セキュリティガイドラインに準拠したアプリケーション作成支援に関する一提案

鈴木富明[†1]　白井丈晴[†2]　小林真也[†1]　川端秀明[†3]　西垣正勝[†1]

アプリケーションのセキュアコーディングの指針としてセキュリティガイドラインが公開されている．しかし，セキュリティガイドラインに従った設計を行うためには，アプリケーション開発者に相応の知識が要求される．セキュリティガイドラインに準拠したアプリケーションをもっと容易に作成することができれば望ましい．そこで本稿ではセキュリティガイドライン非準拠のアプリケーション（非準拠アプリ）をセキュリティガイドラインに準拠したアプリケーション（準拠アプリ）として動作させる仕組みの提案を行う．本稿では Self-Protecting 技術を用いて Web アプリケーションにおける DOM based XSS を防止するガイドラインに対する本方式の適用および有用性を検証する．

A Proposal for Making Support of Application that conforms to the Security Guidelines

TOMIAKI SUZUKI[†1]　TAKEHARU SHIRAI[†2]
SHINYA KOBAYASHI[†1] HIDEAKI KAWABATA[†3] MASAKATSU NISHIGAKI[†1]

Security guidelines are published for developing secure application to application developers. But to develop a secure application that conforms to Security guidelines, it is necessary to know about security. It is required that they develop it more easily.
Now we propose a new method to run the secure application that conforms to security guidelines it is also not. In this paper, we implement a this proposal in focus Web Application’s DOM-based XSS Security Guidelines and evaluate it.

1. はじめに

アプリケーションの脆弱性から引き起こされる被害が近年においても数多く報告されている[1]. 特に Web アプリケーションにおいてはクロスサイトスクリプティング（XSS）と呼ばれるエクスプロイトにより，悪意を持ったユーザによって任意の JavaScript コードが実行させられる踏み台として，そのようなアプリケーションが利用されている[2].

近年では Android アプリケーションにおいてセキュアコーディングの指針としてセキュアコーディングガイドが公開されるなど，アプリケーションの脆弱性を開発者の手によって防止する動きがある[3]. Web アプリケーションにおいても OWASP によるセキュリティガイドラインがセキュアコーディングの指針として公開されており，この活動が広がりを見せている[4].

しかし，セキュリティガイドラインに従った設計を行うためには，アプリケーション開発者に相応の知識が要求される[5]. セキュリティガイドラインに準拠したアプリケーションの作成において，アプリケーションの脆弱性を検知する方式や，脆弱性を持つアプリケーションへの入力を検証するライブラリなどが利用できると考えられているが，セキュリティガイドラインに準拠したアプリケーションをより容易に作成することができれば望ましい.

そこで本稿ではセキュリティガイドライン非準拠のアプリケーション（非準拠アプリ）をセキュリティガイドラインに準拠したアプリケーション（準拠アプリ）として動作させる仕組みを提案する．非準拠アプリを準拠アプリへと自動変換させることにより，知識のない開発者でも容易に準拠アプリを作成することが可能となることが見込まれる．本稿では Self-Protecting 技術[6]を用いてクライアントサイドの Web アプリケーションにおける脆弱性として現在もなお攻撃が盛んな DOM based XSS を防止するガイドラインへアプリケーションを準拠させる実装を行い，その有効性を検証する.

2. DOM-based XSS

2.1 XSS

クロスサイトスクリプティング（Cross-Site Scripting：XSS）は脆弱性をもつアプリケーションに対して，悪意あるユーザがスクリプトを挿入することにより，悪性なスクリプトをユーザのブラウザ上で実行させる攻撃である[2]. XSS はサーバ側の Web アプリケーションの脆弱性によって引き起こされる蓄積型 XSS（Stored XSS）および反射型 XSS（Reflected XSS），クライアント側の Web アプリケーションの脆弱性によって引き起こされる DOM-based XSS の三種類に大別される[7][8]. 一般的な XSS 対策として，アプリケーションへの入力の検証，アプリケーションからの出力の検証，テイント解析，入出力の無毒化などの対策が取られている.

†1 静岡大学大学院情報学研究科
Graduate school of Informatics, Shizuoka University.
†2 静岡大学情報学部
Faculty of Informatics, Shizuoka University
†3 (株)KDDI 研究所
KDDI R&D Laboratories, Inc.

