Trusted Typesを用いたデータフローの解析による DOM-Based XSS脆弱性検知手法の提案

情18-41　井手 脩太

目 次

図 目 次

表 目 次

第1章　序論

1.1　研究背景

現在，Web はコミュニケーション，電子商取引，動画視聴など様々な用途で利用され，社会インフラのひとつであるといえるほど重要なものになっている．一方で，Web の果たす役割が大きくなるとともに，Web アプリケーションに存在する脆弱性が悪用された場合のリスクも大きくなっている．

Web アプリケーション上で発生しうる脆弱性のひとつに，Cross-Site Scripting（以下，XSS）がある．XSS は，ユーザが入力したデータを適切に検証しないまま出力するといった原因によって，攻撃者によって注入された攻撃コードがユーザの Web ブラウザ上で実行されるという脆弱性である．情報処理推進機構によれば，2021 年第 4 四半期において，脆弱性対策情報データベースに登録された 4676 件のうち，XSS は 501 件で最も多い [1].

XSS の中でも特にクライアント側の処理に起因するものを DOM-Based XSS と呼ぶ．クライアント側の担う処理が増え，コードが大規模化する傾向にある中，DOM-Based XSS の有効な検知手法や防御手法が求められている．

1.2　研究目的

本研究では，どのようなユーザ入力がどのような API に流れたかを意味するデータフローに着目する．Web サイト上に存在するデータフローの情報を収集し，出力することで開発者に修正のヒントを与え，また DOM-Based XSS の防止を容易にすることを目的として，DOM-Based XSS を自動で検知するシステムを提案する．

Trusted Types と呼ばれるセキュリティ機構を利用して危険なデータフローの情報を収集することで，Web ブラウザや JavaScript エンジンのソースコードに変更を加えることなく，DOM-Based XSS を検知できるシステムを目指す．

第2章　技術解説

2.1　Cross-Site Scripting

Cross-Site Scripting（XSS）は，Web アプリケーションにおいて，ユーザが入力した文字列を検証や無害化といった処理を行わず出力に含めることによって，攻撃者によって注入された悪意のある JavaScript コードの実行を可能とする脆弱性である．

ソフトウェアやハードウェアに存在しうる脆弱性を分類した Common Weakness Enumeration（CWE）では，XSS は CWE-79[2] として定義されている．CWE-79 では，XSS の種類として，その原因から Reflected XSS，Stored XSS，DOM-Based XSS の 3 種類に細分化されている．

2.1.1　Reflected XSS

Reflected XSS は，XSS でも特に，GET パラメータやフォームの入力データといったユーザが送信した HTTP リクエストに含まれる危険な入力を，Web アプリケーションが HTTP レスポンス中の HTML に展開することによって発生するものをいう．

Reflected XSS が存在する，PHP で記述された Web アプリケーションの例をソースコード 2.1 に示す．この PHP コードは，`name` というキーの GET パラメータが与えられた場合に，その値をほかの文字列と結合して出力する．

この PHP コードは，ユーザが入力した値を検証や無害化を行わずに出力する．したがって，`http://localhost:8000/reflected.php?name=<script>alert(123)</script>`のように HTML タグを含む文字列を `name` というキーの GET パラメータの値として与えるとそのまま出力され，`alert(123)` という JavaScript コードが実行される．

ソースコード 2.1: Reflected XSS が存在する Web アプリケーションの例

```
1 <?php
2 if (isset($_GET['name'])) {
3     echo 'こんにちは，' . $_GET['name'] . 'さん';
4 }
```

2.1.2 Stored XSS

Stored XSS は，XSS でも特に，データベースに蓄積されたユーザの危険な入力を，Web アプリケーションが HTTP レスポンス中の HTML に展開することによって発生するものをいう．XSS が発生する原因がサーバ側にあるという点では Reflected XSS と共通しているが，Stored XSS では攻撃コードが持続的に出力されうる点で異なる．

Stored XSS が存在する，PHP で記述された Web アプリケーションの例をソースコード 2.2 に示す．この PHP コードは，memo というキーの GET パラメータが与えられた場合に，その値を memo.txt というファイルに保存する．GET パラメータが与えられていない場合には，memo.txt の内容を出力する．

この PHP コードでは，ユーザが入力した値である GET パラメータを検証や無害化を行わずに保存しているため，memo の値として HTML タグが含まれていたとしても，memo.txt にその内容がそのまま保存される．memo.txt の内容を出力する際にもその内容の検証や無害化は行われていないため，その後にアクセスしたユーザには，memo というキーの GET パラメータが与えられていなかったとしても，memo.txt に保存された HTML タグがそのまま出力される．

そのため，http://localhost:8000/stored.php?memo=<script>alert(123)</script>のように script タグを含む文字列を memo というキーの GET パラメータの値として与えると，その後にアクセスしたユーザの Web ブラウザ上では alert(123) という JavaScript コードが実行される．

ソースコード 2.2: Stored XSS が存在する Web アプリケーションの例

```
1 <?php
2 if (isset($_GET['memo'])) {
3     file_put_contents('memo.txt', $_GET['memo']);
4     echo 'メモを書き込みました．';
5     exit;
6 }
7
8 echo file_get_contents('memo.txt');
```

2.1.3 DOM-Based XSS

DOM-Based XSS は，XSS でも特にクライアント側の処理に起因するものをいう．Web ブラウザが提供する API の中には，JavaScript によって Web ページの構造を変更できるものも含ま

れる．ユーザが入力した文字列を不適切にこのような API に与えることによって，DOM-Based XSS が発生する．

DOM-Based XSS が存在する HTML の例をソースコード 2.3 に示す．この HTML では，9 行目から 12 行目の `script` 要素に含まれる JavaScript コードによって，8 行目に存在する `span` 要素の内容が書き換えられる．11 行目では，`location.hash` という API を利用することで，URL の「`#`」以降の文字列を取得する．この際，`decodeURIComponent` という関数を呼び出すことで，文字列に含まれる`%3C` のようなパーセントエンコーディングされた箇所をデコードする．`span.innerHTML` というプロパティにその値を代入することで，`span` 要素の内容を，URL の「`#`」以降の文字列を HTML として解釈したものに置き換える．

URL の「`#`」以降の文字列は，ユーザによってその内容を自由に変更することができるため，`http://localhost:8000/dom-based-xss.html#%3Cimg%20src=x%20onerror=alert(123)%3E` のように細工した HTML を含めることで，`alert(123)` という JavaScript コードが実行される．

ソースコード 2.3: DOM-Based XSS が存在する Web アプリケーションの例

```
1  <!doctype html>
2  <html lang="en">
3      <head>
4          <meta charset="utf-8">
5          <title>Vulnerable Page</title>
6      </head>
7      <body>
8          <p>Hello, <span id="username">John</span>!</p>
9          <script>
10         const span = document.getElementById('username');
11         span.innerHTML = decodeURIComponent(location.hash.slice(1));
12         </script>
13     </body>
14 </html>
```

表 2.1: CSP において利用可能なディレクティブ名の例

ディレクティブ名	説明
`default-src`	デフォルトの設定を示す．リソースに対応するディレクティブが定義されていなければ，この設定が代わりに利用される．
`script-src`	JavaScript コードの設定を示す．ソースにはドメイン名や `https:` のようなスキームといったリソースに関する条件のほか，`eval` を呼び出せるようにするかといった設定ができる．
`frame-src`	`frame` 要素や `iframe` 要素の設定を示す．

2.2　Content Security Policy

Content Security Policy（CSP）[3] は，Web アプリケーション上で読み込むことが可能なリソースを制限することによって，XSS を始めとした脆弱性の緩和を目指す Web ブラウザのセキュリティ機構である．CSP の仕様は World Wide Web Consortium（W3C）によって標準化されている．

CSP を導入したい Web アプリケーションは，`Content-Security-Policy` と呼ばれる HTTP レスポンスヘッダ（CSP ヘッダ）を通じて，どのようなリソースが読み込み可能であるかを指定するポリシーと呼ばれる文字列を Web ブラウザに提示する．Web ブラウザは，ポリシーが提供されている Web ページにおいて発生した，ポリシーに反するリソースの読み込みをブロックする．

ポリシーは画像，スタイルシート，JavaScript コードといったリソースの種類ごとにどのようなリソースが読み込み可能かを指定する，ディレクティブと呼ばれる要素から構成される．各ディレクティブは，リソースの種類を示すディレクティブ名と，その種類のリソースについてどのようなものの読み込みを許可するかを意味する値からなる．

CSP において利用可能なディレクティブ名の一部を表 2.1 に示す．リソースの種類によって，利用可能なディレクティブの値は異なる．`script-src` ディレクティブにおいて利用可能なディレクティブの値の一部を表 2.2 に示す．

表 2.2: script-src において利用可能な値の例

ディレクティブの値	説明
'none'	JavaScript コードの実行は一切許可しない.
'self'	Web ページと同じオリジンにある JavaScript ファイルの実行を許可する.
'unsafe-eval'	eval 関数による JavaScript コードの実行を許可する.
'unsafe-inline'	script 要素内の JavaScript コードの実行を許可する.

CSP ヘッダの例をソースコード 2.4 に示す．この CSP ヘッダの例で提示されているポリシーには，default-src と script-src の 2 つのディレクティブが含まれている．JavaScript コードの読み込み時には script-src ディレクティブで指定された設定が利用され，Web ページと同じオリジンに存在するリソースでなければ（'self'），読み込みはブロックされる．JavaScript コード以外のリソースの読み込み時には default-src ディレクティブで指定された設定が利用され，すべてのリソースの読み込みがブロックされる（'none'）.

ソースコード 2.4: CSP ヘッダの例

```
1 Content-Security-Policy: default-src 'none'; script-src 'self'
```

2.3 Trusted Types

Trusted Types[4] は，引数を検証しないままに利用すると危険な振る舞いをする，Sink（4.2 節参照）となる API について，検証や無害化といった処理を行う関数を明示的に呼び出し，引数となる文字列が安全であると確認できた場合にのみ利用できるようにする Web ブラウザのセキュリティ機構である.

Trusted Types が利用できる環境では，TrustedHTML，TrustedScript，TrustedScriptURL の 3 つの安全な型が追加されている（表 2.3）．Trusted Types が有効化されている Web ページにおいては，Sink となる API を利用するには，引数が API の種類に対応する安全な型でなければならない．API の種類に対応する型以外のオブジェクトが引数として与えられた場合，例外が発生し API の呼び出しは行われない.

表 2.3: Trusted Types が導入する安全な型の一覧

型名	説明
`TrustedHTML`	引数が HTML として解釈され出力される Sink において利用される.
`TrustedScript`	引数が JavaScript コードとして実行される Sink において利用される.
`TrustedScriptURL`	引数が JavaScript コードを返す URL を意味する Sink において利用される.

文字列を Trusted Types によって追加された 3 つの安全な型に変換するには，ポリシーと呼ばれるオブジェクトのメソッドを呼び出す必要がある．ソースコード 2.5 に示すように，Web ブラウザがグローバル変数として提供している `trustedTypes` と呼ばれるオブジェクトに含まれる，`createPolicy` と呼ばれるメソッドを呼び出すことによって，ポリシーを定義できる．

ソースコード 2.5: Trusted Types のポリシーを定義する例

```
1 const escapeHTML = trustedTypes.createPolicy('escape-html', {
2   createHTML(string) {
3     return string.replaceAll('<', '&lt;').replaceAll('>', '&gt;');
4   }
5 });
```

`createPolicy` は，第 1 引数としてポリシー名を，第 2 引数として文字列を安全な型に変換する際に呼び出される関数を含むオブジェクトを取る．第 2 引数のオブジェクトのプロパティとして，入力の検証や無害化を行うような関数を与えることで，Sink となる API を利用する際に，Sink に与える引数を一度それらの関数を通さなければならないということになる．

`createPolicy` の第 2 引数として与えるオブジェクトが持つ関数のプロパティ名は，表 2.4 に示すようにその返り値として得たい型に応じて選択する必要がある．例えば，文字列を `TrustedHTML` 型に変換したい場合には，`createHTML` という名前のプロパティを持たせる必要がある．

表 2.4: ポリシーが持つメソッドの名前と返り値の型の一覧

メソッド名	返り値の型
createHTML	TrustedHTML
createScript	TrustedScript
createScriptURL	TrustedScriptURL

このように，明示的に入力を検証もしくは無害化しなければならないようにできる仕組みの導入によって，Trusted Types は DOM-Based XSS の防止を図る．Trusted Types は，CSP ヘッダによって有効化し，また JavaScript コード側で Trusted Types に対応する処理を追加することで導入できる．CSP ヘッダでは，require-trusted-types-for ディレクティブによって Trusted Types を有効化し，trusted-types ディレクティブによって信頼できる Trusted Types のポリシー名を提示する．JavaScript コード側では，CSP ヘッダで提示した名前を持つポリシーを定義し，Sink の呼び出し時に，ポリシーを使って引数を安全な型に変換するよう変更を加えることで対応する．

