Vol.2019-CSEC-87 No.5
2019/12/4

既存のWebアプリケーションへの適用性を考慮したプリミティブなTrusted Typesによる Client-Side XSS防御手法の提案

山崎 勇二[1,a)]　垣内 正年[2]　新井 イスマイル[2]　藤川 和利[2]

概要：現在，Webはインターネットの中でも最も重要なプラットフォームの一つとなっている．その一方で攻撃者が脆弱なWebアプリケーションに悪意のあるスクリプトを挿入し実行させるクロスサイトスクリプティング (XSS) の脆弱性が問題となっており，近年ではクライアント側の機能増加に伴い，XSSの一種であるClient-Side XSSが問題となっている．この問題に対処するために，テイントトラッキングによる検知・防御策が提案されているが，パフォーマンスに影響が発生し，実用的なものが無いのが現状である．本研究では，パフォーマンスと既存のWebアプリケーションへの影響を小さくしたClient-Side XSSの防御を目的とする．具体的には，安全な文字列であるかを型で検証するTrusted Typesを，プリミティブな型として定義することによる防御手法を提案する．Trusted Typesを用いることにより，JavaScriptを生成する実行シンクは開発者が安全な文字列であると明示した文字列のみを許可するため，Client-Side XSSの脆弱性を塞ぐことが可能となり，加えて防御に必要な処理は文字列の型を検証するのみとなるため，パフォーマンスへの影響を軽減しながら防御が可能となる．このTrusted Typesをプリミティブな型として定義し，JavaScriptソースの構文解析の段階で割り当てることで，既存のWebアプリケーションに変更を加えずTrusted Typesを適用させる．

キーワード：Webセキュリティ，XSS，Client-Side XSS，Trusted Types

1. はじめに

現在，Webはインターネットの中でも最も重要なプラットフォームの一つとなっており，コンテンツが配置してあるサーバの機能が増加している．その一方で，Webブラウザなどのクライアント側でも機能が増加していることが特徴として挙げられる．その中でもJavaScriptは，人気のあるWebアプリケーション、ブラウザー拡張機能（またはアドオン)、HTML5モバイルアプリケーション（WebView、Windows 8 Metroアプリ）など，クライアント側で動作するスクリプト言語として広く普及している．

その一方で問題となっているのが攻撃者が脆弱なWebアプリケーションに悪意のあるスクリプトを挿入し実行させるクロスサイトスクリプティング (以下XSS) の脆弱性であり，2000年に最初に発見されて以来 [1]、XSSは、Webアプリケーションにおけるセキュリティ上の問題とされている．さらに近年では，JavaScriptの安全でない記述によりクライアント側で発生するClient-Side XSSの脆弱性が問題となっている．2013年の調査ではAlexa Top 5000の約10%に少なくとも1つのClient-Side XSSの脆弱性を含んでいる [2] ことが示されており，この脆弱性が最新のWebアプリケーションにおいても依然として広く蔓延している [3] ことが示されている．

このClient-Side XSSの問題に対処するために，テイントトラッキングを用いて実行シンクに格納される文字列を検査し，検知・防御を行う手法 [4] や，安全でないコードを安全なコードに置換するパッチを自動生成することによる防御策 [5] などが提案されているが，Webのパフォーマンスに影響が発生する，または防御手法の導入が容易でないなど課題が多く，依然として実用的なものが無いのが現状である．

このような背景で，近年Client-Side XSSの防御策とし

[1] 奈良先端科学技術大学院大学先端科学技術研究科
Graduate School of Science and Technology, Nara Institute of Science and Technology, 8916-5 Takayama-cho, Ikoma, Nara
[2] 奈良先端科学技術大学院大学総合情報基盤センター
Information Initiative Center, Nara Institute of Science and Technology, 8916-5 Takayama-cho, Ikoma, Nara
[a)] yamasaki.yuji.yx6@is.naist.jp

```
let name = "Your name is " + location.hash.slice(1);
document.write(name);
```