ⓒ2015 Information Processing Society of Japan

2.2 DOM-based XSS

脆弱性を持つクライアント側の Web アプリケーションに対してスクリプトを挿入することによってユーザのブラウザ上で悪意あるスクリプトを実行する DOM-based XSS（Document Object Model based Cross Site Scripting）が報告されている[8]. この脆弱性の実体は JavaScript に存在する．動作の流れを次の二点にまとめる．

1. アプリケーションに対する悪性な入力
2. DOM 出力 API による悪性な入力の書き出し

攻撃の侵入経路としてアプリケーションに対する入力が存在する．Web アプリケーションにおける入力の代表例として HTTP リクエストのパラメータがある．JavaScript では document オブジェクトの location プロパティを参照することにより, パラメータを含む URL をアプリケーション内へ取得することができる．

一方で JavaScript には DOM に文字列の書き出しを行う API として document オブジェクトの持つ write プロパティが提供されている．document.write はスクリプトを含めた HTML コードの書き出しを行う API であるため，引数にスクリプトタグおよびその中で実行させるスクリプトを渡すことにより, JavaScript からスクリプトを動的に実行することが可能である．

つまり HTTP リクエストのパラメータに「<script>attack();</script>」といったスクリプトタグおよび悪性なスクリプトが入力され，window.location によりアプリケーションでこれを取得，document.write の API の引数にこれが渡されたときに，この悪性なスクリプトがアプリケーションにて実行されてしまう．図 1 にそのような DOM-based XSS の脆弱性を持つ JavaScript コードを記述する．

```
1: var url = window.location;
2: document.write(“The URL is “+ url +“.“);
```

図 1　DOM-based XSS の具体例

3. 既存研究

DOM-based XSS の対策として入力検証，テイント解析，セキュアコーディング，悪性なスクリプトの実行防止手法などが提案されている．

3.1 入力検証

従来 XSS 対策として適用された入力検証が DOM-based XSS に対しても有効とされている. DOM-based XSS に対する入力検証ツールとして NoScript[9]，XSS Filter[10]が公開されている．

NoScript は Mozilla-addons にて公開されている Firefox ブラウザ向けのプラグインであり，登録されたドメイン以外のスクリプトの読み込みを防止する機能や，悪性な HTTP リクエストのクエリを検知した際にリクエストに対して適切にサニタイズ処理を行う機能を備えている．これにより DOM-based XSS および Reflected XSS の対策として効果が期待できるが，リクエストに対するサニタイズにより，適切なリクエストを行えないことが指摘されている[11].

XSS Filter は IE8 に実装されている機能である．悪性な HTTP リクエストのクエリを検知した際に, HTTP レスポンスに悪性なスクリプトがエコーしたときに，これをサニタイズすることにより Reflected XSS を防止する方式である．NoScript における適切なリクエストを行うことができない課題は解決できているが，クエリに対するサニタイズがなされていないため，アプリケーションに残留する悪性なクエリ情報をブラウザから取得できてしまう．そのため DOM-based XSS に対して有効な対策とは言えない．

いずれの方式においても，入力のエンコードにより入力検証をすり抜ける攻撃が発見されている[12]．このため単なる入力検証は現在推奨されていない．

3.2 テイント解析

URL の GET パラメータなどのアプリケーションへの外部入力点を汚染源（Source）として，DOM への書き出し API などセキュリティ上重要な出力（Sink）まで汚染されたデータが渡されていなかどうかを調べるテイント解析と呼ばれる手法が XSS を検知するために用いられている．DOMinator[13]を初めとして，これまでに様々なテイント解析手法が提案されている．

FLAX[14]はバイトコードレベルでテイント解析およびファジングを行うことによりクライアント Web アプリケーションにおける XSS やコードインジェクションなどの脆弱性を検知する方式として提案されている．しかしその適用範囲は JASIL と呼ぶシンプルな JavaScript コードに限定され，かつ独自のブラウザにて検査を行っているため，実際の挙動を完全に把握できない．

Sebastian らはこれに対してオープンソースブラウザである Chromium の JavaScript エンジンに変更を施しテイント解析を行う手法を提案および実装している[15].

テイント解析においては一般的に汚染源の定義および汚染させる経路の定義が難しく，テイント漏れによる検知漏れやテイントのしすぎによる誤検知の問題がある．

3.3 セキュアコーディング

アプリケーション開発時に開発者が脆弱性を含む場所に対して適切な処置を施すことでアプリケーションの脆弱性を減らす，セキュアコーディングを行うためのライブラリが公開されている．開発者がアプリケーションにおける脆弱性の箇所を適切にライブラリによる処置を施すことで, アプリケーションは脆弱性を解消しながら適切な動作を行うことが期待される．

WordPress では PHP ライブラリとして入力検証を行い，危険な HTML タグを取り除く機能を持つ kses[16]が取り入れられている．これとともに引数へ渡した HTML タグを取り除いて返す関数も公開されている．同様のライブラリと

Ⓒ2015 Information Processing Society of Japan

して PHP 向けのライブラリ HTML Purifier[17], Java 向けのサニタイズライブラリ AntiSamy[18]，HTMLSanitizer[19]，様々なサーバ向けプログラムに加えて JavaScript もカバーするサニタイズライブラリ ESAPI[20]などが公開されている．しかし PHP などのサーバプログラムにおけるサニタイズ機能は DOM-based XSS に対しては有効ではない．