ソースコード 2.6 は，Trusted Types を導入し，DOM-Based XSS の防止を図った例である．6〜7 行目において，meta 要素を利用して CSP ヘッダを提示し，escape-html という名前の Trusted Types のポリシーを信頼できるものとする．

ソースコード 2.6: Trusted Types を利用する例

```
1  <!doctype html>
2  <html lang="en">
3    <head>
4      <meta charset="utf-8">
5      <title>Trusted Types</title>
6      <meta http-equiv="Content-Security-Policy"
7            content="require-trusted-types-for␣'script';␣trusted-types␣escape
                 -html">
8    </head>
9    <body>
10     <div id="content"></div>
11     <script>
12       const escapeHTML = trustedTypes.createPolicy('escape-html', {
13         createHTML(string) {
```

```
14            return string.replaceAll('<', '&lt;').replaceAll('>', '&gt;');
15          }
16        });
17
18        const div = document.getElementById('content');
19
20        const dangerousHtml = '<img src="x" onerror="alert(123)">';
21        try {
22          div.innerHTML = dangerousHtml;
23        } catch (e) {
24          console.log('[error]', e);
25        }
26
27        const safeHtml = escapeHTML.createHTML(dangerousHtml);
28        div.innerHTML = safeHtml;
29      </script>
30    </body>
31  </html>
```

12〜16 行目で escape-html という名前のポリシーを定義している．第 2 引数として「<」や「>」を無害な文字列に置換する関数を持つオブジェクトを与えることで，この関数によって文字列を無害化し，引数が HTML として解釈され出力される Sink を呼び出せるようになる．

20〜25 行目では，定義されたポリシーを利用して文字列の安全な型への変換を行わないままに，Element.prototype.innerHTML の呼び出しを試みる．Trusted Types はこのような引数の安全な型への変換を伴わない Sink の呼び出しをブロックする．Google Chrome の開発者ツールに含まれる JavaScript コンソールのログを開くと，図 2.1 に示すように，この Sink を呼び出すためには引数の型を TrustedHTML に変換する必要があるというエラーメッセージが出力されていることがわかる．

27〜28 行目では，定義されたポリシーを利用して文字列の安全な型への変換を行った上で，Element.prototype.innerHTML の呼び出しを試みる．Sink の種類に対応する安全な型が引数として渡っていることから，Trusted Types はこのような Sink の呼び出しを許可する．Google Chrome の開発者ツールを用いて Web ページの DOM ツリーを確認すると，図 2.2 に示すように，div 要素に含まれるテキストが変更されていることがわかる．また，そのテキストは 20 行目で定義された文字列に含まれる「<」と「>」が，それぞれ「<」と「>」に置換され無害化されたものであることもわかる．

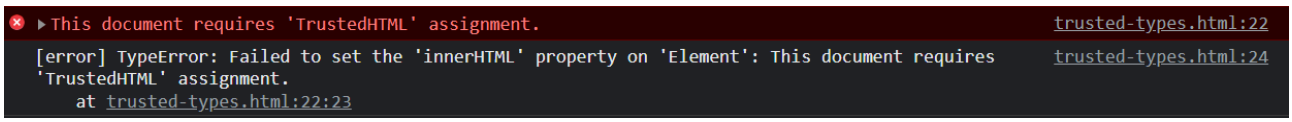

図 2.1: Trusted Types により Sink の呼び出しがブロックされる例

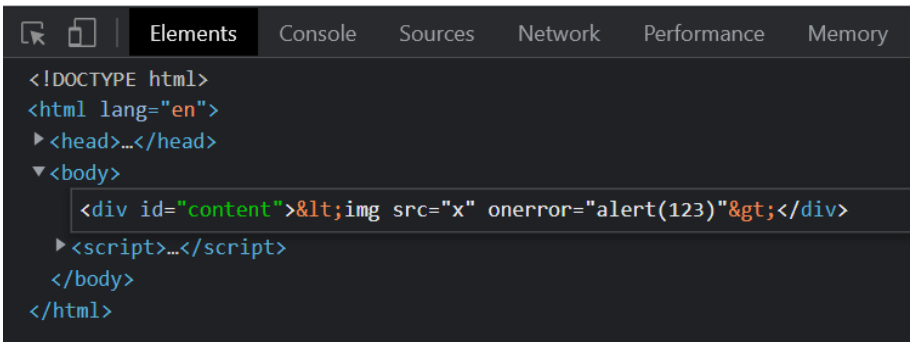

図 2.2: Trusted Types により Sink の呼び出しが許可される例

2.3.1　デフォルトポリシー

デフォルトポリシーは，明示的にポリシーを使って引数を安全な型に変換せずとも，特殊なポリシーを定義することで自動的に引数の検証や無害化を行い，安全な型に変換して Sink（4.2 節参照）となる API を利用できるようにする Trusted Types の機能である．デフォルトポリシーは，CSP ヘッダで信頼できるポリシー名として `default` を提示し，また JavaScript コード側で `default` という名前のポリシーを定義することで有効化される．

デフォルトポリシーが有効化されると，Sink となる API が呼び出された際に与えられた引数が安全な型でなかった場合に，定義したデフォルトポリシーを通して自動的に安全な型への変換が行われるようになる．このときの安全な型への変換は，ポリシーの定義時に `createPolicy` メソッドへ第 2 引数として与えたオブジェクトに含まれる，Sink の種類に対応する関数によって行われる．

デフォルトポリシーを導入した例をソースコード 2.7 に示す．まず 6〜7 行目において，`meta` 要素を利用して CSP ヘッダを提示し，`default` という名前の Trusted Types のポリシーを信頼できるものとしている．さらに，11〜15 行目で `default` という名前のポリシーを定義することによって，デフォルトポリシーを有効化している．

17 行目では，明示的にポリシーを通して引数の安全な型への変換を行うことなく，`document.body.innerHTML` を呼び出そうとしている．このとき，デフォルトポリシーが有効化されているため，ポリシーの定義時に与えた関数が自動的に呼び出され，引数は安全な型に変換される．

ソースコード 2.7: デフォルトポリシーを利用する例

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Default Policy</title>
    <meta http-equiv="Content-Security-Policy"
          content="require-trusted-types-for 'script'; trusted-types
              default">
  </head>
  <body>
    <script>
      trustedTypes.createPolicy('default', {
        createHTML(string) {
          return string.replaceAll('<', '&lt;').replaceAll('>', '&gt;');
        }
      });

      document.body.innerHTML = '<img src="x" onerror="alert(123)">';
    </script>
  </body>
</html>
```

2.4　Node.js

Node.js[5] は，サーバサイドでの JavaScript コードの実行を可能とする，JavaScript のランタイム環境である．Windows，macOS，Linux を含む様々なプラットフォームに対応しており，環境に依存した処理を利用しない限り，複数の異なるプラットフォーム間で共通の JavaScript コードが利用できるという特徴を持っている．

Node.js では，ネットワークやファイルの取り扱いなど，様々な機能を持つ標準ライブラリが提供されている．このほか，npm[6] と呼ばれるパッケージ管理システムを利用することで，ユーザによって公開されているライブラリを容易にインストールし，自身のプログラムにおいて利用できる．リスト 2.1 に npm を使ってライブラリをインストールするコマンドの例を示す．npm install に続けてライブラリの名前を入力することで指定したライブラリをインストールできる．この例では，typescript というライブラリをインストールしている．

リスト 2.1: npm を使ってライブラリをインストールする例

```
$ npm install typescript

added 1 package, and audited 2 packages in 2s

found 0 vulnerabilities
```

npm を使ってインストールしたライブラリに依存している JavaScript コードを別の環境で実行したい時には，その環境で再度ライブラリをインストールする必要がある．このような場合には，JavaScript コードが依存しているライブラリのリストや，そのコードのバージョンといった情報が含まれる package.json と呼ばれる JSON ファイルを利用することで，容易に実行環境が整えられる．

package.json の例をソースコード 2.8 に示す．この JSON にはコードの名前を意味する name，コードのバージョンを意味する version，コードが依存するライブラリのリストを意味する dependencies の 3 つのプロパティが含まれている．この例では，typescript と呼ばれるライブラリに依存しているという情報が含まれている．^4.5.5 という文字列はライブラリのバージョンを意味しており，package.json を利用したライブラリのインストール時に，そのバージョンと互換性を保っている範囲で可能な限り最新のバージョンを選択することを意味する．

ソースコード 2.8: package.json の例

```
{
  "name": "test",
  "version": "1.0.0",
  "dependencies": {
    "typescript": "^4.5.5"
  }
}
```

リスト 2.2 に package.json を使ってライブラリをインストールするコマンドの例を示す．npm install というコマンドにライブラリ名を与えず実行すると，npm は package.json から依存するライブラリの情報を取得して自動でインストールする．

リスト 2.2: package.json を使ってライブラリをインストールする例

```
$ ls
package.json
```

```
$ npm install

added 1 package, and audited 2 packages in 2s

found 0 vulnerabilities
```

2.5　Playwright

Playwright[7] は，Microsoft によって開発されている，Node.js による Web ブラウザの操作や情報の取得を支援するフレームワークである．Playwright は Chromium や Firefox といった複数の Web ブラウザに対応している．どの Web ブラウザにおいても Playwright は共通の API を提供しているため，操作や情報の取得の対象となる Web ブラウザが変わっても共通のコードを利用できるという特徴を持つ．

Playwright を用いて Chromium の操作や情報の取得を行う例をソースコード 2.9 に示す．

ソースコード 2.9: Playwright を用いるコードの例

```
const { chromium } = require('playwright');

(async () => {
  const browser = await chromium.launch();
  const page = await browser.newPage();
  await page.goto('https://example.com/');

  const result = await page.evaluate(() => {
    return {
      title: document.title,
      location: location.href
    };
  });
  console.log(result);

  await browser.close();
})();
```

4 行目で Chromium を起動し，起動した Chromium に 5 行目で新しいタブを作成させる．6 行目で，作成したタブ上で `https://example.com/` という URL の Web ページにアクセスさせる．

8〜13 行目では，Chromium に，開いた Web ページ上で JavaScript コードを実行させ，そ

の返り値を取得している．返り値はオブジェクトであり，title と location という 2 つのプロパティを持つ．title プロパティの値である document.title は，その Web ページのタイトルを意味する．location プロパティの値である location.href は，その Web ページの URL を意味する．14 行目において，8〜13 行目で実行した JavaScript コードの返り値を出力している．

リスト 2.3 に示すコマンドによって，ソースコード 2.9 に示した JavaScript コードを実行する．出力されたオブジェクトを確認すると，title というプロパティには Example Domain という Web ページのタイトルが含まれていることがわかる．また，location というプロパティには https://example.com/という URL が含まれていることがわかる．

リスト 2.3: Playwright を用いるコードを実行するコマンド