図 1　Client-Side XSS の脆弱性を含むコードの例

て議論されているのが，Trusted Types[6] である．この Trusted Types を用いることにより，開発者が安全な文字列であると明示した文字列のみ，JavaScript を生成する実行シンクへの格納を許可するため，Client-Side XSS の脆弱性を塞ぐことが可能となる．加えて，防御に必要な処理は文字列の型を検証するのみとなるため，パフォーマンスへの影響を軽減しながら防御が可能となる．一方で課題となるのが，Trusted Types を既存の Web アプリケーションに適用させるにはコードの変更が必要となり，近年の大規模な Web アプリケーションに対して変更を行うことが現実的でない点が挙げられている．

本研究では，パフォーマンスと既存の Web アプリケーションへの適用性を考慮した Client-Side XSS の防御を目的とし，安全な文字列であるかを型で検証する Trusted Types を，プリミティブな型として定義することにより，既存の Web アプリケーションに変更を加えずに Trusted Types を適用し，Client-Side XSS を防御する手法を提案する．

2. Client-Side XSS の概要

本章では，本研究で対象とする Client-Side XSS の概要と従来の XSS 防御策の回避，攻撃モデルに関して述べる．

2.1 Client-Side XSS

Client-Side XSS は，2005 年に Amit Klein[7] によって DOM Based XSS という用語で議論が始まったものであり，脆弱なサーバ側のプログラムによって引き起こされるのではなく，ユーザが制御可能な入力がクライアント側で安全に処理されない場合に発生する．したがって，eval() や document.write などの，文字列を引数として受け取り解析して JavaScript コードとして実行するシンクに，URL や location.hash などの入力ソースに格納されている文字列が渡される場合，そのコードは Client-Side XSS の脆弱性を含んでいることとなる．

図 1 に Client-Side XSS の脆弱性を含むコードの例を示す．このコードが本来想定している動作は，URL のあるパラメータからユーザーの名前を抽出し DOM に動的に追加することで表示するものであるが，適切なエンコードやサニタイズが行われないため，Client-Side XSS の脆弱性を含んでいる．Web ブラウザによっては，location.hash などの URL におけるパラメータを自動でエンコードする機能を有するが，その機能を持たない Microsoft Edge などの Web ブラウザでは，URL のパラメータを

```
<div
    data-dojo-type="dijit/Declaration"
    data-dojo-props="}-alert(1)-{">
</div>
```

図 2　XSS Auditor を回避する Script Gadgets を利用した例 [9]

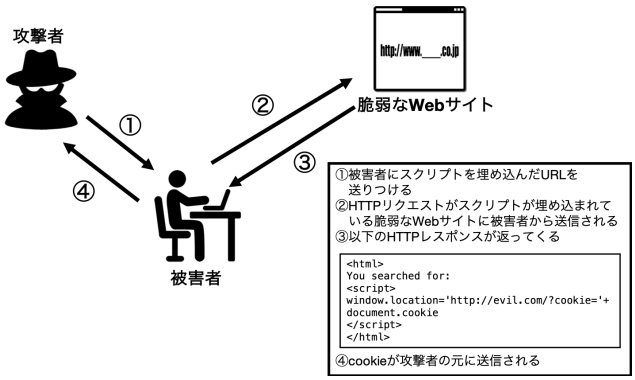

図 3　Client-Side XSS の標準的な攻撃モデル [10]

<script>alert(1)</script> などのようにして攻撃者が被害者に悪意のあるスクリプトを実行させることが可能である．

2.2 サードパーティと従来の XSS 防御策の回避

Client-Side XSS の脆弱性は Web アプリケーションの開発者による JavaScript コードだけでなく，サードパーティの JavaScript ライブラリにも含まれている．Stock らの調査 [8] では，Alexa Top 10000 のドメイン上に存在した 1,273 個の Client-Side XSS 脆弱性のデータセットの分析を行い，内 273 個はサードパーティのライブラリが原因であったと示している．加えて，Lekies ら [9] は，Script Gadgets と呼ばれるコードブロックを使用することで，サードパーティライブラリに含まれる脆弱性を用いて Client-Side XSS が可能だと示しており，この Script Gadgets を利用することで，従来の XSS 防御策である HTML サニタイザや XSS フィルタ，CSP などを回避することが可能だと示している．