Django[21]，SafeHtml[22]，Ctemplate[23]などのテンプレート式のサニタイズライブラリも提供されている．これらは開発者がサニタイズを行いたい文字列をテンプレートで囲むことで適切なサニタイズが行われるものである．しかしこれもサーバプログラムにおけるサニタイズ機能しか持たないため，DOM-based XSS に対しては有効ではない．

いずれの方式においてもサニタイズを行う文字列やそのサニタイズ方法について開発者が手動で決定する必要があるため，正確に行うためには相応の知識が必要である．

3.4 悪性なスクリプトの実行防止

悪性なスクリプトの実行を防止する様々な手法が提案および公開されている．

XSS Auditor[24]は Google Chrome など WebKit をベースとしたブラウザに実装されている機能であり，HTML へ書き出された悪性なスクリプトを HTML パーサによって検出し，取り除くことにより悪性なスクリプトの実行を禁止している．しかし eval など HTML へ直接書き出さない関数による攻撃が成功することが指摘されている[15]．

ブラウザにおけるセキュリティ機能の実装としては Content Security Policy（CSP）が提供されている[25]．これはサーバの HTTP レスポンスのヘッダにセキュリティポリシを付加することによってブラウザにおけるアプリケーションの動作を制限する方式である．インラインスクリプトを防ぐ設定などを行うことで DOM-based XSS を含む攻撃を防ぐことが可能である．しかしサーバプログラムへのセキュリティポリシの設定が必要である．またブラウザ毎に対応状況が異なるため動作を保証しない．

BrowserShield[26]，BEEP[27]では開発者がアプリケーションにセキュリティポリシを記述することによって，クライアントブラウザ上で JavaScript のコードを書き換え，セキュリティ上重要な API をブラウザからフックすることによりセキュリティポリシに記述された API へのアクセスを限定する方式を提案している．同様に ConScript[28]においてはセキュリティ上重要な API に対してアスペクト指向プログラミング（Aspect Oriented Programming : AOP)を用いてブラウザ上でフックすることによりホワイトリスト型のセキュリティポリシをアプリケーションに適用する方式を実装している．しかし，いずれの方式においても，開発者によるセキュリティポリシの設定に加えてブラウザの変更も必要としている．そこで Phung らは Self-Protecting 技術を用いてそれらのセキュリティポリシの実装を JavaScript アプリケーションの変更のみで実現する方式を提案している[6]（詳細については 5.2 節参照）．またセキュリティポリシの実装に関してはオートマトンによる状態遷移モデルを利用することにより，より幅広いセキュリティポリシをサポートしている．

これらの方式においても開発者が適切なセキュリティポリシを設定する必要があるため，開発者には相応の知識が要求される．

4. 提案方式

4.1 コンセプト

本研究ではアプリケーション開発者がアプリケーション開発の際にセキュアコーディングやセキュリティポリシの設定を指針としていることに着目する．つまり開発者はアプリケーションをセキュリティガイドラインに準拠させることを意識していることに着目する．近年では Android においてセキュアコーディングガイドが公開されるなどして注目を浴びており，Web アプリケーションにおいても OWASP が現在策定を進めているため，これが注目を浴びることが予想される．しかしセキュリティガイドラインに準拠したコーディングを行うためには，3.3 節にて紹介したセキュアコーディング手法を用いる手段が現在公開されているだけであるため，完全にこれに準拠したアプリケーションを作成することは開発者の知識，ヒューマンエラーの観点より容易ではない．

そこで我々はセキュリティガイドライン非準拠のアプリケーションに対してセキュリティガイドラインに準じてアプリケーションを自動で変更する方式を提案する．具体的には Self-Protecting 技術[6]を用いてセキュリティ上重要な API に対してセキュリティガイドラインに準拠したラッピングをクライアント Web アプリケーションに施すことによりこれを実現する．アプリケーション開発者はこの自動変更をアプリケーションに対して施すだけでセキュリティガイドラインに準拠したアプリケーションを作成することができるため，開発者の知識不足やヒューマンエラーによるアプリケーションの脆弱性を解決することが可能となる．

4.2 プロシージャ

提案方式を適用した場合のアプリケーションの開発からクライアントにアプリケーションが届くまでの流れを図 2 に示す．

① アプリケーション開発者は自由にアプリケーションの開発を行う

② アプリケーションに対してセキュリティガイドラインに準拠した変更を自動で施す．この自動変更はアプリケーションのビルドツールやサーバプログラム（またはクライアントの実行環境（ブラウザ））などを通じて提供される．