```
1 $ node playwright.js
2 { title: 'Example Domain', location: 'https://example.com/' }
```

このように，Playwright を用いることで，Web ページ上での JavaScript コードの実行やその返り値の取得ができるほか，Web ページへのアクセス時に送信される HTTP リクエストヘッダの変更，Web サーバから返ってきた HTTP レスポンスの書き換えなど，Web ブラウザの操作や情報の取得が容易に行える．

第3章　関連研究

3.1　Ran Wang らの研究

Ran Wang らの研究 [8] では，テイント伝播を用いて Web サイト上に存在するデータフローを解析し，発見したデータフローを使った攻撃を試みる攻撃シミュレーションによって脆弱性であるかを検証する 2 段階によって，DOM-Based XSS を発見する手法が提案された．

閲覧している Web サイトの URL を意味する `location.href`，ウィンドウ名を意味する `window.name` などの，ユーザによって返す値を制御できる API を Source（4.1 節参照）と呼ぶ．また，引数を JavaScript コードとして実行する関数である `eval`，代入された値を HTML として解釈し Web ページに挿入するプロパティである `innerHTML` など，引数を検証しないままに利用すると危険な振る舞いをする API を Sink（4.2 節参照）と呼ぶ．Source から Sink にデータが渡るような流れをデータフローと呼び，引数を適切に処理していない場合には脆弱性となりうる．

この手法では，WebKit をベースとした Web ブラウザである PhantomJS[9] のソースコードに変更を加えることで，Web サイト上でのテイント伝播の追跡を可能にする．Source となる API が返す文字列に対して，どのユーザ入力に由来しているかを意味するテイントタグが付与されるようにする．テイントタグが付与された文字列に置換や結合といった加工がされれば，新しく作り出された文字列に対してもテイントタグを付与し伝播させる．このようなテイントタグの伝播により，データの流れを追跡できるようにする．Sink となる API に引数としてテイントタグが付与された文字列が渡ってくれば，テイントタグに対応する Source からその Sink へのデータフローが存在していることを意味する．

この手法には，実際に Web サイト上に設置されている JavaScript コードを実行してデータフローの情報を収集していることから，発見されたデータフローは確実に存在しているという特徴がある．一方で，Web ブラウザのソースコードに修正を加えることでデータフローの解析を実現しているため，元となる Web ブラウザ向けに配信されたアップデートを，修正を加

えた Web ブラウザに適用しようとすると，ソースコードの変更箇所の衝突を修正するための作業を必要とする可能性があるという課題がある．また，データフローの解析は実行された範囲にあるものに限られることから，Web ページに存在するすべてのデータフローを網羅することが難しいという課題がある．

3.2　山崎らの研究

山崎らの研究 [10] では，プリミティブな型として Trusted Types を扱う仕組みの導入によって，DOM-Based XSS を防止する手法が提案された．

Trusted Types が有効化されている Web サイト上では，Sink となる API に検証もしくは無害化がされていない文字列（`String`）が引数として渡った際にエラーが発生し，API の呼び出しが中止される．検証もしくは無害化を行って文字列を Trusted Types が提供する安全な型（`TrustedHTML`，`TrustedScript`，`TrustedScriptURL`）に変換することで，ようやく Sink となる API を利用できるようになる．

このような性質を利用するために，すべての Web サイトで明示的に指定せずとも Trusted Types を有効化するよう Web ブラウザに変更を加える．さらに，JavaScript コード中に元から存在する文字列リテラルに Trusted Types の型を割り当てることによって，既存の Web アプリケーションに変更を加えることなく，DOM-Based XSS を防止する．

プリミティブな型として Trusted Types を取り入れることで，既存の Web アプリケーションにおいても，特別な作業を行わず DOM-Based XSS を有効に防止できることを示した．また，Trusted Types が明示的に有効化されていない Web サイトにおいても，強制的に有効化させることで Sink となる API の利用を制御できることを示した．一方で，Web ブラウザに変更を加えることでこのような機能を実現しているため，一般ユーザへの普及が難しいという課題がある．

3.3　Untrusted Types

Untrusted Types[11] は，DOM-Based XSS の発見を目的として，Google Chrome 向けに拡張機能として Filedescriptor によって開発されたツールである．このツールを Web ブラウザ上で有効化すると，解析対象の Web サイトにおいて，Trusted Types のデフォルトポリシーによって Sink となる API にフックし，引数として渡された文字列の監視を始める．このツールの利用者があらかじめ設定した `d0mxss` のような任意の文字列が，Sink となる API に引数として渡されると，Web ブラウザの開発者ツール上にどの Sink が利用されたかという情報が出

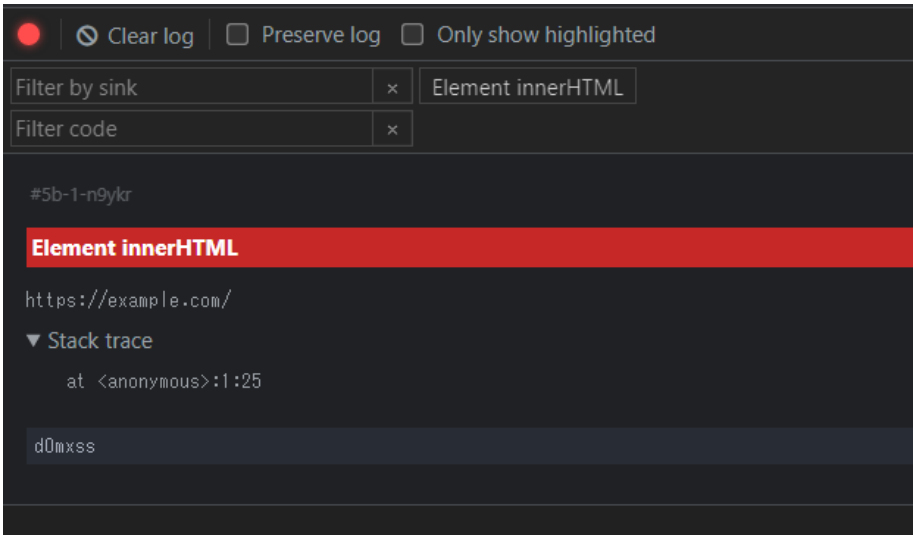

図 3.1: Untrusted Types におけるログの出力例

力される（図 3.1）.

Untrusted Types は Web ブラウザのソースコードの改変によらず，拡張機能として実装されているために，ツールの導入や利用，アンインストールが容易であるという特徴がある．一方で，GET パラメータやフォームへの入力といった操作や，ログの確認による脆弱性があるかの判断はツールの利用者が手作業で行う必要があり，負担が大きいという課題がある．

第4章　諸定義

4.1 Source

Sourceとは，URLに含まれる要素やウィンドウ名など，ユーザによって自由にその値が変更可能であるJavaScriptのAPIを指す．DOM-Based XSSにおいて，本研究で対象とするSourceとなりうるAPIの一覧を表4.1に示す．

表4.1: Sourceとなる APIの一覧

API名	説明
`Cookie`	JavaScriptからは`document.cookie`と呼ばれるプロパティからアクセスできる．
`GET`パラメータ	現在閲覧しているWebページのURLのうち,「`?`」から始まり「`#`」もしくはURLの最後までの区間を意味する．JavaScriptからは`location.search`と呼ばれるプロパティからアクセスできる．
フラグメント識別子	現在閲覧しているWebページのURLのうち,「`#`」以降の区間を意味する．JavaScriptからは`location.hash`と呼ばれるプロパティからアクセスできる．
リファラ	現在閲覧しているWebページの直前に閲覧していたWebページのURLを意味する．JavaScriptからは`document.referrer`と呼ばれるプロパティからアクセスできる．

表は次ページに続く

前ページからの続き

API 名	説明
localStorage	Web ページ上において，Web ブラウザ側で容易にデータを保存できる仕組みである Web Storage API が提供する API のひとつを意味する．localStorage によって保存されたデータは，Web ブラウザを再起動しても保持される．JavaScript からは localStorage と呼ばれるオブジェクトの getItem メソッドにキーを与えるか，localStorage.key や localStorage['key'] のようにキーをプロパティとして与えることでアクセスできる．
sessionStorage	Web ページ上において，Web ブラウザ側で容易にデータを保存できる仕組みである Web Storage API が提供する API のひとつを意味する．sessionStorage によって保存されたデータは，Web ブラウザを閉じると破棄される．JavaScript からは sessionStorage と呼ばれるオブジェクトの getItem メソッドにキーを与えるか，sessionStorage.key や sessionStorage['key'] のようにキーをプロパティとして与えることでアクセスできる．
ウィンドウ名	window.open 関数によってウィンドウが開かれた際などに設定された，ウィンドウの名前を意味する．JavaScript からは window.name と呼ばれるプロパティからアクセスできる．

4.2　Sink

Sink とは，引数として与えられた文字列が JavaScript コードとして実行され，あるいは HTML として解釈され出力されるなど，引数の検証や無害化を行わずに使用すると危険な挙動を示しうる API を指す．DOM-Based XSS において，本研究で対象とする Sink となりうる API の一覧を表 4.2 に示す．

表 4.2: Sink となる API の一覧

API 名	説明
Node.textContent	与えた文字列を，指定した要素内にテキストとして書き込む．書き込まれる対象が script 要素であった場合には，文字列を JavaScript コードとして実行する．
Element.innerHTML	与えた文字列を HTML の断片として解釈し，指定した要素内に書き込む．
Element.outerHTML	与えた文字列を HTML の断片として解釈し，指定した要素を置き換える．
HTMLElement.innerText	与えた文字列を，指定した要素内にテキストとして書き込む．書き込まれる対象が script 要素であった場合には，文字列を JavaScript コードとして実行する．
HTMLElement.outerText	与えた文字列をテキストとして解釈し，指定した要素を置き換える．指定した要素の親要素が script 要素であった場合には，文字列を JavaScript コードとして実行する．
HTMLScriptElement.src	与えた文字列を指定した script 要素の src 属性の値として設定し，読み込んでその内容を JavaScript コードとして実行する．
HTMLScriptElement.text	与えた文字列を指定した script 要素内に書き込み，JavaScript コードとして実行する．
HTMLIFrameElement.srcdoc	与えた文字列を HTML ドキュメントとして解釈し，指定した iframe 要素の内容として設定する．
HTMLIFrameElement.src	与えた文字列を指定した iframe 要素の埋め込み先として設定し，読み込む．
HTMLFrameElement.src	与えた文字列を指定した frame 要素の埋め込み先として設定し，読み込む．

表は次ページに続く

前ページからの続き

API 名	説明
HTMLFormElement.action	与えた文字列を，指定したフォームの送信先として設定する．javascript:から始まる URL であった場合には，フォームの送信時に JavaScript コードを実行する．
HTMLInputElement.formAction	与えた文字列を，指定した input 要素をクリックした場合の送信先として設定する．javascript:から始まる URL であった場合には，フォームの送信時に JavaScript コードを実行する．
HTMLButtonElement.formAction	与えた文字列を，指定した button 要素をクリックした場合の送信先として設定する．javascript:から始まる URL であった場合には，フォームの送信時に JavaScript コードを実行する．
HTMLAnchorElement.href	与えた文字列を，指定した a 要素のリンク先として設定する．javascript:から始まる URL であった場合には，リンクのクリック時に JavaScript コードを実行する．
HTMLObjectElement.data	与えた文字列を object 要素の data 属性の値として設定し，読み込んで表示する．script 要素を含む SVG ファイルが読み込まれた場合には，JavaScript コードが実行される．
location.href	与えた文字列を URL として解釈し，遷移する．javascript:から始まる URL であった場合には，JavaScript コードを実行する．
location.pathname	現在の URL のパス名を与えた文字列に置換する．
location.protocol	現在の URL のプロトコル名を与えた文字列に置換する．
location.host	現在の URL のホスト名とポート番号を与えた文字列に置換する．
location.hostname	現在の URL のホスト名を与えた文字列に置換する．
location.hash	現在の URL の「#」以降を与えた文字列に置換する．

表は次ページに続く

前ページからの続き

API 名	説明
`location.search`	現在の URL のクエリ文字列を与えた文字列に置換する.
`location`	与えた文字列を URL として解釈し，遷移する．`javascript:`から始まる URL であった場合には，JavaScript コードを実行する.
`eval`	与えた文字列を JavaScript コードとして実行する.
`Function`	与えた文字列を関数本体の JavaScript コードとして，関数を生成する.
`setTimeout`	与えた文字列を JavaScript コードとして実行する.
`setInterval`	与えた文字列を JavaScript コードとして実行する.
`setImmediate`	与えた文字列を JavaScript コードとして実行する．現行の Web ブラウザで対応しているものはなく，Internet Explorer のみが対応している.
`fetch`	与えた文字列を URL として解釈し，HTTP リクエストを送信する.
`Document.write`	与えた文字列を HTML の断片として解釈し，ドキュメントに書き込む.
`Document.writeln`	与えた文字列を HTML の断片として解釈し，ドキュメントに書き込む.
`Element.append`	与えた文字列を，指定した要素内にテキストとして追加する．書き込まれる対象が `script` 要素であった場合には，文字列を JavaScript コードとして実行する.
`Element.setAttribute`	与えた文字列を，指定した要素の指定した属性の値として設定する.
`Element.setAttributeNS`	与えた文字列を，指定した要素の指定した属性の値として設定する.
`Element.insertAdjacentHTML`	与えた文字列を HTML の断片として解釈して，指定した要素の指定した位置に書き込む.

表は次ページに続く

前ページからの続き

API 名	説明
`Range.createContextualFragment`	与えた文字列を HTML の断片として解釈して，`DocumentFragment` を作成する．
`location.assign`	与えた文字列を URL として解釈し，遷移する．`javascript:`から始まる URL であった場合には，JavaScript コードを実行する．
`location.replace`	与えた文字列を URL として解釈し，遷移する．`javascript:`から始まる URL であった場合には，JavaScript コードを実行する．
`window.open`	与えた文字列を URL として解釈し，指定したコンテキストで開く．`javascript:`から始まる URL であった場合には，JavaScript コードを実行する．
`XMLHttpRequest.open`	与えた文字列を URL として解釈し，HTTP リクエストの送信を準備する．
`XMLHttpRequest.send`	与えた文字列を HTTP リクエストボディとして設定し，HTTP リクエストを送信する．

第5章　提案システム

5.1　システムの実行環境

本システムは，Ubuntu 20.04 上に JavaScript のランタイム環境である Node.js v16.13.1 をインストールした環境において，JavaScript で書かれたプログラムによって動作する．Node.js における Web ブラウザの操作や情報の取得を支援するフレームワークである Playwright を利用し，データフローの解析と攻撃シミュレーションの 2 段階において，Chromium 97.0.4666.0, Firefox 93.0 の 2 種類の Web ブラウザによる DOM-Based XSS の検知を行う．

システムの実行にあたっては，システムが依存するライブラリを事前にインストールする必要がある．これらのライブラリの名前とバージョンは `package.json` というファイルに JSON 形式で格納されている（p.61 のソースコード A.1）．表 5.1 に，システムが依存しているライブラリの一覧を示す．

表 5.1: システムが依存するライブラリの一覧

ライブラリ名	バージョン
axios	0.21.1
chalk	4.1.1
commander	7.2.0
playwright	1.16.1

`package.json` が存在するディレクトリ上で，リスト 5.1 に示すコマンドを実行することで，`npm` によりライブラリがインストールされる．

リスト 5.1: システムが依存するライブラリをインストールするコマンドの実行例