図 2 に XSS Auditor を回避する Dojo Toolkit に存在する Script Gadgets を利用したコードの例 [9] を示す．この例では，Dojo Toolkit という広く使われている JavaScript ライブラリに存在する Script Gadgets を利用することで，Web ブラウザである Google Chrome などに採用されている XSS フィルタの XSS Auditor を回避し，Client-Side XSS を行うことが可能である．この例のように，Client-Side XSS では攻撃に利用される入力が複雑なことが多く，従来の XSS 対策では検知や防御が難しいため，従来のものとは別に Client-Side XSS に向けた防御策が必要だとされている．

2.3 攻撃モデル

図 3 に Client-Side XSS の標準的な攻撃モデル [10] を示す．Client-Side XSS は，攻撃者があらかじめ Client-Side XSS の脆弱性を含んだ Web サイトの情報を知った上で，

(1) 被害者にスクリプトを埋め込んだ URL を送りつける
(2) スクリプトが埋め込まれた HTTP リクエストが脆弱な Web サイトに送信される
(3) HTTP レスポンスが返信される
(4) Web ブラウザでスクリプトが実行され，被害者の Cookie が攻撃者のサーバに送信される

といった流れで攻撃が行われる．

従来の XSS 防御策では，(1) 以前にサーバ側でスクリプトの埋め込みを防ぐことに重点を置いていたが，Client-Side XSS では，埋め込まれたスクリプトコードのログがサーバに残らない場合もあり，サーバ側に施す従来の XSS 防御策では対応が難しく，被害者であるクライアント側にて防御することが求められている．

3. 関連研究

Client-Side XSS に対する検知・防御策に関する研究について紹介する．ここでは，Client-Side XSS の直接的な要因となり得る，文字列の操作や型に焦点を当てたものをまとめる．

3.1 テイントトラッキングによる検知・防御

Stock ら [4] の研究では、JavaScript で取り扱う文字列全てにテイントを付加し，そのテイントを追跡することで Client-Side XSS の検知・防御を行う手法が提案されている．この手法では，JavaScript エンジンによる JavaScript ソースの構文解析の間に，全ての文字列にビットであるテイントを付加させ，文字列の操作や結合があった場合でも，そのテイントが伝播する仕組みとなっているため，ある文字列が攻撃者が制御可能とする不正な起源としたものか否か判断が可能となる．したがって，定義されている実行シンクに文字列が格納される場合に，不正な起源かを検知し，実行を終了させることが可能な上，テストデータを意図的に挿入することで実行シンクに到達するかどうかで，不正なデータフローが Web アプリケーションに存在するか確認が可能である．

一方で，本来存在しない機能であるテイントトラッキングが文字列の定義や操作の度に動作するため，通常の処理速度に比べて 7〜17%のオーバーヘッドが生じることや，テイントが付加された文字列を，JSON.parse() が許容せず例外処理が必要などの課題が挙げられている．

3.2 安全なコードに置換するパッチを自動生成することによる防御策

Parameshwaran ら [5] の研究では，安全でないコードを安全なコードに置換するパッチを自動生成することによる Client-Side XSS の防御策が提案されている．この手法では，Web にアクセスする際はプロキシサーバを経由させ，そのプロキシサーバにおいてユーザがアクセスしようとした Web サイトを，テイントトラッキングを利用して Client-Side XSS の脆弱性分析を行い，脆弱性を含んでいるコードと，そのコードを安全なコードに置換したものを生成してデータベースに保存する．そしてデータベースをもとにパッチを生成し，プロキシサーバに保存している Web サイトのファイルに適用させ，ユーザは安全なプロキシサーバ上のファイルを閲覧する．この手法により，ユーザが利用するブラウザに依存せず，Client-Side XSS の脆弱性を塞ぐことを可能とする．