③ 変更が施されたアプリケーションをサーバ上に公開する

④　クライアントはサーバに対してアプリケーションのリクエストを行う

⑤　サーバはリクエストに応じてクライアントへ変更が施されたアプリケーションを渡す

⑥　変更が施されたアプリケーションがクライアントの実行環境上で動作する（またはクライアントの実行環境上で変更を施した後に動作する）

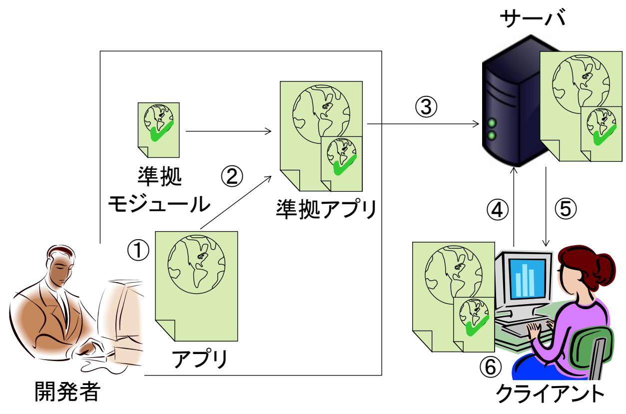

図 2　概要図

5. 実装

セキュリティガイドラインにおける DOM-based XSS に準拠したアプリケーションへと自動変更の詳細について説明する．

5.1 DOM-based XSS のセキュリティガイドライン

OWASP によって DOM-based XSS のセキュリティガイドラインが公開されている[29][30]．その基本的な理念は，セキュリティ上重要な API へと信頼できないデータを渡す際に適切なサニタイズを行うことである．

サニタイズに関しては 3.3 節で紹介した ESAPI などのセキュアコーディングのためのサニタイズモジュールを利用することが推奨されているためこれを用いる．

DOM-based XSS におけるセキュリティ上重要な API とは，スクリプトを含む文字列を DOM へ書き出すことができる API，javascript 疑似プロトコル[1]を設定できる API および dataURI スキーム[2]を設定できる API である

今回セキュリティガイドラインより実装の考慮をした API およびサニタイズ方針を表 1 に示す．なお (DOM).setAttribute については DOM に対して第一引数に渡された文字列の属性の値に第二引数に渡された文字列をセットする関数であるため，スクリプトをセットできる属性 onclick,onfocus などの「on*」，URL リクエストを行うことができる属性「src」，「href」，「action」，「formaction」が第一引数に現れたときの第二引数に対してサニタイズを施す．同様に setTimeout,setInterval は第二引数に設定した時間が来たときに第一引数の関数，または関数となる文字列を呼び出す関数であるため，第一引数が文字列のとき，これにサニタイズを行う．

表 1　実装を行う API とサニタイズ方針[3]

API	サニタイズ方針
document.write, document.writeln	HTML エンコード[4]および JavaScript エンコード[5]
(DOM).innerHTML, (DOM).outerHTML	HTML エンコードおよび JavaScript エンコード
Script.textContent (IE のみ innerText)	HTML エンコードおよび JavaScript エンコード
(DOM).setAttribute	「on*」：JavaScript エンコード 「src」，「href」，「action」， 「formaction」: URL エンコード[6]
(script/frame/iframe/embed).src，Anchor.href object.data，form.action， (Input/button).formaction	URL エンコード
window.location, document.location	URL エンコード
eval,Function	JavaScript エンコード
setTimeout,setInterval	JavaScript エンコード

5.2 Self-Protecting 技術

Self-Protecting 技術について説明を行う．Self-Protecting 技術の基本設計はビルトインメソッド（API）の呼び出しをインターセプトしてセキュリティポリシとの一致確認を行い，呼び出しの可否を判定することである．JavaScript におけるメソッドのインターセプトは次の二つの工程によって成り立つ．

- ビルトインメソッドの退避（退避したビルトインメソッドをオリジナルと呼ぶ）
- ビルトインメソッドのオーバーライドおよびオーバーライドした関数内でのオリジナルのコール

インターセプトの基本設計の実装を図 3 に示す． 図 3 において 2 行目のコードにより変数 original へとビルトインメソッド built_in を退避，3〜6 行目のコードによりビルトインメソッド built_in をオーバーライドしている．さらにオーバーライドしている関数内で 5 行目のコードによりオリジナル original を，call 関数によりビルトインメソッドの呼び出し時の引数をとって呼び出しを行っている．その際プログラムに設定されたセキュリティポリシとの一致確認を行う関数 security_check をオリジナルのコールを行う

1 「javascript:script();」の形でスクリプト script を実行することができる疑似プロトコル
2 「data:type,data」により type に指定したデータを data の URI から読み込むスキーム

3 この他にもブラウザ依存でスクリプトを実行できる API を持つが今回は除外する．
4 HTML として特殊な意味を持つ文字を特殊な意味を持たない文字へ置換する．「&」，「<」，「>」，「"」をそれぞれ「&」，「<」，「>」，「"」などと変換する
5 文字，数字以外の記号を全てユニコード「¥uXXXX」形式へ変更する
6 URL のパラメータに対するパーセントエンコーディング，なお javascript 疑似プロトコルは禁止とする

前に確認する処理を4行目に加えることでセキュリティポリシに基づいてビルトインメソッドの呼び出しの可否を決めることができる.