```
1 $ npm install
2
3 added 55 packages, and audited 56 packages in 10s
```

```
4
5 6 packages are looking for funding
6   run ‘npm fund‘ for details
7
8 found 0 vulnerabilities
```

5.2　システムの実行

本システムは，リスト 5.2 に示すように，コマンドライン上で DOM-Based XSS の検知を行いたい Web ページの URL をコマンドライン引数として，本システムのエントリーポイントである `src/index.js`（p.61 のソースコード A.2）に与え実行することで，データフローの解析と攻撃シミュレーションの 2 段階による Web ページの解析を開始する．

リスト 5.2: システムを実行するコマンドの例

```
1 $ node src/index.js http://localhost:8080/vulnerable.html
```

GET パラメータ，Cookie，`localStorage`，`sessionStorage` などのキーを指定できる Source については，別途コマンドライン引数として与えることにより，Web ページの解析時に使われるキーを変更できる．これらのキーが与えられていない場合には，システムは `test` というキーを利用する．実際の Web ページで `test` というキーが利用されることは考えにくいため，本システムの使用者は，解析対象となる Web ページで使われているこれらのキーを事前に調べ，コマンドライン引数から与える必要がある．

ソースコード 5.1 に，本システムによる DOM-Based XSS の発見のために，GET パラメータのキーを指定する必要がある例を示す．

ソースコード 5.1: GET パラメータのキーの指定を必要とする Web ページの例

```
1  <!doctype html>
2  <html lang="en">
3    <head>
4      <meta charset="utf-8">
5      <title>Vulnerable Page</title>
6    </head>
7    <body>
8      <p>Hello, <span id="username">John</span>!</p>
9      <script>
10     const params = new URLSearchParams(location.search);
11     const username = params.get('username');
12
```

```
13      const span = document.getElementById('username');
14      span.innerHTML = username;
15      </script>
16    </body>
17  </html>
```

この例では，まず 10〜11 行目で，GET パラメータの username というキーに設定されている値が取得されている．13〜14 行目で Sink となる API である innerHTML にこの値が渡ることによって，DOM-Based XSS が発生する．

ソースコード 5.1 に示した Web ページに存在する DOM-Based XSS を本システムによって発見するために，コマンドライン引数から GET パラメータのキーを指定するコマンドの例をリスト 5.3 に示す．この例では，-g username というコマンドライン引数を与えられたために，Web ページの解析時に username という GET パラメータのキーが利用される．

リスト 5.3: システムが解析に利用する GET パラメータのキーを変更する例

```
1  $ node src/index.js -g username http://localhost:8080/vulnerable.html
```

このように，本システムの使用者は対象となる Web ページの URL と，その Web ページで使われている GET パラメータや Cookie などのキーを与えることで，本システムによる Web ページに存在する DOM-Based XSS の検知が自動で行える．

システムが解析に利用する GET パラメータの指定のほかに，システムで利用可能なコマンドラインオプションの一覧を表 5.2 に示す．

表 5.2: システムで利用可能なコマンドラインオプションの一覧

オプション名	説明
-f, --file	解析対象の Web ページのリストを，コマンドライン引数ではなく JSON ファイルから指定する．
-t, --timeout	解析時に，Web ブラウザが解析対象の Web ページに留まる時間をミリ秒単位で指定する．指定されていない場合には，3,000 ミリ秒が指定されたものとする．

-r，--result-format	解析結果の出力フォーマットを指定する．verbose を指定すると，発見したデータフローや脆弱性の詳細な情報を出力する．simple を指定すると，解析対処の Web ページ上でデータフローもしくは脆弱性を発見したかどうかという情報だけが出力される．指定されていない場合には，verbose が指定されたものとする．
-c，--cookie	解析に利用する Cookie のキーを変更する．カンマ区切りで複数のキーを与えることができる．指定されていない場合には，test というキーを利用する．
-l，--localstorage	解析に利用する localStorage のキーを変更する．カンマ区切りで複数のキーを与えることができる．指定されていない場合には，test というキーを利用する．
-s，--sessionstorage	解析に利用する sessionStorage のキーを変更する．カンマ区切りで複数のキーを与えることができる．指定されていない場合には，test というキーを利用する．
-g，--getparameter	解析に利用する GET パラメータのキーを変更する．カンマ区切りで複数のキーを与えることができる．指定されていない場合には，test というキーを利用する．

コマンドライン引数として与えられた，URL や GET パラメータのキーといった解析対象となる Web ページの情報を元にして，src/index.js は src/analyzeTarget.js（p.64 のソースコード A.3）を呼び出して DOM-Based XSS の検知を行う．src/analyzeTarget.js は src/getSuspiciousPaths.js（p.65 のソースコード A.4）によって，まず解析対象の Web ページに存在するデータフローの情報を収集する．もしデータフローが発見できれば，src/simulateAttack.js（p.79 のソースコード A.7）によって攻撃シミュレーションを行う．これらの工程が終わると，src/index.js は src/printResults.js（p.81 のソースコード A.8）によって Web ページの解析結果を出力する．

コマンドライン上から実行された本システムは，図 5.1 に示す流れで Web ページ上に存在する DOM-Based XSS の検知を行う．

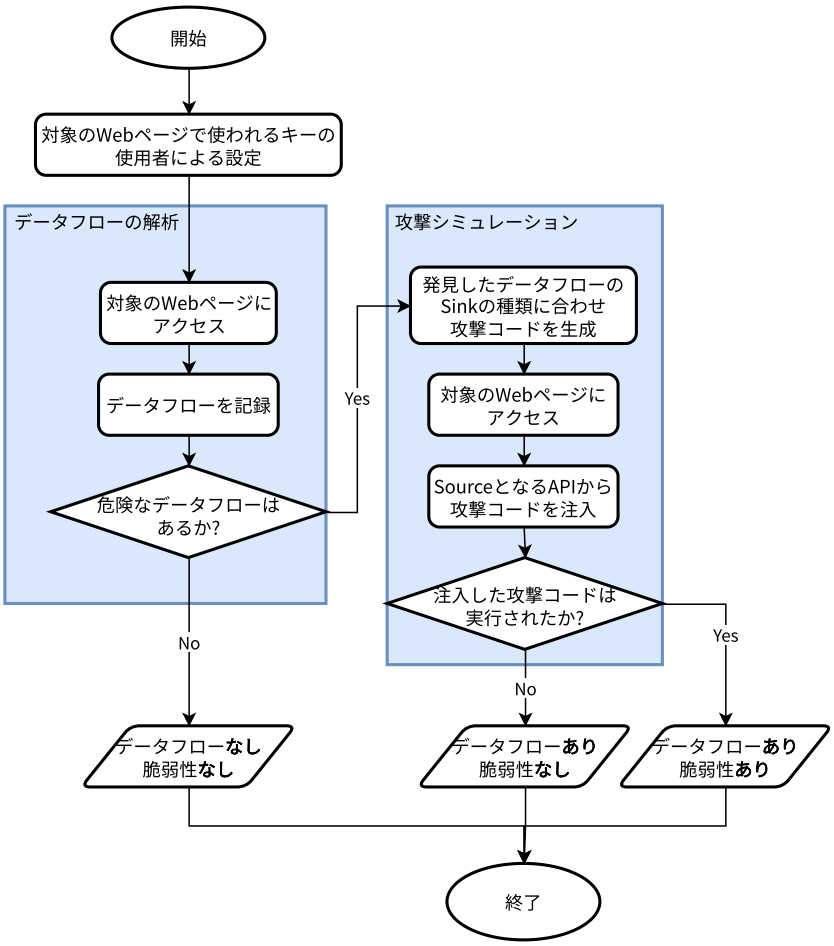

図 5.1: システム全体の流れ

5.3　システムのファイル構成

本システムを構成するファイルの一覧を表 5.3 に示す．これらのファイルの詳細な内容は付録 A に添付する．

表 5.3: システムを構成するファイルの一覧

ファイル名	説明
`package.json`（ソースコード A.1）	システムが依存するライブラリなどの情報を含む．

表は次ページに続く

前ページからの続き

ファイル名	説明
src/index.js（ソースコード A.2）	システムのエントリーポイントであり，与えられたコマンドライン引数をパースし，解析対象となる Web ページのリストや，それらの Web ページで使われるパラメータのキーなどの情報を取得する．これらの情報に基づいて，src/analyzeTarget.js に Web ページの URL を与えて呼び出すことで 2 段階の解析を行い，src/printResults.js を呼び出して解析結果を出力する．
src/analyzeTarget.js（ソースコード A.3）	src/getSuspiciousPaths.js を呼び出して，与えられた URL に存在するデータフローの一覧を取得する．これらのデータフローに対して，src/simulateAttack.js を呼び出して攻撃シミュレーションを行う．
src/getSuspiciousPaths.js（ソースコード A.4）	5.4 節で述べる，Web ページに存在するデータフローの解析を行う．
src/generateInjectScript.js（ソースコード A.5）	5.4.1 節で述べる Source の設定や，5.4.2 節で述べる Sink となる API へのフックといった処理のための，Web ブラウザに Web ページ上で実行させる JavaScript コードを生成する．
src/fireEvents.js（ソースコード A.6）	5.4.6 節で述べるイベントの強制発火処理を実装している．Web ページ上に存在するイベントリスナが登録されているイベントの取得と，それらの発火を行う．
src/simulateAttack.js（ソースコード A.7）	5.5 節で述べる，発見したデータフローへの攻撃シミュレーションを行う．
src/printResults.js（ソースコード A.8）	5.6 節で述べる，データフローの解析と攻撃シミュレーションの結果の出力を行う．
src/util.js（ソースコード A.9）	ランダムな文字列の生成や，5.4.3 節で述べるコールスタックのパースといった処理をまとめている．

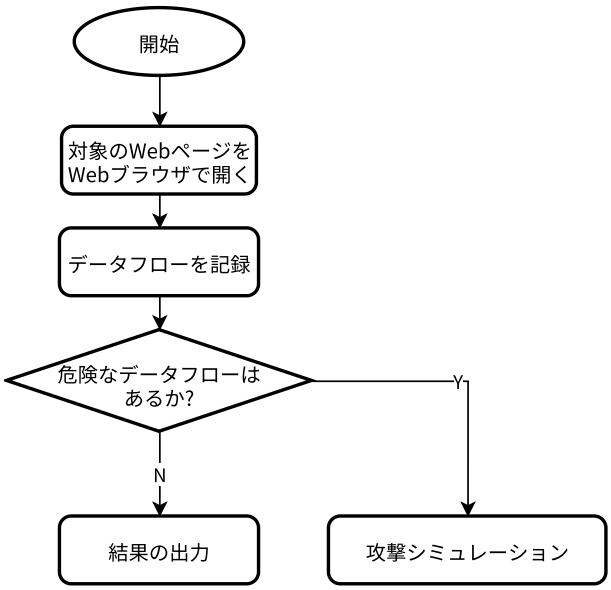

図 5.2: データフローの解析の流れ

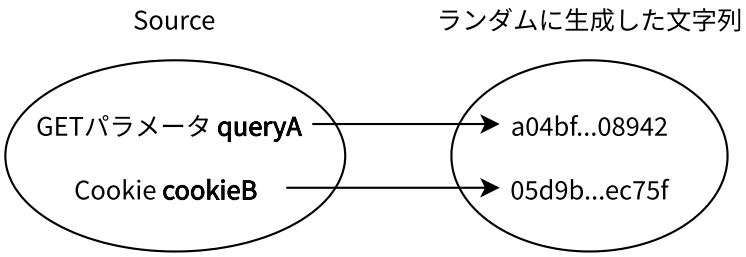

図 5.3: Source へのランダムに生成した文字列の割り当て

5.4　データフローの解析

コマンドラインから実行された本システムは，Chromium および Firefox の 2 つの Web ブラウザによって解析対象の Web ページを開き，Sink となる API がどのように利用されたかという情報を記録することによって，危険なデータフローを発見する（図 5.2）.

5.4.1　Source の設定

データフローの解析の段階では，まず本システムが解析の対象とする Web ページの Cookie, GET パラメータ，フラグメント識別子，リファラ，`localStorage`，`sessionStorage`，ウィンドウ名の 7 つの Source について，それぞれ一意に識別可能になるようにランダムに生成した文字列を割り当てる（図 5.3）.

これらの生成した文字列は，解析対象の Web ページ上で JavaScript コードから対応する Source にアクセスされた際にその値を含んで返されるよう設定する.

GETパラメータ，フラグメント識別子，リファラの3種類のSourceについては，データフローの解析の際に閲覧するWebページのURLや，送信されるHTTPリクエストヘッダとして付与することにより設定する．

Cookie，localStorage，sessionStorage，ウィンドウ名の4種類のSourceは，Webページを開く前のPlaywrightからの設定は難しい．したがって，src/generateInjectScript.jsの一部（p.77のソースコードA.5，317〜336行目）であるソースコード5.2に示す処理によって，これらのSourceが対応する文字列を返すよう設定するJavaScriptコードを生成する．setCookie，setWindowNameなどの変数にはそれぞれ対応するSourceに文字列を代入するJavaScriptコードが含まれている．システムは最終的にそれらのJavaScriptコードを結合し，Webページが開かれた際にWebブラウザに実行させる．

ソースコード 5.2: Sourceが対応する文字列を返すよう設定するJavaScriptコードの生成