一方で，テイントトラッキングによる脆弱性分析とパッチを適用させる処理のオーバーヘッドが，どのブラウザでも 5%以上生じることや，プロキシサーバやデータベースを配置しなければならず，導入が容易でないことが課題として挙げられる．

```
const myPolicy = TrustedTypes.createPolicy("mypolicy", {
    createHTML: (unsafe) => {
        return unsafe
            .replace(/</g, "\&lt;")
            .replace(/>/g, "\&gt;")
    }
})
let str = "unsafe";
let trusted = myPolicy.createHTML(str);

document.body.innerHTML = str; // error
document.body.innerHTML = trusted; // success
```

図 4 Trusted Types の例

3.3 Trusted Types による Client-Side XSS の予防

Kotowicz ら [6] は，実行シンクに渡される文字列が安全な文字列であるかを型で検証する，Trusted Types の提案を行っている．この Trusted Types では，対応するコンテンツセキュリティポリシー (以下 CSP) を設定した場合，innerHTML や document.write などの実行シンクに通常の文字列が格納できなくなり，Trusted Types と呼ばれるオブジェクトのみ格納を許可する仕組みとなっている．

図 4 に Trusted Types の例を示す．既に Trusted Types に対応する CSP の設定を行っている場合，実行シンクに通常の文字列が格納できなくなる．したがって，11 行目で通常の文字列を innerHTML に渡しているが，エラーが発生し，DOM は更新されない．Web アプリケーションの開発者が，何らかの理由で実行シンクに文字列を渡して DOM を更新する場合は，1 行目のように開発者が自身でポリシーを作成・設定し，そのポリシーに基づいて文字列

を Trusted Types と呼ばれるオブジェクトにすることで，DOM の更新が可能になる．

この Trusted Types を適用させることで、Client-Side XSS の脆弱性が生じる可能性を限りなく減らすことが出来るとされているほか、防御する際も，文字列の型を検証するだけでよく，パフォーマンスへの影響も少ない手法となっている．

この Trusted Types の課題として，Web アプリケーションの開発方法に手順が追加されるため，開発者への負担が大きくなることと，あくまでも Trusted Types は Web アプリケーション開発時の手法として定義されているため，既存の Web アプリケーションに存在する脆弱性を利用した Client-Side XSS を防ぐことが出来ないことである．仮に Trusted Types を既存の Web アプリケーションに適用させる場合でも，サードパーティのライブラリを含むコードの変更が必要で，近年の大規模な Web アプリケーションの変更は現実的でない点が大きな課題となっている．

以上より，Client-Side XSS の防御手法

従来手法で用いられているテイントトラッキングでは，正確な不正なデータフローを検知・防御可能だが，テイントを保持する空間や，テイントを文字列間で伝播させる処理が追加されるため，パフォーマンスが低下する可能性がある．

4. 提案手法

本研究では，パフォーマンスと既存の Web アプリケーションへの影響を小さくした Client-Side XSS の防御を目的としている．そのため，関連研究で用いられているテイントトラッキングは，文字列に安全であるか示す情報となるテイントを付加することで，不正なデータフローを正確に検知し防御可能だが，テイントを保持する空間や，テイントを文字列間で伝播させる処理が追加されるため，パフォーマンスが低下する恐れがあり，不向きだと考えられる．一方 Trusted Types は，安全であるかは文字列の型で検証することで，パフォーマンスへの影響を小さく防御が可能だが，開発者が明示的にポリシーと信頼できるオブジェクトを生成しなければならないため，既存の Web アプリケーションに適用させるには大幅な変更が必要であり，大きな課題となっていた．