しかし，このままでは変数へ格納されたオリジナルを直接コールされてしまったときにビルトインメソッドへのインターセプトを行うことができない．そこでクロージャ[31]を用いる方法が取り入れられている．クロージャはJavaScript のスコープによりグローバルな名前空間の汚染をせずに変数へ値を格納しておくことができる技術である．図3では，1行目，7行目のコードにより無名な即時関数を作り，内部でインターセプトの記述を行うことにより，クロージャ内へ関数内でのみ有効な型 var で宣言されたoriginal 変数を保持しておくことができる．無名な即時関数内に保持された変数の呼び出しを直接行うようなビルトインメソッドは用意されていないため，original に保持されたオリジナルを無名な即時関数外のスコープから直接コールすることはできない.

さらにこれらのインターセプトをアプリケーションにおける JavaScript プログラムの開始位置（html ファイル内の head タグの始め）に付加（外部ファイルとして保持し<script src=”policy.js”></script>のような形で付加）することで，開発されたアプリケーションのビルトインメソッドのコールを全てインターセプトすることが可能となる.

なお JavaScript におけるオブジェクトのプロパティはデータプロパティとアクセサプロパティと呼ばれる二種類に分類され，それぞれ呼び出し方法，オーバーライド方法が異なる[32]．ビルトインメソッドもプロパティであるため，この実装においてもそれぞれの実装が必要である.

5.2.1 データプロパティ

データプロパティは単純な変数（値，関数など）を保持する一般的なプロパティである．プロパティの取得，オーバーライドの操作は通常の代入操作によって行うことが可能であるため図3の形によりインターセプトを実現できる.

5.2.2 アクセサプロパティ

アクセサプロパティは set/get 属性を持つプロパティである．取得には Object.getOwnPropertyDescripter 関数を必要とする．またオーバーライドの操作には Object.defineProperty 関数を必要とする．図4にプロパティディスクリプタのインターセプトの実装を示す.

5.2.3 オーバーライドできないアクセサプロパティ

プロパティは変更が可能かどうかの属性情報を格納する configurable 属性が存在する．configurable 属性に false が設定されていた場合，プロパティのオーバーライドを行うことができない．そのため，このようなプロパティが存在する場合，異なる実装アプローチをとる必要がある.

このようなプロパティの代表としてアクセサプロパティである window.location,document.location がある．Phung らはこれに対して watch 関数[7]を使用することにより代入を検知する試みを行っている．図5に window.location に対する実装を示す.

```
1: (function(){
2:    var original = built_in;
3:    built_in = function(){
4:      if(security_check(arguments)==”OK”)
5:        return original.call(this,arguments);
6:    }
7: })();
```

図 3　データディスクリプタのラッピング

```
(function(){
  var original =
    Object.getOwnPropertyDescriptor(built_in,”method”);
  Object.defineProperty(built_in,”method”,
  {set: function(){
     if(security_check(arguments)==”OK”)
       return original.set.call(this,arguments); },
   get: function(){ // getter のインターセプト
     if(security_check()==”OK”)
        return original.get.call(this); }
  });
})();
```

図 4　プロパティディスクリプタのラッピング

```
// obj.watch(“method”,function(id,old,new){~})
// obj オブジェクトの method プロパティの監視を
// 行う，代入された値が第三引数に渡され，
// 返り値が実際の値の代入となる
window.watch(“location”,function(id,oldloc,newloc){
  if(security_check(arguments)==”OK”)
    return original.call(this,arguments);
});
```

図 5　window.location の実装

5.3 Self-Protecting 技術による DOM-based XSS の防止

セキュリティガイドラインに準拠したアプリケーションへと変更させるためには，表1に示した API をインターセプトして，同じく表1に示したサニタイズを施した値をオリジナルの引数に与えることで実現できると考えられる．全ての API コールをインターセプトし，かつクライアント側の Web アプリケーションに対する簡単なプログラムの変更のみでこれを実現するためには，前節で説明を行った Phung らの Self-Protecting 技術が有効である．しかし Phung らの提案した方式では，ビルトインメソッドを呼び出そう

7 JavaScript 言語の標準規格 ECMAScript には存在しないが Gecko エンジンにて採用されている，最近では ECMAScript 標準として策定が進む observe 関数を用いた監視方法を利用する方法も存在する[33].