```
1  // Cookieを設定するコード
2  let setCookie = '';
3  for (const cookie of identifiers.getIdentifiersBySource('cookie')) {
4    setCookie += `document.cookie = '${cookie.key}=${cookie.value}';\n`;
5  }
6
7  // localStorageを設定するコード
8  let setLocalStorage = '';
9  for (const storage of identifiers.getIdentifiersBySource('localStorage'))
       {
10   setLocalStorage += `localStorage.setItem('${storage.key}', '${storage.
       value}');\n`;
11 }
12
13 // sessionStorageを設定するコード
14 let setSessionStorage = '';
15 for (const storage of identifiers.getIdentifiersBySource('sessionStorage'
     )) {
16   setSessionStorage += `sessionStorage.setItem('${storage.key}', '${
       storage.value}');\n`;
17 }
18
19 // ウィンドウ名を設定するコード
20 let setWindowName = `window.name = '${identifiers.getIdentifierBySource('
     windowName').value}';\n`;
```

リスト 5.4 に示すコマンドにより，Cookie は keyA，GET パラメータは keyB，localStorage は keyC，sessionStorage は keyD をそれぞれ解析に使うキーとして，http://localhost:8080 という URL の Web ページにおいてデータフローを解析する場合を例にとる．

リスト 5.4: http://localhost:8080 に存在するデータフローを解析するコマンド

```
1 $ node src/index.js -c keyA -g keyB -l keyC -s keyD http://localhost:8080
```

このとき，データフローの解析の段階では，まず表 5.4 に示すように，各 Source に対応するランダムな文字列が生成される．

表 5.4: Source と対応するランダムな文字列の例

Source	生成されたランダムな文字列
Cookie（keyA）	14b01b45
GET パラメータ（keyB）	e68d0b95
フラグメント識別子	8ac7d2d5
リファラ	cf286729
localStorage（keyC）	009aa1fc
sessionStorage（keyD）	a1df33bf
ウィンドウ名	9bd86e3a

対象の Web ページの URL と同じオリジンを持つ http://localhost:8080/?test=cf286729 という URL をリファラとして，システムは Web ブラウザに GET パラメータやフラグメント識別子を含む http://localhost:8080/?keyB=javascript:e68d0b95#javascript:8ac7d2d5 という URL を開かせる．Web ページが読み込まれると，p.77 のソースコード A.5，348〜351 行目を元にして，ソースコード 5.3 に示すような JavaScript コードが生成され，Web ページ上で実行される．このような手順で Source の設定が行われる．複数の Web ページの URL が解析対象として与えられた場合には，それぞれ新たに Source に対応するランダムな文字列の生成と対応付けを行う．

ソースコード 5.3: Source の設定をする JavaScript コードの例

```
1 document.cookie = 'keyA=14b01b45';
2 localStorage.setItem('keyC', '009aa1fc');
3 sessionStorage.setItem('keyD', 'a1df33bf');
4 window.name = '9bd86e3a';
```

5.4.2　Sink となる API の呼び出しへのフック

どのような種類の Sink が，どのような引数を伴って呼び出されたかといった利用状況を記録できるよう，Web ページ上で Sink となる API が呼び出された際にそれらの情報を記録するフック処理を行う．

Sink の呼び出しのフック時には，Sink に与えられた引数，Sink の名前，その Sink を呼び出すために必要な安全な型，コールスタックの 4 つの情報を取得する．これらの情報は，それぞれ表 5.5 に示す目的のために収集している．

表 5.5: Sink の呼び出しのフック時に収集する情報と収集の目的

情報の種類	情報を収集する目的
Sink に与えられた引数	いずれかの Source から，呼び出された Sink へのデータフローが存在しているか判断するため
Sink の名前	データフローの解析結果を出力する際に，より詳細な情報を出力するため
Sink を呼び出すために必要な安全の型	攻撃シミュレーションの段階における，Sink に対応するペイロードの生成のため
コールスタック	Sink がソースコード上のどの位置で呼び出されているかを特定するため

Sink の利用状況の記録自体は Web ブラウザ側で行われているため，取得した情報を Node.js のメインプログラム側に引き渡す必要がある．Playwright には，指定した変数名で Web ブラウザ側から Node.js 側の関数にアクセスできるようになる page.exposeFunction と呼ばれる関数が存在している．

page.exposeFunction を利用する例をソースコード 5.4 に示す．この例では，4〜5 行目で

Chromium を立ち上げ，6〜8 行目で printConsole という名前の，引数として与えられた文字列を出力する関数を Web ブラウザ側からアクセスできるようにしている．その後，9 行目で Chromium に http://localhost:8080/expose-func.html という Web ページへアクセスさせている．

ソースコード 5.4: page.exposeFunction の利用例

```
1  const { chromium } = require('playwright');
2
3  (async () => {
4    const browser = await chromium.launch();
5    const page = await browser.newPage();
6    await page.exposeFunction('printConsole', message => {
7      console.log('[called]', message);
8    });
9    await page.goto('http://localhost:8080/expose-func.html');
10   await browser.close();
11 })();
```

http://localhost:8080/expose-func.html が返す HTML をソースコード 5.5 に示す．8〜10 行目では printConsole という関数を，'hello' という文字列を引数として呼び出している．Chromium がデフォルトの状態で提供するグローバル変数の中には，printConsole という関数は存在しない．

ソースコード 5.5: Web ページから Node.js 側の関数を呼び出す例

```
1  <!doctype html>
2  <html lang="en">
3    <head>
4      <meta charset="utf-8">
5      <title>page.exposeFunction</title>
6    </head>
7    <body>
8      <script>
9        printConsole('hello');
10     </script>
11   </body>
12 </html>
```

リスト 5.5 に示すコマンドによって，ソースコード 5.4 に示した JavaScript コードを実行する．“[called] hello” と出力されており，Web ページ側で printConsole という関数を呼び

出すことで，Node.js 側で定義した関数を実行できることが確認できる．また，Web ページ側から Node.js 側に引数として'hello' という文字列が渡っていることも確認できる．

リスト 5.5: page.exposeFunction を呼び出すコードを実行するコマンド

```
1 $ node page-exposefunction.js
2 [called] hello
```

この Playwright の機能を利用して，p.67 のソースコード A.4，86〜109 行目に示すように取得した情報を受け取る関数を Node.js 側から Web ブラウザ側に公開する．Web ブラウザ側では，Sink の利用状況を取得した後に，この関数を通じて Node.js 側に引き渡す．Node.js 側は page.exposeFunction の返り値として，Web ブラウザ側から送られてきたオブジェクトを受け取ることができない．したがって，page.exposeFunction に与えた関数内から，関数外で定義されている変数に含まれる配列に，要素として追加することでオブジェクトを受け取る（p.68 のソースコード A.4，106 行目）．

フックを実現する処理は，利用している Web ブラウザにおいて Trusted Types が利用可能かどうかにより方法を変更する．

5.4.3　コールスタックからの Sink の呼び出し位置の取得

Chromium と Firefox では，コード内で発生したエラー情報を持つ Error オブジェクトは，そのエラーがどのような経緯で発生したかを意味するコールスタックを返す，stack と呼ばれるプロパティを持つ．Error オブジェクトの stack プロパティを参照し，出力する JavaScript コードの例をソースコード 5.6 に示す．

この例では，f という関数は g という関数を呼び出し，g という関数は Error オブジェクトを生成し，throw 文によってエラーを発生させる．この f を try-catch 文中で呼び出し，g が生成した Error オブジェクトのコールスタックを表示させている．

ソースコード 5.6: コールスタックを表示する JavaScript コード

```
1 <!doctype html>
2 <html lang="en">
3     <head>
4         <meta charset="utf-8">
5         <title>Call Stack</title>
6     </head>
7     <body>
8         <script>
```

```
9           function g() {
10            throw new Error();
11          }
12
13          function f() {
14            g();
15          }
16
17          try {
18            f();
19          } catch (e) {
20            console.log(e.stack);
21          }
22          </script>
23      </body>
24  </html>
```

Chromium においては，リスト 5.6 に示すようなコールスタックが表示される．この例では call-stack.html の 18 行 11 列目において f が呼び出され，14 行 11 列目において g が呼び出され，そして 10 行 17 列目でエラーが発生しているという情報が出力されている．

リスト 5.6: Chromium におけるコールスタックの例

```
1  Error
2      at g (call-stack.html:10:17)
3      at f (call-stack.html:14:11)
4      at call-stack.html:18:11
```

Firefox においては，リスト 5.7 に示すようなコールスタックが表示される．Chromium の例と同様の情報が表示されているが，フォーマットが異なっている．

リスト 5.7: Firefox におけるコールスタックの例

```
1  g@http://localhost:8000/call-stack.html:10:17
2  f@http://localhost:8000/call-stack.html:14:11
3  @http://localhost:8000/call-stack.html:18:11
```

このようなコールスタックの性質を利用して，p.77 のソースコード A.5，341〜342 行目に示すように，本システムは Sink となる API のフック処理において意図的にエラーを発生させ，取得した Error オブジェクトからコールスタックの情報を取得することで，Sink が呼び出されたソースコード上の位置を特定する．Chromium と Firefox でコールスタックのフォーマッ

トが異なっていることから，それぞれ別のパーサを用いてパースする（p.83 のソースコード A.9，13〜66 行目）．

5.4.4　Chromium における Sink となる API へのフック

Chromium 97.0.4666.0 では，p.69 のソースコード A.5，3〜10 行目に示すように，Filedescriptor の提案した手法により Trusted Types のデフォルトポリシーを利用して Sink となる API へのフックを行う．

デフォルトポリシーの定義に加えて，Web サーバから HTTP レスポンスが返ってきた際にフックし，CSP ヘッダを `require-trusted-types-for 'script'` という値に書き換える．この書き換えにより，元々は Trusted Types が有効化されていなかった Web ページにおいても，強制的に Trusted Types が有効化される．

このように Trusted Types とデフォルトポリシーが有効化された状態で，Web ページ上で Sink となる API が呼び出されることで，その利用状況が記録されるようになる．

5.4.5　Firefox における Sink となる API へのフック

Firefox 93.0 では，Trusted Types が実装されておらず，デフォルトポリシーを利用した Sink となる API へのフックができない．したがって，p.69 のソースコード A.5，12〜69 行目に示すように，Firefox においては Sink となる API の定義を置き換えることによって，Trusted Types を用いず，Sink となる API の呼び出しへのフック処理を実現する．

Sink となる API には，`eval` のように関数として振る舞うもののほか，`Element.prototype.innerHTML` のようにプロパティとして振る舞うものがある．関数として振る舞う Sink は，引数として入力を与え関数として呼び出すことによって動作する．プロパティとして振る舞う Sink は，代入時の右辺として入力を与えることによって動作する．

JavaScript には，指定したプロパティへの代入が行われた時に，事前にユーザが定義した関数が呼び出されるよう設定できるセッターと呼ばれる機能が存在している．プロパティとして振る舞う Sink は，このようなセッターによって実現されている．

関数として振る舞う Sink であればその関数を，プロパティとして振る舞う Sink についてはそのセッターを Sink の利用状況を記録する関数に置き換えることで，Trusted Types のデフォルトポリシーを利用せず Sink となる API へのフックを行う．`fetch` という API を置き換えた際の動作を図 5.4 に示す．置き換え処理後に `fetch` 関数が呼び出されると，まずフック処理として登録された関数が呼び出され，渡された引数などの情報が Node.js 側に渡される．情報の

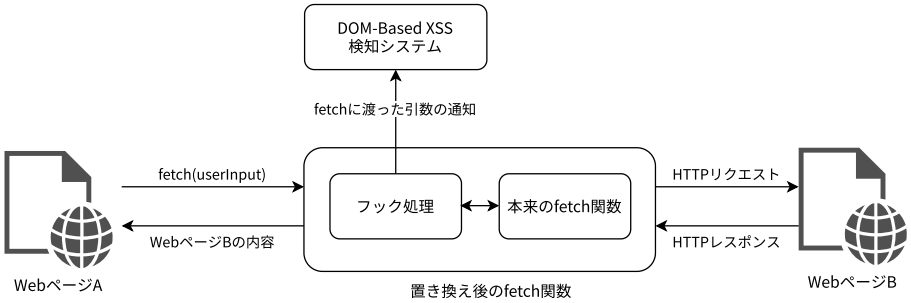

図 5.4: fetch を置き換えた際の Web ブラウザの挙動

引き渡しが終わった後に本来の fetch 関数を呼び出すことで，fetch を呼び出した元の Web ページの動作に影響を与えない．

関数またはセッターの置き換えは，オブジェクトのプロパティの定義や再定義ができる関数である Object.defineProperty によって行う．Object.defineProperty を用いた関数の置き換えの例をソースコード 5.7 に示す．この例では，関数としての呼び出しによって動作する eval の置き換えを試みる．

この例では eval を，“eval called” と出力する関数に置き換えている．第 1 引数として window，第 2 引数として’eval’，そして第 3 引数として value というプロパティを持つオブジェクトを与えている．これは，eval というプロパティにアクセスした際に，Object.defineProperty の呼び出し時に設定した関数を返すように変更することを意味する．

ソースコード 5.7: Object.defineProperty による関数の置き換えの例