そこで本研究では，この Trusted Types をプリミティブな型として定義することで，既存の Web アプリケーションに変更を加えることなく，Trusted Types を適用させる手法を提案する．そして Trusted Types を利用することで，安全であるか示す情報は型で保持させ，Client-Side XSS の防御に必要な処理は文字列の型を検証するのみとし，パフォーマンスへの影響も最小限に抑える．

以降では，提案手法の概要，提案手法によるユースケースのカバー，および実装と評価について述べる．

```
let str = location.hash;                 // String
let trusted = "https://example.co.jp/"; // Trusted Types

document.body.innerHTML = str; // error
document.body.innerHTML = trusted; // success
```

図 5　プリミティブな Trusted Types の例

4.1 Trusted Types をプリミティブな型として定義

プリミティブな型とは，オブジェクトでなく，メソッドを持たないデータのことを指し，JavaScript では現在，Boolean, Null, Undefined, Number, BigInt, String, Symbol の 7 種類のプリミティブな型が定義されている．そして，JavaScript ではデータ同士の演算はこのプリミティブな型同士でのみ許可されており，演算子のオーバーロードは許可されていない．オブジェクト同士で演算を行う場合は，一度このプリミティブな型に変換された上で計算が行われる．したがって，Trusted Types をプリミティブな型として定義することで，Web アプリケーションの開発方法で用いられる＋演算子による文字列同士の結合などが，信頼できるかどうかの情報を保持したまま行うことができ，既存の Web アプリケーション上でも動作させることが可能になる．

4.2 Trusted Types の割り当て

プリミティブな Trusted Types の割り当ては，JavaScript エンジンによる JavaScript ソースの解析の間に行う．JavaScript エンジンの動作の大まかな順序は
(1) JavaScript ソースの構文解析
(2) 抽象構文木の生成
(3) コンパイルにより実行する機械語の生成
となる．本研究では，他のプリミティブな型と同様に，構文解析の段階で Trusted Types の割り当てを行う．構文解析の段階で割り当てを行うことで，従来のように開発者が明示的に Trusted Types の割り当てを行う必要がなくなるほか，あらかじめ型を割り当てるため，従来の String である文字列を Trusted Types にする処理自体も減らすことが可能になる．

図 5 にプリミティブな Trusted Types の例を示す．本研究では，Trusted Types を割り当てる文字列は，JavaScript ソース中の開発者が記述した文字列のみとする．具体的には，JavaScript 中のシングルクォーテーションまたは，ダブルクオーテーションで囲まれた文字列を Trusted Types としてプリミティブな型を割り当てる．したがって，URL などのグローバルオブジェクトに格納されている文字列は，通常の String を割り当てることとなる．そして，従来のものと同様に動的に DOM を更新する実行シンクには Trusted Types のみを許可し，通常の String は許可しない．そのため，図 5 では 4 行目にて innerHTML に渡され

```
let name = "Your name is ";
document.writeln(name); // success

name += location.hash;
document.writeln(name); // error
```

図 6　String と Trusted Types が結合する例

```
let trusted = " Excepteur sint occaecat";
trusted = trusted.trim();
trusted = trusted.substr(0, 9);

document.body.outrHTML = trusted; // success
```

図 7　ラッパーオブジェクトのメソッドを使用する例

る文字列が，グローバルオブジェクトである location.hash を起源とする String のため，格納が許可されない．また，渡される文字列を信頼できるもののみに制限する実行シンクの種類は，従来の Trusted Types に準拠する．

4.3　ユースケースのカバー

プリミティブな Trusted Types を新たに定義することで，JavaScript エンジン中に文字列を扱う型が複数になるなど変更が加わるため，従来のユースケースをカバーする必要が生じる．ここでは代表的なものを例に挙げる．

4.3.1　String と Trusted Types が結合される場合

上述した通り，JavaScript ソース内では，＋演算子による文字列の結合が頻繁に用いられている．提案手法によりプリミティブな Trusted Types を定義した場合，文字列の型が 2 種類存在するため，これら同士が結合される可能性がある．