としたときの引数の値を状態の判定後そのままオリジナルの引数へ与えている．そこで我々はサニタイズした値を，オリジナルを呼び出す際の引数に与えるよう変更を行った．実装例として，データプロパティである document.write への実装を図 6 に，アクセサプロパティである innerHTML への実装を図 7 に，window.location への実装を図 8 に示す．なお図 6，7 における sanitize 関数は表 1 に示したサニタイズ関数を適切に選択して引数にこれが施されているものとする．これらを Phung らの方式と同様にアプリケーション実行の初めである head タグの始めに付加する．

```
(function(){
  var original = document.write;
  document.write = function(){
    return original.call(this,sanitize(arguments));
  }
})();
```

図 6 document.write のサニタイズ

```
(function(){
  var original = Object.getOwnPropertyDescriptor(
  Element.prototype,"innerHTML");
  Object.defineProperty(Element.prototype,"innerHTML",
  {set: function(){
     if(security_check(arguments)=="OK")
       return original.set.call(this,sanitize(arguments));},
   get: function(){
     if(security_check()=="OK")
         return original.get.call(this);
   });
})();
```

図 7 innerHTML のサニタイズ

```
window.watch("location",function(id,oldloc,newloc){
  var url = new URL(newloc);
  if(url.protocol=="http:"||url.protocol=="https:"){
   return url.protocol+"//"+ url.host + url.pathname
              + sanitize(url.search);
});
```

図 8 window.location のサニタイズ

6. 実験

セキュリティガイドライン非準拠で DOM-based XSS に対して脆弱なアプリケーションに対し提案方式を施すことにより，DOM-based XSS の攻撃となるコードを防止できるかどうかを実験により検証する．

まず DOM-based XSS の脆弱性を持つアプリケーションとして，表 1 に示した API に対して URL のパラメータを渡すアプリケーションを作成した．これに対して OWASP Xenotix[34]に含まれる XSS となるペイロード 4,808 個を URL のパラメータへ渡すことによって，パラメータへと渡したペイロードによってスクリプトが実行されるかどうかの検証を行った．実行環境については次の通りである．

・OS：Window8.1
・CPU：Inter(R)Core(TM)i5-3337U
・ブラウザ：Firefox 35.0.1
・サーバ：Apache Tomcat 8.0.14
・Source API：window.location.search 8

実験結果を表 2 に示す．

表 2 実験結果

API	動作スクリプト数	提案方式実装後に動作したスクリプト数
document.write	1033	0
innerHTML	522	0
textContent	72	0
(DOM).setAttribute	44	0
(DOM).href	12	1
window.location	12	1
eval	71	0
setTimeout	74	0

7. 考察

提案方式においても攻撃が成功するペイロードについて，実装に対する攻撃，適用範囲について考察を行う．

7.1 攻撃に成功するペイロード

6 章の実験において攻撃が成功したペイロードは，悪性なサイトを示す URL を挿入するコードである．セキュリティガイドラインにおける URL の検査は項目には含まれていないためにこの攻撃は成功した．

また今回のペイロードには含まれなかったが，次のようなペイロードには対応できない．

- 悪性なスクリプトを持つ URL
- XSS 脆弱性を持つページに悪性なペイロードを与えた URL（パラメータのサニタイズをすり抜けるもの）
- サニタイズを通りぬけるペイロード（ブラウザの脆弱性をつくもの，ライブラリの穴をつくもの，etc）

URL のペイロードに対しては，悪性な URL かどうかの検査が有効である．この検査は Phung らが実装を行っているため，セキュリティガイドラインにその項目が足されれば対処が可能である．

サニタイズを通り抜けるペイロードに対しては，サニタイズ方法の改善やホワイトリスト方式などがセキュリティガイドラインに導入されることによって軽減できることが期待される．

8 Firefox35.0.1 ではデフォルトでブラウザのリクエストの時点で URL エンコードが行われているため，検証のため取得時に適切にデコードを行う

ⓒ2015 Information Processing Society of Japan

7.2 Self-Protecging 技術の課題

Self-Protecging 技術には様々な攻撃方法が存在することが報告されている[6][35]．開発者が能動的に攻撃することは考えにくいが，セキュリティガイドラインへの準拠を示すためには対策が必要であるため, これについて検討する．

7.2.1 実体の参照

JavaScript はプロトタイプベースの言語であり, 全てのオブジェクトは Object オブジェクトのプロトタイプに記述されたプロパティを継承している．DOM への書き出しを行う API である document.write を例に挙げて説明を行う．

document.write と呼び出しを行った際，document オブジェクトの write プロパティを見た目上参照しているように見えるが，実際のところ document オブジェクトに write プロパティは宣言されていない．このとき JavaScript エンジンでは document オブジェクトのプロトタイプのオブジェクトのプロパティの参照を行う．そのオブジェクトにもプロパティが存在しなければ，さらにそのプロトタイプのオブジェクトへと参照が行われる．そのようなオブジェクトのチェーンをプロトタイプチェーンと呼ぶ．このとき HTMLDocumentPrototype オブジェクトに write プロパティを見つけ，これを参照して DOM への書き出しが実際には行われている．そのため実装において，このようにビルトインメソッドの実体をインターセプトする必要がある．同様のプロトタイプチェーンを JavaScript にて行う手法が文献[35]にて紹介されているため，図 9 にこれを示す．