```
1 Object.defineProperty(window, 'eval', {
2   value: () => { console.log('eval called'); }
3 });
```

Object.defineProperty を用いたセッターの置き換えの例をソースコード 5.8 に示す．この例では，文字列の代入によって動作する Element.prototype.innerHTML の置き換えを試みる．

この例では Element.prototype.innerHTML を，呼び出された際に “innerHTML called” と出力する関数に置き換えている．第 1 引数としてプロパティが置き換えられるオブジェクトである Element.prototype，第 2 引数として置き換えるプロパティ名である’innerHTML’，そして第 3 引数として置き換え後のプロパティの設定を意味するオブジェクトを与えている．第 3 引数として set というプロパティを持つオブジェクトを与えているが，これはセッターをその値として持つ関数に置き換えることを意味する．

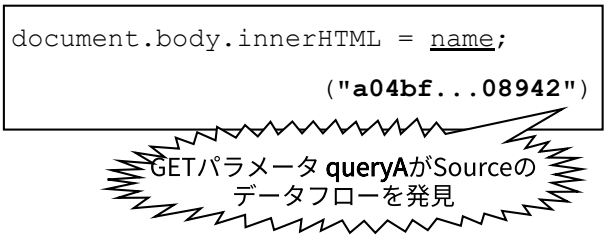

図 5.5: 引数の確認によりデータフローが発見できた例

ソースコード 5.8: Object.defineProperty によるセッターの置き換えの例

```
1 Object.defineProperty(Element.prototype, 'innerHTML', {
2   set: () => { console.log('innerHTML called'); }
3 });
```

5.4.6　イベントの強制発火

データフローの中には，ボタンのクリックやフォームへの入力といった操作をきっかけとして，Sink となる API が呼び出されるものもある．このようなデータフローを考慮して，操作に紐付いたイベントの強制的な発火により検知を試みる．

まず，解析対象の Web ページ上に存在しているすべての HTML の要素に対して，Playwright による Web ブラウザのデバッグ機能を用いて，発火時に実行される関数（イベントリスナ）が登録されているイベントのリストを取得する（p.78 のソースコード A.6，21〜24 行目）．取得したイベントについて，それぞれ発火させる JavaScript コードを生成し，Web ページ上で実行する（p.78 のソースコード A.6，26〜36 行目）．

5.4.7　データフローが存在しているかの判定

解析対象の Web ページ上にデータフローが存在しているかの判定は，Sink となる API へのフックによって得られた情報である，与えられた引数を確認することにより行う．もし引数に，いずれかの Source に対応するランダムな文字列が含まれていれば，その Source から呼び出された Sink へのデータフローが存在しているとみなす（図 5.5）．もしデータフローが発見できれば，それらが脆弱性であるかどうかを検証する攻撃シミュレーションの段階に進む．

5.5　攻撃シミュレーション

攻撃シミュレーションの段階では，前の段階で発見できたデータフローについて，それぞれ対応する Source にアクセスされた際に，ペイロードがその返り値に含まれるように URL やリファラなどを変更し，攻撃を試みることで脆弱性であるかどうかを検証する（図 5.6）．

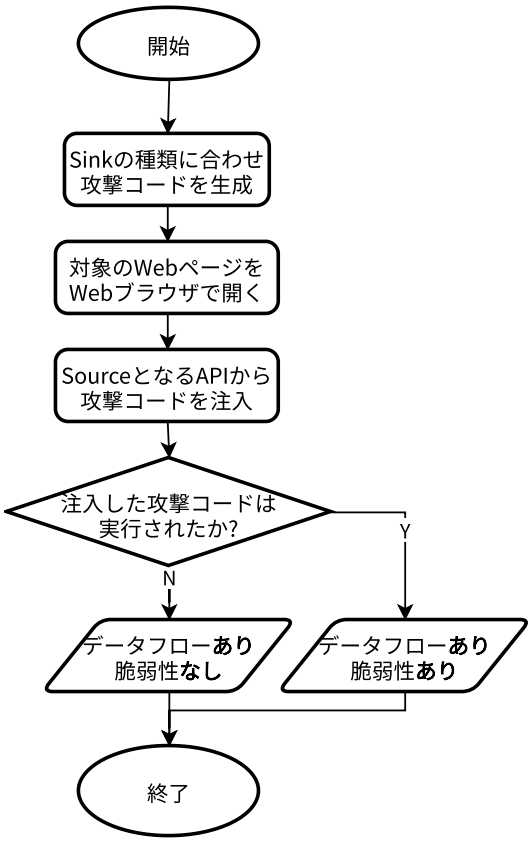

図 5.6: 攻撃シミュレーションの流れ

5.5.1　ペイロードの生成

攻撃シミュレーションの対象となるデータフローについて，その Sink の種類に合わせて，`src/simulateAttack.js` の一部（p.79 のソースコード A.7，8〜28 行目）であるソースコード 5.9 に示す処理により，`alert` 関数が呼び出される挙動を示すペイロードを生成する．このとき，`alert` 関数の引数としてランダムに生成した文字列が与えられるようにする．

ソースコード 5.9: Sink の種類に応じてペイロードを生成する処理

```
1  function generatePayload(uniqueString, { source, type }) {
2    const payload = `alert('${uniqueString}')`;
3
4    // eval等
5    if (type === 'TrustedScript') {
6      return `javascript:${payload}`;
7    }
8
9    // innerHTML等
10   if (type === 'TrustedHTML') {
11     return `<img src="xxx" onerror="${payload}">`
```

```
12    }
13
14    // scriptのsrc等
15    if (type === 'TrustedScriptURL') {
16      return `data:,${payload}`
17    }
18
19    return '';
20  };
```

eval のように，与えられた引数が JavaScript コードとして実行される Sink であれば，実行可能な JavaScript コードをペイロードとして生成する．JavaScript コードの前に javascript: という文字列を結合する処理が施されているが，これは location.href などの指定した URL への遷移が行われる Sink も，Trusted Types では与えられた引数が JavaScript コードとして実行されるものとして扱われるためである．そのような Sink にも対応するため，javascript: スキームを含む URL としても，また javascript: の部分がラベル名として扱われるために JavaScript コードとしても有効であるペイロードを生成している．

Element.prototype.innerHTML のように，与えられた引数が HTML として解釈され展開される Sink であれば，表示されると JavaScript コードが実行される HTML をペイロードとして生成する．ペイロードに含まれる HTML として img 要素を利用しているが，これは画像の読み込みが失敗した場合に属性値が JavaScript コードとして実行される onerror 属性を付与することにより，JavaScript コードを実行させることが容易であるためである．

script 要素の src 属性のように，与えられた引数が JavaScript コードの URL として扱われる Sink であれば，読み込まれると JavaScript コードが実行される URL をペイロードとして生成する．生成されるペイロードは data: スキームを先頭に含む URL である．

5.5.2　攻撃シミュレーションの準備

DOM-Based XSS であるかの判定の対象となるデータフローについて，解析対象の Web ページ上でその Source にアクセスされた際に生成したペイロードを含む値が返されるよう設定する．5.4.1 節に示したデータフローの解析の段階における方法と同様に，Web ページの閲覧以前に設定可能な GET パラメータなどの Source は，アクセスする URL の GET パラメータやフラグメント識別子に含めるほか，送信される HTTP リクエストヘッダを変更することで対応する（p.80 のソースコード A.7，69〜76 行目）．それ以外の Source は，localStorage や

document.cookie といったプロパティの書き換えを行う JavaScript コードを，閲覧時に Web ページ上で実行することで対応する（p.79 のソースコード A.7，30〜52 行目）.

5.5.3　攻撃シミュレーションの成否の判定

Source へのペイロードの設定が終わると，システムは解析対象の Web ページを開き，攻撃シミュレーションを行う.

このとき，alert 関数が呼び出されるとダイアログが表示されることを利用して，攻撃シミュレーションが成功したかどうかを判定する．脆弱性の含まれない通常の処理の中にダイアログを表示する処理が含まれている可能性があるため，ダイアログの表示時に引数として与えられた文字列を確認し，攻撃シミュレーションによるものかを判断する．事前に生成したランダムな文字列が引数として与えられていれば，そのダイアログの表示は攻撃シミュレーションによるものと判断する（p.80 のソースコード A.7，78〜88 行目）.

攻撃シミュレーションが成功した場合には，解析対象の Web ページにそのデータフローを原因とする DOM-Based XSS があるとみなす.

5.6　解析結果の出力

データフローの解析と攻撃シミュレーションの 2 段階による Web ページの解析を終えると，本システムは発見したデータフローや DOM-Based XSS の情報を出力する（p.81 のソースコード A.8，7〜57 行目）.

本システムによる DOM-Based XSS の発見を試みる Web ページの例をソースコード 5.10 に示す．この Web ページにはデータフローが 2 つ存在している．一方はフラグメント識別子を Source，Element.innerHTML を Sink とするものであり，11 行目において Source からの値の取得と Sink の呼び出しが行われている．もう一方は key1 をキーとする GET パラメータを Source，Element.innerHTML を Sink とするものであり，13〜14 行目において Source からの値の取得と Sink の呼び出しが行われている.

この Web ページに存在している 2 つのデータフローのうち，key1 をキーとする GET パラメータを Source とするものは，Element.innerHTML の呼び出し時に引数から「<」を取り除き，HTML タグが含まれていたとしても有効でない形に無害化されるため脆弱性ではない.

ソースコード 5.10: システムによる DOM-Based XSS の発見を試みる Web ページの例

```
1 <!doctype html>
2 <html lang="en">
3   <head>
```

```
4       <meta charset="utf-8">
5       <title>Vulnerable Page</title>
6     </head>
7     <body>
8       <p>Hello, <span id="username">John</span>!</p>
9       <script>
10      const span = document.getElementById('username');
11      span.innerHTML = decodeURIComponent(location.hash.slice(1));
12
13      const params = new URLSearchParams(location.search);
14      span.innerHTML = params.get('key1').replaceAll('<', '');
15      </script>
16    </body>
17  </html>
```

解析結果の出力例をリスト 5.8 に示す．この例では，ソースコード 5.10 に示した HTML を返す `http://localhost:8080/vulnerable.html` という URL の Web ページの解析を行っている．

Web ブラウザの挙動や API の実装状況などの差異により，脆弱性によっては Firefox では悪用できるが，Chromium では悪用できないような可能性があることから，Web ブラウザによる差異を確認できるようにするため，解析に使用した Web ブラウザごとに解析結果を出力している．

4 行目，13 行目でともに出力されている “1 vulnerability found” 以下のメッセージは，解析対象の Web ページにおいて DOM-Based XSS が 1 件見つかったことを意味する．“Path” から始まる行はその脆弱性に対応する Source と Sink を意味し，この例ではフラグメント識別子が Source，`Element.prototype.innerHTML` が Sink となる脆弱性が見つかったことを意味する．“Location” から始まる行は脆弱性が存在している Sink の箇所を意味し，この例では `http://localhost:8080/vulnerable.html` の 11 行 20 列目に脆弱性が存在していることを意味する．Firefox においては脆弱性が 11 行 22 列目に存在していると出力しているが，いずれも同一の Sink の箇所を示している．これは Chromium と Firefox がそれぞれ採用している JavaScript エンジンの実装の差異によるものである．

8 行目，17 行目でともに出力されている “1 possible vulnerability found” 以下のメッセージは，解析対象の Web ページにおいて，検知できたものの攻撃シミュレーションが失敗したデータフローが 1 件見つかったことを意味する．9 行目，18 行目で出力されているメッセージは，

key1 という GET パラメータから Element.prototype.innerHTML へのデータフローを発見したことを意味する．データフローに関する情報は，脆弱性を発見できた場合と同じものを出力する．

攻撃シミュレーションが失敗したとしても，その理由が攻撃シミュレーションに使用されたペイロードがそのデータフローには合わないものであったというような場合であれば，実際には脆弱性が存在している可能性がある．このような理由から，攻撃シミュレーションの成否によらずデータフローの情報を出力している．

リスト 5.8: 本システムによる Web ページの解析結果の出力例

