TR/XMLHttpRequest2/
☆16 Access Control for Cross-Site Requests, W3C Working Draft 12 September 2008, http://www.w3.org/TR/access-control/
☆17 ただし 2008 年 11 月現在，Cross-Site XMLHttpRequest を利用できるのは Firefox の拡張機能などの特権を持つコードだけであり，Web アプリケーションからは利用できない．詳細は Cross-Site XMLHttpRequest, http://developer.mozilla.org/en/Cross-Site_XMLHttpRequest を参照のこと．
☆18 Microsoft, Better Ajax Development: Windows Internet Explorer 8. Whitepaper (Mar. 2008).
☆19 Google Chrome, http://www.google.com/chrome/
☆20 https://mail.mozilla.org/pipermail/es-discuss/2008-August/003400.html
☆21 Caja, http://code.google.com/p/google-caja/
☆22 ADSafe, http://www.adsafe.org/

参考文献

1）Johns, M. and Winter, J. : RequestRodeo : Client Side Protection against Session Riding, OWASP Europe Conference (May 2006).
2）Jackson, C. and Wang, H. : Subspace : Secure Cross-Domain Communication for Web Mashups, 16th International World Wide Web Conference (WWW'07), Banff, Alberta, Canada (May 8-12, 2007).
3）Keukelaere, F. D., Bhola, S., Steiner, M., Chari, S. and Yoshihama, S. : SMash : Secure Component Model for Cross-Domain Mashups on Unmodified Browsers, 17th International World Wide Web Conference (WWW'08), Beijing, China (Apr. 25-28, 2008).
4）Howell, J., Jackson, C., Wang, H. J. and Fan, X. : MashupOS - Operating System Abstractions for Client Mashups, 11th Workshop on Hot Topics in Operating Systems (HotOS XI), San Diego, CA (May 7-9, 2007).
5）Grier, C., Tang, S. and King, S. : Secure Web Browsing with the OP Web Browser, IEEE Symposium on Security and Privacy (2008).

（平成 20 年 9 月 30 日受付）

吉濱佐知子（正会員） sachikoy@jp.ibm.com

日本アイ・ビー・エム（株）東京基礎研究所専任研究員．トラステッド・コンピューティング，情報フロー制御，Web セキュリティなどの研究に従事．ACM 会員．

石田　愛（正会員） aiishida@jp.ibm.com

平成 18 年奈良女子大学院修士課程修了．同年日本アイ・ビー・エム（株）東京基礎研究所研究員．現在 Web セキュリティの研究にかかわる．日本データベース学会会員．

浦本　直彦（正会員） uramoto@jp.ibm.com

1990 年九州大学総合理工学研究科修了．同年，日本アイ・ビー・エム（株）入社．東京基礎研究所にて，自然言語処理，XML，Web サービス，SOA，Web 2.0 等に関する研究開発に従事．博士（工学）．