7.2.2 ビルトインメソッドの参照

Self-Protecting 技術においてはビルトインメソッドのオーバーライドを行っているが, ビルトインメソッドを生成，参照する方法が知られている. JavaScript はページが生成される際に window オブジェクトを作成する．そのときにビルトインメソッドが window の配下に作成されるため，window オブジェクトを作成するようなビルトインメソッドによってそれらは生成される．これが可能なビルトインメソッドが文献[6]では二つ紹介されている．一つは window.open，もう一つは frame/iframe の contentWindow を利用したものである．Phung らはこれらの呼び出しを禁止することで対策を行っているが，セキュリティガイドラインに準拠したアプリケーションへと変更するという目的の達成を考えると，そのような実装は非常に大きな制約になるものと我々は考える．そこで我々はそれらから生成された window となるオブジェクトに対して 5.3 節で紹介したビルトインメソッドと同様のサニタイズを施す実装を行うことを考える．window.open に対する実装を図 10 に示す．

ただし frame/iframe における contentWindow は DOM への書き出しが終わった後に生成されるため，アプリケーションの読み込み時には参照ができず，オーバーライドなど行うことができない．そのため frame/iframe を含む DOM の読み込みが終わった時や DOM 変更時など iframe が挿入された可能性があるときに iframe に対するサニタイズの処理を施す必要がある．これについては現在実装を検討中である．

```
// - object.hasOwnProperty(method)
//   object が method プロパティを持つときに
//   true を，そうでなければ false を返す
// - Object.getPrototypeOf(object)
//   object のプロトタイプを返す
while(!object.hasOwnProperty(method)
        && Object.getPrototypeOf(object)){
        object = Object.getPrototypeOf(object);
 }
 obj.method // ビルトインメソッドの実体
```

図 9 実体の参照

7.2.3 その他の攻撃

インターセプトを行っている中で利用している関数に対する攻撃が考えられる．本方式での無毒化に， escape 関数[9]を用いた場合を想定する．ここで escape 関数をインターセプトし，エスケープ処理を行わずに返す処理を行う関数へと変更が行われた場合に無毒化が働かなくなってしまうことが想定される．これに対してはビルトインメソッドをオリジナルと同様無名な即時関数内に保持し，サニタイズ処理をする際にこれを用いることで解決できる．

watch 関数は後から設定したものに上書きされる仕様となっている．このため window.location，document.location に対する watch 関数を上書きされないような実装がなされている必要がある．これに対しては watch 関数をインターセプトし，window.location，document.location への新しい watch 関数を登録させない，もしくは開発者が定義しようとする関数へ，最初に登録したものを内包させる必要がある．しかし watch 関数は引数にプロパティ名をとるが，対象のオブジェクトをとらないため，どのオブジェクトに対して watch 関数を作成しているのか判別することができない．これには検討が必要である．

7.3 本方式の限界

本方式はアプリケーションをセキュリティガイドラインに準拠したアプリケーションとして動作させる仕組みであるため，元のアプリケーションがこれに準拠した上で正常に動作することまでは保証しない．原因と影響の大きさについては現在調査中である．これに対してはセキュリティガイドラインにおいてテイント手法を取り入れるなど，サニタイズの粒度を上げることで改善することが期待される．また本方式はアプリケーションをセキュリティガイドラインに準拠させることを目標としているため，その性能はセキュリティガイドラインに紹介されている方法に依存する．

9 非 ASCII 文字列を 16 進数のエスケープシーケンスに文字列を置換する関数

ⓒ2015 Information Processing Society of Japan

```
// - function.apply(thisArg[,argsArray])
//   関数 function の呼び出しを行う, その際の this を第一引
// 数に，引数を第二引数に配列型で指定する
// - Array.prototype.slice.call(arguments)
//   arguments を配列型へ変換
var win_open = window.open;
window.open = function(){
  return open_wrap.apply(this,
[argumets:arguments,_win_oepn:win_open]);}
function open_wrap(arg){
  var win = arg._win_open.apply(this,
      Array.prototype.slice.call(arg.arguments));
  var _win_open_local = win.open;
  win.open = function(){
    return open_wrap.apply(this,
      [arguments:arg.arguments,_win_open:_win_open_local]);
  }; return win; }
```

図 10　window.open に対する実装

8. おわりに

本稿では Self-Protecting 技術を用いてセキュリティガイドライン非準拠のアプリケーションをセキュリティガイドラインに準拠したアプリケーションとする方式の提案を行った．本方式によりセキュリティガイドラインに準拠したアプリケーションへと自動変更を行うことが可能であるため，アプリケーション開発者は開発したアプリケーションを容易にセキュリティガイドラインに準拠させることが可能となる．提案方式を Web アプリケーションにおける DOM-based XSS のセキュリティガイドラインに準拠させる実装を行い，攻撃に対する有効性を示した．