```
$ node src/index.js -g key1 http://localhost:8080/vulnerable.html
[*] Target: http://localhost:8080/vulnerable.html
[*] Browser: chromium
  [+] 1 vulnerability found
    Path        hash → Element innerHTML
    Location    http://localhost:8080/vulnerable.html:11:20

  [+] 1 possible vulnerability found
    Path        getParameter key1 → Element innerHTML
    Location    http://localhost:8080/vulnerable.html:14:20

[*] Browser: firefox
  [+] 1 vulnerability found
    Path        hash → Element.prototype.innerHTML
    Location    http://localhost:8080/vulnerable.html:11:22

  [+] 1 possible vulnerability found
    Path        getParameter key1 → Element.prototype.innerHTML
    Location    http://localhost:8080/vulnerable.html:14:41

==========
```

第6章　実験と考察

6.1　実験内容と結果

Google が脆弱性スキャナなどのテスト用として提供している，脆弱な Web ページの事例集である Firing Range[12] を対象として，本システムによる DOM-Based XSS の検知を試みた．68 サンプルについて検知を試み，うち 56 サンプル（82.4%）でデータフローを発見した．

データフローを発見した 56 サンプルのうち，26 サンプル（46.4%）が攻撃シミュレーションにより DOM-Based XSS を含むと判断された．全体としては，68 サンプルのうち 26 サンプル（38.2%）で DOM-Based XSS を発見した．68 サンプルにおいて解析にかかった時間は，合計で 5 分 16 秒であった．

本システムによる各サンプルの解析結果を表 6.1 に示す．表の列はそれぞれサンプルにおいて使われている Source と Sink の組み合わせ，データフローが検知されたか，DOM-Based XSS が検知されたかを意味する．データフローが検知されたか，DOM-Based XSS が検知されたかを意味する 2 つの列では，検知された場合には Yes，検知されなかった場合には No と表記している．

表 6.1: 本システムによる各サンプルの解析結果

Source	Sink	データフロー	脆弱性
`location.hash`	`location.assign`	Yes	Yes
`location.hash`	`document.write`	Yes	No
`location.hash`	`document.writeln`	Yes	No
`location.hash`	`eval`	Yes	Yes
`location.hash`	`Element.innerHTML`	Yes	No
`location.hash`	`Range.createContextualFragment`	Yes	No

表は次ページに続く

前ページからの続き

Source	Sink	データフロー	脆弱性
location.hash	location.replace	Yes	Yes
location.hash	setTimeout	Yes	Yes
location.hash	Function	Yes	Yes
location.hash	setAttribute	Yes	Yes
location.hash	addEventListener	Yes	No
location.hash	Element.onclick	Yes	No
location.hash	HTMLFormElement.action	Yes	No
location.hash	location	Yes	Yes
location.hash	HTMLIFrameElement.src	Yes	No
location.hash	HTMLBaseElement.href	Yes	No
location.hash	HTMLEmbedElement.src	Yes	No
location.hash	HTMLObjectElement.data	Yes	Yes
location.hash	HTMLScriptElement.src	Yes	Yes
location.hash	XMLHttpRequest.open	Yes	No
location.hash	HTMLInputElement.formAction	Yes	No
location.hash	HTMLAnchorElement.href	Yes	No
location.hash	HTMLFormElement.action	No	No
location.hash	window.open	Yes	No
location.hash	HTMLLinkElement.href	Yes	No
location.hash	HTMLParamElement.value	No	No
location.hash	Element.setAttributeNS	Yes	Yes
location.hash	fetch	Yes	No
location	location.assign	No	No
location	document.write	Yes	No
location	document.writeln	Yes	No
location	eval	No	No
location	Element.innerHTML	Yes	No

表は次ページに続く

前ページからの続き

Source	Sink	データフロー	脆弱性
location	Range.createContextualFragment	Yes	No
location	location.replace	No	No
location	setTimeout	Yes	No
location.search	document.write	Yes	No
location.search	location.assign	No	No
location.search	HTMLFrameElement.src	Yes	No
location.search	HTMLAreaElement.href	No	No
location.search	HTMLButtonElement.formAction	Yes	No
location.pathname	document.write	No	No
document.documentURI	document.write	Yes	No
document.baseURI	document.write	Yes	No
document.URL	document.write	Yes	No
document.URLUnencoded	document.write	No	No
document.cookie	document.write	Yes	Yes
document.cookie	Element.innerHTML	Yes	Yes
document.cookie	eval	Yes	Yes
document.referrer	document.write	Yes	No
document.referrer	Element.innerHTML	Yes	No
document.referrer	eval	Yes	No
window.name	document.write	Yes	Yes
window.name	Element.innerHTML	Yes	Yes
window.name	eval	Yes	Yes
localStorage['key']	eval	Yes	Yes
localStorage.key	document.write	Yes	Yes
localStorage.getItem	document.write	Yes	Yes
localStorage.getItem	Element.innerHTML	Yes	Yes
localStorage.getItem	eval	Yes	Yes

表は次ページに続く

前ページからの続き

Source	Sink	データフロー	脆弱性
sessionStorage['key']	eval	Yes	Yes
sessionStorage.key	document.write	Yes	Yes
sessionStorage.getItem	document.write	Yes	Yes
sessionStorage.getItem	Element.innerHTML	Yes	Yes
sessionStorage.getItem	eval	Yes	Yes
postMessage	document.write	No	No
postMessage	Element.innerHTML	No	No
postMessage	eval	No	No

6.2　考察

6.2.1　脆弱性が存在しなかったため攻撃シミュレーションに失敗したサンプル

location.hash，location.documentURI，document.URL といった URL の一部分を返す API を Source として持つ 16 サンプル（表 6.2）で攻撃シミュレーションに失敗したのは，解析に使用した Web ブラウザにおいて，これらの Source がパーセントエンコーディングされた URL を返すためであると考えられる．

表 6.2: URL の一部分を返す API を Source として持つサンプル

Source	Sink
location.hash	document.write
location.hash	document.writeln
location.hash	Element.innerHTML
location.hash	Range.createContextualFragment
location	document.write
location	document.writeln
location	Element.innerHTML
location	Range.createContextualFragment

表は次ページに続く

前ページからの続き

Source	Sink
location	setTimeout
location.search	document.write
document.documentURI	document.write
document.baseURI	document.write
document.URL	document.write
document.referrer	document.write
document.referrer	Element.innerHTML
document.referrer	eval

location.hash を Source，document.write を Sink とするサンプルを例にとる（ソースコード 6.1）．Sink となる document.write は，引数を HTML として解釈し出力する API である．vuln.html#<img src=x onerror=alert(123)>のように URL のフラグメント識別子に HTML を含めることによる攻撃が考えられるが，Chromium および Firefox では，このペイロードは機能しない．これは，いずれの Web ブラウザにおいても location.hash は，フラグメント識別子に含まれる「<」や「>」といった HTML で特殊な意味を持つ文字が，%3C や%3E といった無害な形に変換された文字列を返すためである．

ソースコード 6.1: location.hash を Source，document.write を Sink とするサンプル

```
1 <html>
2   <head><title>Address based DOM XSS</title></head>
3   <body>
4     <script>
5       var payload = window.location.hash.substr(1);document.write(payload);
6
7     </script>
8   </body>
9 </html>
```

HTMLLinkElement.href を Sink とする 1 サンプルで攻撃シミュレーションに失敗したのは，解析に使用した Web ブラウザにおいて既に削除されている，HTML Imports と呼ばれる機能が使われていたためと考えられる．HTML Imports は，rel 属性の値として import を持つ link

要素を作成することで，外部から HTML を取り込むことができる機能である [13].

このサンプルのソースコードをソースコード 6.2 に示す．このサンプルではフラグメント識別子を返す location.hash の一部を URL として，HTML Imports を利用し読み込みを試みる．しかしながら，Firefox では HTML Imports が実装されず [14]，また Chromium においても一時期実装されていたものの，2020 年 2 月までに削除された [15]．本システムでは，HTML Imports を実装していないか，既に機能が削除されたバージョンの Chromium と Firefox を利用しており，このサンプルは動作しない.

ソースコード 6.2: HTML Imports を Sink とするサンプル

```
1  <html>
2    <head>
3      <title>URL-based DOM XSS</title>
4    </head>
5    <body>
6      <script>
7        var payload = document.location.hash.substr(1);
8        var linkElement = document.createElement("link");
9        linkElement.rel = "import";
10       linkElement.href = payload;
11       document.body.appendChild(linkElement);
12     </script>
13   </body>
14 </html>
```

6.2.2　Sink が呼び出されなかったため攻撃シミュレーションに失敗したサンプル

Element.onclick，HTMLInputElement.formAction などを Sink とする 6 サンプル（表 6.3）で攻撃シミュレーションに失敗したのは，システムがこれらの Sink の呼び出しに必要な操作を見つけられなかったためであると考えられる.

表 6.3: Sink の呼び出しに特殊な操作が必要なサンプル

Source	Sink
location.hash	addEventListener
location.hash	Element.onclick
location.hash	HTMLFormElement.action

表は次ページに続く

前ページからの続き

Source	Sink
location.hash	HTMLInputElement.formAction
location.hash	HTMLAnchorElement.href
location.search	HTMLButtonElement.formAction

location.hash を Source，HTMLAnchorElement.href を Sink とするサンプルを例にとる（ソースコード 6.3）．Sink となる HTMLAnchorElement.href は，a 要素のリンク先の URL を意味する href 属性の値を変更する API である．javascript:alert(123) のように javascript: スキームを含む URL が a 要素の href 属性の値として設定されると，そのリンクをクリックした際に JavaScript コードが実行される．

本システムはイベントの探索によって発火になんらかの操作を必要とするデータフローの検知も試みるが，この探索はイベントリスナが登録されていることを前提としている．javascript: スキームを含む URL を a 要素の href 属性の値として設定したとしても，この操作によって a 要素のクリックなどのイベントにイベントリスナが登録されるわけではない．そのため，本システムはイベントの探索ではデータフローの発火に必要な操作を発見できない．

ソースコード 6.3: HTMLAnchorElement.href を Sink とするサンプル

```
1  <html>
2    <head>
3      <title>URL-based DOM XSS</title>
4    </head>
5    <body>
6      <script>
7        var payload = document.location.hash.substr(1);
8        var anchor = document.createElement("a");
9        anchor.href = payload;
10       anchor.text = "Here's␣an␣anchor␣link";
11       document.body.appendChild(anchor);
12     </script>
13   </body>
14 </html>
```

6.2.3　生成されたペイロードが適切でないため攻撃シミュレーションに失敗したサンプル

7 サンプル（表 6.4）で攻撃シミュレーションに失敗したのは，システムが対象の Sink に適切なペイロードを選択できなかったためであると考えられる．これらのサンプルが利用している Sink は，いずれも URL を引数として取る．

表 6.4: 生成されたペイロードが適切でなかったサンプル

Source	Sink
`location.hash`	`HTMLIFrameElement.src`
`location.hash`	`HTMLBaseElement.href`
`location.hash`	`HTMLEmbedElement.src`
`location.hash`	`XMLHttpRequest.open`
`location.hash`	`window.open`
`location.hash`	`fetch`
`location.search`	`HTMLFrameElement.src`

`location.hash` を Source，`HTMLBaseElement.href` を Sink とするサンプルを例にとる（ソースコード 6.4）．Sink となる `HTMLBaseElement.href` は，`base` 要素の `href` 属性の値を変更する API である．これにより，`base` 要素の影響範囲にある HTML タグによって，相対 URL で指定された JavaScript コードや画像などの読み込みが試みられると，その読み込み先の URL は `base` 要素の `href` 属性の値として設定された URL を基点として解決される．

このサンプルでは，`href` 属性が変更された `base` 要素を Web ページに追加した後に，`exploit.js` という相対 URL で JavaScript コードの読み込みが試みられる．`base` 要素の `href` 属性の値には外部の URL も設定できるため，攻撃者の Web サーバの URL を設定し，その Web サーバが悪意のある JavaScript コードを返すことで DOM-Based XSS が成立する．

このように，攻撃を成立させるためには `base` 要素の `href` 属性の値として細工した URL を設定する必要がある．本システムはこのような Sink が現れると，5.5.1 節で示したように `data:` スキームを含む URL をペイロードとして生成する．Chromium と Firefox のいずれにおいても，`base` 要素の `href` 属性の値として `data:` スキームを含む URL を設定することは許されていないため，このようなペイロードを使った攻撃シミュレーションは失敗する．

ソースコード 6.4: HTMLBaseElement.href を Sink とするサンプル