図 6 に String と Trusted Types が結合する例を示す．この例では，1 行目で Trusted Types の変数が定義され，4 行目でその変数にグローバルオブジェクトである location.hash が結合されている．この例のように Trusted Types と String が結合された場合，結果となる文字列は String とし，実行シンクへの格納は許可しない．

4.3.2　ラッパーオブジェクトのメソッドを使用する場合

プリミティブな型にはメソッドが存在しないが，JavaScript における文字列は，対応するラッパーオブジェクトである String オブジェクトに自動変換させることでメソッドを呼び出すことが可能である．

図 7 にラッパーオブジェクトのメソッドを使用する例を示す．従来の文字列と同様に，Trusted Types にも対応するラッパーオブジェクトを定義し，String オブジェクトと同様のメソッドやプロパティを利用可能とする．これにより，既存の Web アプリケーションに影響を及ぼさないようにするほか，開発者が String との差を意識しないようにする．そのため，定義する Trusted Types のラッパーオブジェクトのメソッドの返却値は基本的に Trusted Types とし，図 7 の例のように実行シンクに格納することを許可する．例外として，concat() などのように，文字列の引数が指定可能でその引数が String であった場合，返却値は String とする．

```
<script src="jquery.min.js"></script>

<script>
let trusted = "Lorem ipsum dolor sit amet";
let string = location.search;

$("#div").innerHTML = trusted; // success
$("#div").innerHTML = string;  // error
</script>
```

図 8　サードパーティのライブラリを利用する例

4.3.3　サードパーティのライブラリを利用する場合

Client-Side XSS の脆弱性は，サードパーティのライブラリが原因となるものも多く存在することが知られている．そのため，サードパーティのライブラリを利用している場合でも，Trusted Types を適用させる．

図 8 にサードパーティのライブラリを利用する例を示す．この例のように，サードパーティのライブラリが利用されている場合でも，最終的に実行シンクに文字列が格納される場合，文字列の型を検証することで防御を行う．プリミティブな型として定義しているため，サードパーティのライブラリにも変更を加えず Trusted Types を適用させることが可能となる．

4.4　提案手法の実装

提案手法の実装は，実行シンクに渡される文字列は Trusted Types のみを許可するものと，プリミティブな Trusted Types の定義・割り当ての 2 つを分けて行う．

実行シンクへの制限は，従来の Trusted Types の JavaScript ファイルを変更し，CSP の設定無しで実行シンクは Trusted Types のみを許可する設定とする．また，従来のものは実行シンクの種類によって許可するオブジェクトは TrustedHTML や TrustedScript などに分けられているが，本研究では Trusted Types の 1 つのみを許可するオブジェクトとして定義する．この JavaScript ファイルをブラウザでデフォルトで読み込ませる実行シンクへの制限を起動させる．

Trusted Types の定義と割り当ては，オープンソースの Web ブラウザ、JavaScript エンジンに実装し，現在 Chromium と V8 に実装を進めている．定義の実装は，既存の Web アプリケーションにおける文字列で用いられるメソッドなどを動作させるため，ラッパーオブジェクトも定義する．型の割り当ての実装として，Trusted Types の割り当てで述べたとおり，JavaScript ソースの構文解析の段

階で割り当てる．また，Web ブラウザによる JavaScript エンジンの呼び出しに変更を加えることで，Web ブラウザによる JavaScript の初期化と，開発者が記述した JavaScript ソースの解析処理を分け，JavaScript ソース中の文字列にのみ Trusted Types を割り当てる．

4.5 評価と実験

提案手法の評価項目には，

- Client-Side XSS のエクスプロイトに対する防御率
- 通常の Web アクセスに対する偽陽性率
- JavaScript のパフォーマンスの変化