本方式への攻撃耐性および適用による影響について今後調査を進める予定である．

謝辞　論文執筆にあたり，KDDI 研究所竹森敬祐氏にご助言を頂いたためここに謝意を表する．

参考文献

1)　IPA ソフトウェア等の脆弱性関連情報に関する届出状況 [2014 年第 4 四半期（10 月～12 月）] http://www.ipa.go.jp/security/vuln/report/vuln2014q4.html
2)　CERT. Advisory ca-2000-02 Malicious Html Tags Embedded in Client Web Requests. http://www.cert.org/advisories/CA-2000-02.html
3)　Android アプリのセキュア設計・セキュアコーディングガイド http://www.jssec.org/dl/android_securecoding.pdf
4)　OWASP AJAX Security Guidelines https://www.owasp.org/index.php/OWASP_AJAX_Security_Guidelines
5)　atmarkit なぜ、いま「セキュアコーディング」なのか？(1/2) http://www.atmarkit.co.jp/ait/articles/1210/24/news005.html.
6)　Phung, Phu H. et al:"Lightweight self-protecting JavaScript." Proceedings of the 4th International Symposium on Information, Computer, and Communications Security. ACM, 2009.
7)　“Cross-Site Scripting”, Web Application Security Consortium, February 23rd, 2004 http://www.webappsec.org/projects/threat/classes/cross-site_scripting.shtml
8)　“DOM Based Cross Site Scripting or XSS of the Third Kind” (WASC writeup), Amit Klein, July 2005 http://www.webappsec.org/projects/articles/071105.shtml
9)　Georgio Maone. Noscript., http://www.noscript.net/.
10)　David Ross. IE8 Security Part IV: The XSS Filter , http://blogs.msdn.com/b/ie/archive/2008/07/02/ie8-security-part-iv-the-xss-filter.aspx
11)　Stock, et al. "Precise client-side protection against dom-based cross-site scripting." Proceedings of the 23rd USENIX security symposium. 2014..
12)　Bates et al: "Regular expressions considered harmful in client-side XSS filters." Proceedings of the 19th international conference on World wide web. ACM, 2010.
13)　Stefano Di Paola. DominatorPro: Securing Next Generation of Web Applications. https://dominator.mindedsecurity.com/
14)　Saxena, et al. "FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications." NDSS. 2010.
15)　Lekies et al:"25 million flows later: large-scale detection of DOM-based XSS." Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. ACM, 2013.
16)　SourceForge.net kses http://sourceforge.net/projects/kses/
17)　HTMLPurifier http://htmlpurifier.org/
18)　OWASP AntiSamy https://www.owasp.org/index.php/Category:OWASP_AntiSamy_Project
19)　OWASP Java HTMLSanitizer https://www.owasp.org/index.php/OWASP_Java_HTML_Sanitizer_Project
20)　OWASP Enterprise Security API https://www.owasp.org/index.php/Category:OWASP_Enterprise_Security_API
21)　Django http://www.djangoproject.com/
22)　GWT SafeHTML http://www.gwtproject.org/javadoc/latest/com/google/gwt/safehtml/shared/SafeHtml.html
23)　Ctemplate https://code.google.com/p/ctemplate/
24)　XSS Auditor, https://github.com/WebKit/webkit/blob/master/Source/WebCore/html/parser/XSSAuditor.cpp
25)　Mozilla Developer Network. Introducing Content Security Policy. https://developer.mozilla.org/en/Introducing_Content_Security_Policy
26)　Reis et al "BrowserShield: Vulnerability-driven filtering of dynamic HTML." ACM Transactions on the Web (TWEB) 1.3 (2007): 11.
27)　Jim et al: "Defeating script injection attacks with browser-enforced embedded policies." Proceedings of the 16th international conference on World Wide Web. ACM, 2007.
28)　Meyerovich et al "ConScript: Specifying and enforcing fine-grained security policies for Javascript in the browser." Security and Privacy (SP), 2010 IEEE Symposium on. IEEE, 2010.
29)　OWASP DOM based XSS Prevention Cheat Sheet　https://www.owasp.org/index.php/DOM_based_XSS_Prevention_Cheat_Sheet
30)　OWASP XSS Prevention Cheat Sheet https://www.owasp.org/index.php/XSS_%28Cross_Site_Scripting%29_Prevention_Cheat_Sheet
31)　MDN クロージャ https://developer.mozilla.org/ja/docs/Web/JavaScript/Guide/Closures
32)　Internet Explorer Dev Center, Data Porperties and Accessor Properties https://msdn.microsoft.com/en-us/library/ie/hh965578%28v=vs.94%29.aspx
33)　GitHub ECMAScript observe https://github.com/arv/ecmascript-object-observe
34)　OWASP Xenotix XSS Exploit Framework https://www.owasp.org/index.php/OWASP_Xenotix_XSS_Exploit_Framework
35)　Magazinius,et al "Safe wrappers and sane policies for self protecting javascript." Information Security Technology for Applications. Springer Berlin Heidelberg, 2012. 239-255.

ⓒ2015 Information Processing Society of Japan