の 3 つを考えており，JavaScript パフォーマンスの変化は，Web の表示速度と JavaScript エンジンによる処理速度を評価する．

したがって実験は，Client-Side XSS のエクスプロイトのデータセットを準備し，防御率を計測するほか，UI テストツールである Selenium[11] と，JavaScript ベンチマークツールである SunSpider[12] を利用し，偽陽性率とパフォーマンスを計測し，提案手法を実装していない Web ブラウザや関連研究との比較を考えている．

5. おわりに

近年，Web におけるクライアント側の機能が増加するにつれて，JavaScript の安全でない記述によりクライアント側で発生する Client-Side XSS の脆弱性が問題となっている．しかし従来の Client-Side XSS 防御手法では，Web のパフォーマンスに影響が発生する，または既存の Web アプリケーションに適用が困難など課題が多かった．

そこで，本研究では，パフォーマンスと既存の Web アプリケーションへの適用性を考慮した Client-Side XSS の防御を目的とし，安全な文字列であるかを型で検証する Trusted Types を，プリミティブな型として定義することにより，既存の Web アプリケーションに変更を加えずに Trusted Types を適用し，Client-Side XSS を防御する手法を提案した．

今後の課題として，提案手法の実装を行い，Client-Side XSS のエクスプロイトのデータセットと UI テストツールなどを用いた実験にて防御率と偽陽性率，パフォーマンスを定量的に評価し，考察を行う．

参考文献

[1] CERT: *Advisory CA-2000-02 Malicious HTML Tags Embedded in Client Web Requests*, 入手先 ⟨https://www.rigacci.org/docs/biblio/online/CA-2000-02/CA-2000-02.html⟩ (2019.10.28)

[2] Sebastian Lekies, Ben Stock, and Martin Johns: *25 million flows later: large-scale detection of dom-based xss*, In Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security, pp. 1193-1204, 2013

[3] William Melicher, Anupam Das, Mashmood Sharif, Lujo Bauer, and Limin Jia: *Riding out DOMsday: Toward Detecting and Preventing DOM Cross-Site Scripting*, The Network and Distributed System Security Symposium, 2018

[4] Ben Stock, Sebastian Lekies, Tobias Mueller, Patrick Spiegel, Martin Johns: *Precise Client-side Protection against DOM-based Cross-Site Scripting*, 23rd USENIX Security Symposium, 2014

[5] Inian Parameshwaran, Enrico Budianto, Shweta Shinde: *Auto-Patching DOM-based XSS At Scale*, Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, pp. 272-283, 2015.

[6] Krzysztof Kotowicz, Mike West, *Trusted Types*, 入手先 ⟨https://w3c.github.io/webappsec-trusted-types/dist/spec/⟩, (2019.10.28).

[7] Amit Klein: *DOM based cross site scripting or XSS of the third kind*, Web Application Security Consortium, Articles, 2005.

[8] Ben Stock, Stephan Pfistner, Bernd Kaiser, Sebastian Lekies, and Martin Johns: *From Facepalm to Brain Bender: Exploring Client-Side Cross-Site Scripting*, ACM Conference on Computer and Communications Security, 2015.

[9] Sebastian Lekies, Krzysztof Kotowicz, Samuel Groß, Eduardo A. Vela Nava, Martin Johns: *Code-Reuse Attacks for the Web: Breaking Cross-Site Scripting Mitigations via Script Gadgets*, The ACM Conference on Computer and Communications Security, 2017.

[10] Upasana Sarmaha, D.K. Bhattacharyyaa, J.K. Kalitab: *A survey of detection methods for XSS attacks*, Journal of Network and Computer Applications, pp. 113-143, 2018.

[11] SeleniumHQ: *Selenium - Web Browser Automation*, 入手先 ⟨https://www.seleniumhq.org/⟩, (2019.10.28).

[12] Filip Pizlo: *Announcing sunspider 1.0.*, 入手先 ⟨https://www.webkit.org/blog/2364/announcing-sunspider-1-0/⟩, (2019.10.28).