```
<html>
  <head>
    <title>URL-based DOM XSS</title>
  </head>
  <body>
    <script>
      var payload = document.location.hash.substr(1);
      var baseElement = document.createElement("base");
      baseElement.href = payload;
      document.head.appendChild(baseElement);

      var scriptTag = document.createElement("script");
      scriptTag.src = "exploit.js";
      document.body.appendChild(scriptTag);
    </script>
  </body>
</html>
```

第7章　おわりに

本研究では，Trusted Types のデフォルトポリシーを用いることで Sink となる API にフックし，API に渡ってきた引数の解析からデータフローを検知することで，既存の Web ブラウザや JavaScript エンジンのソースコードへの変更を加えることなく，動的解析によって DOM-Based XSS を検知できることが確認できた．

しかし，本システムは Web ブラウザを用いて Web ページ上の JavaScript コードを実行し，解析する動的解析を採用しているため，発見できるデータフローは実行された範囲で存在するものに限られる．したがって，複雑な条件を満たした場合にのみ利用されるデータフローが存在すれば，検知に漏れが発生する可能性がある．また，攻撃シミュレーションの段階においても，本システムは Sink の種類のみに基づいてテンプレートを選択することでペイロードを生成しており，Source から文字列が Sink に渡るまでに切り取りや置換といった加工がされている可能性があるという点を考慮していない．そのため，実際には脆弱性であるデータフローであっても，適切でないペイロードが生成されたために検知に失敗する可能性がある．今後の展望として，データフローの検知の網羅性やペイロードの精度の改善のための，動的解析と静的解析の併用が考えられる．

参考文献，参考URL等

[1] “脆弱性対策情報データベース JVN iPedia の登録状況 [2021 年第 4 四半期（10 月〜12 月）]：IPA 独立行政法人 情報処理推進機構”. https://www.ipa.go.jp/security/vuln/report/JVNiPedia2021q4.html，2022 年 1 月 31 日確認.

[2] “CWE - CWE-79: Improper Neutralization of Input During Web Page Generation (’Cross-site Scripting’) (4.6)”. https://cwe.mitre.org/data/definitions/79.html，2022 年 1 月 31 日確認.

[3] “Content Security Policy Level 2”. https://www.w3.org/TR/CSP2/，2022 年 1 月 31 日確認.

[4] “Trusted Types”. https://w3c.github.io/webappsec-trusted-types/dist/spec/，2022 年 1 月 31 日確認.

[5] “Node.js”. https://nodejs.org/，2022 年 1 月 31 日確認.

[6] “npm”. https://www.npmjs.com/，2022 年 1 月 31 日確認.

[7] “microsoft/playwright: Playwright is a framework for Web Testing and Automation. It allows testing Chromium, Firefox and WebKit with a single API.”. https://github.com/microsoft/playwright，2022 年 1 月 31 日確認.

[8] R. Wang, G. Xu, X. Zeng, X. Li and Z. Feng: “Tt-xss: A novel taint tracking based dynamic detection framework for dom cross-site scripting”, Journal of Parallel and Distributed Computing, **118**, pp. 100–106 (2018).

[9] “PhantomJS - Scriptable Headless Browser”. https://phantomjs.org/，2022 年 1 月 31 日確認.

[10] 山崎, 垣内, 新井, 藤川：“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”, Technical Report 5, 奈良先端科学技術大学院大学先端科学技術研究科, 奈良先端科学技術大学院大学総合情報基盤セン

ター, 奈良先端科学技術大学院大学総合情報基盤センター, 奈良先端科学技術大学院大学総合情報基盤センター (2019).

[11] “filedescriptor/untrusted-types”. https://github.com/filedescriptor/untrusted-types，2022 年 1 月 31 日確認.

[12] “Firing Range”. http://public-firing-range.appspot.com/，2022 年 1 月 31 日確認.

[13] “HTML Imports”. https://www.w3.org/TR/html-imports/，2022 年 1 月 31 日確認.

[14] “877072 - Implement HTML Imports”. https://bugzilla.mozilla.org/show_bug.cgi?id=877072，2022 年 1 月 31 日確認.

[15] “HTML Imports - Chrome Platform Status”. https://chromestatus.com/feature/5144752345317376，2022 年 1 月 31 日確認.

付 録A　本研究のソースコード

ソースコード A.1: package.json

```
{
  "name": "tt-domxss-finder",
  "version": "1.0.0",
  "description": "",
  "main": "src/index.js",
  "scripts": {
    "start": "node␣src/index.js"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "axios": "^0.21.1",
    "chalk": "^4.1.1",
    "commander": "^7.2.0",
    "playwright": "^1.16.1"
  }
}
```

ソースコード A.2: src/index.js

```
const chalk = require('chalk');
const fs = require('fs');
const playwright = require('playwright');
const { Command } = require('commander');
const { analyzeTarget } = require('./analyzeTarget');
const { printPaths } = require('./printResults');

const BROWSERS = ['chromium', 'firefox'];

// メイン処理
const main = async () => {
```

```
12  // コマンドライン引数とか
13  const program = new Command();
14  program.version('0.0.1');
15  program
16    .option('-f, --file <filename>', 'give URL list as JSON')
17    .option('-t, --timeout <milliseconds>', 'set browsing timeout', '3000')
18    .option('-r, --result-format <style>', 'set result format', 'verbose')
19    .option('-c, --cookie <cookie name>', 'set cookie name to be injected',
          'test')
20    .option('-l, --localstorage <localStorage key>', 'set localStorage key
        to be injected', 'test')
21    .option('-s, --sessionstorage <sessionStorage key>', 'set
        sessionStorage key to be injected', 'test')
22    .option('-g, --getparameter <get parameter>', 'set GET parameter to be
        injected', 'test')
23    .option('--no-simulate-attack', 'only shows suspicious data flows',
        false);
24  program.parse(process.argv);
25
26  const opts = program.opts();
27  if (!opts.file && program.args.length === 0) {
28    program.help();
29  }
30
31  // -fオプションがあればJSONファイルを読み込む
32  // そうでなければコマンドライン引数からURLを得る
33  let targets;
34  if (opts.file) {
35    const data = fs.readFileSync(opts.file);
36    targets = JSON.parse(data);
37  } else {
38    targets = program.args;
39  }
40
41  // ブラウザやペイロードに関する各種オプション
42  opts.timeout = parseInt(opts.timeout, 10);
43  const parameters = {
44    cookieKey: opts.cookie.split(','),
45    localStorageKey: opts.localstorage.split(','),
46    sessionStorageKey: opts.sessionstorage.split(','),
47    getParameter: opts.getparameter.split(',')
```

```
48  };
49
50  // 各ブラウザの立ち上げ
51  let browsers = {};
52  for (const browserType of BROWSERS) {
53    browsers[browserType] = await playwright[browserType].launch({
54      headless: true,
55      args: [
56        '--no-sandbox'
57      ]
58    });
59  }
60
61  // 与えられたURLそれぞれに解析と攻撃
62  for (const url of targets) {
63    let foundVulnerability = false;
64    let foundPossibleVulnerability = false;
65
66    if (opts.resultFormat === 'verbose') {
67      process.stdout.write(`[*] Target: ${url}\n`);
68    }
69
70    for (const browserType of BROWSERS) {
71      if (opts.resultFormat === 'verbose') {
72        process.stdout.write(`[*] Browser: ${browserType}\n`);
73      }
74
75      const { vulnerabilities, possibleVulnerabilities } = await
           analyzeTarget(browsers[browserType], browserType, opts, url,
           parameters);
76      if (opts.resultFormat === 'verbose') {
77        printPaths(vulnerabilities, possibleVulnerabilities, browserType);
78      }
79
80      if (vulnerabilities.length > 0) {
81        foundVulnerability = true;
82        foundPossibleVulnerability = true;
83      } else if (possibleVulnerabilities.length > 0) {
84        foundPossibleVulnerability = true;
85      }
86    }
```

```
87
88      if (opts.resultFormat === 'verbose') {
89        process.stdout.write(`==========\n`);
90      }
91
92      if (opts.resultFormat === 'simple') {
93        process.stdout.write(foundPossibleVulnerability ? chalk.red('Y') :
              chalk.green('N'));
94        process.stdout.write(foundVulnerability ? chalk.red('Y') : chalk.
              green('N'));
95        process.stdout.write(`\t${url}\n`);
96      }
97    }
98
99    // 各ブラウザを終了させる
100   for (const browserType of BROWSERS) {
101     await browsers[browserType].close();
102   }
103 };
104
105 main();
```

ソースコード A.3: src/analyzeTarget.js

```
1  const { getSuspiciousPaths } = require('./getSuspiciousPaths');
2  const { simulateAttack } = require('./simulateAttack');
3
4  module.exports = {
5    analyzeTarget
6  };
7
8  async function analyzeTarget(browser, browserType, opts, url, parameters) {
9    // 脆弱と思われるデータフローを列挙
10   const paths = await getSuspiciousPaths(browser, browserType, url,
         parameters);
11
12   // --no-simulate-attackオプションが付与されていなければ攻撃シミュレーション
13   let vulnerabilities = [];
14   let possibleVulnerabilities = [];
15   if (opts.simulateAttack) {
16     vulnerabilities = [];
```

```
17      for (const path of paths) {
18        const isVulnerable = await simulateAttack(browser, browserType, url,
              path, opts.timeout);
19        if (isVulnerable) {
20          vulnerabilities.push(path);
21        } else {
22          possibleVulnerabilities.push(path);
23        }
24      }
25    } else {
26      possibleVulnerabilities = paths;
27    }
28
29    // 解析結果を返す
30    return { vulnerabilities, possibleVulnerabilities };
31  }
```

ソースコード A.4: src/getSuspiciousPaths.js

```
1  const { generateRandomString, parseCallStack } = require('./util');
2  const { fireEvents } = require('./fireEvents');
3  const generateInjectScript = require('./generateInjectScript');
4  const axios = require('axios');
5
6  module.exports = {
7    getSuspiciousPaths
8  };
9
10 // HTTPレスポンスヘッダを改変し，CSPヘッダを書き換える
11 async function modifyResponse(route, request) {
12   const url = request.url();
13   const requestHeaders = request.headers();
14
15   let response;
16   try {
17     response = await axios({
18       responseType: 'arraybuffer',
19       method: request.method(),
20       url,
21       headers: requestHeaders,
22       body: request.postData(),
23     });
```

```
24    } catch (e) {
25      route.abort();
26      return;
27    }
28
29    const responseBody = response.data
30    let responseHeaders = response.headers;
31    responseHeaders['content-security-policy'] = "require-trusted-types-for␣'
          script'";
32
33    route.fulfill({
34      status: response.status,
35      headers: responseHeaders,
36      body: responseBody
37    });
38  }
39
40  class Identifiers {
41    constructor() {
42      this.identifiers = [];
43    }
44
45    add(source, key=null) {
46      this.identifiers.push({ source, key, value: generateRandomString() });
47    }
48
49    getIdentifiersBySource(source) {
50      return this.identifiers.filter(id => id.source === source);
51    }
52
53    getIdentifierBySource(source) {
54      return this.getIdentifiersBySource(source)[0];
55    }
56
57    getAllIdentifiers() {
58      return this.identifiers;
59    }
60  }
61
62  // 脆弱と思われるデータフローを列挙
63  async function getSuspiciousPaths(browser, browserType, url, options) {
```

```
64  // Sinkに与えられたユーザ入力からSourceを特定できるよう,
65  // それぞれにランダムな文字列を割り当てる
66  let identifiers = new Identifiers();
67
68  options.cookieKey.forEach(key => identifiers.add('cookie', key));
69  options.localStorageKey.forEach(key => identifiers.add('localStorage',
        key));
70  options.sessionStorageKey.forEach(key => identifiers.add('sessionStorage'
        , key));
71  identifiers.add('windowName');
72  identifiers.add('referer');
73  identifiers.add('hash');
74  options.getParameter.forEach(key => identifiers.add('getParameter', key
        ));
75
76  let result = [];
77  const page = await browser.newPage();
78
79  // HTTPリクエストをフックし, CSPヘッダを書き換える
80  await page.route('**/*', modifyResponse);
81
82  let visited = new Set();
83
84  // Sinkへのアクセスがあった際にWebブラウザから呼び出される関数を登録
85  const reportFunction = 'report' + generateRandomString();
86  await page.exposeFunction(reportFunction, (input, type, sink, stackTrace)
        => {
87    // 各Sourceに割り当てられた文字列が,
88    // Sinkに渡ったパラメータに含まれていないかどうかで,
89    // SourceからSinkへのデータフローがないか判定する
90    for (const identifier of identifiers.getAllIdentifiers()) {
91      const parsedStackTrace = parseCallStack(stackTrace, browserType);
92
93      const path = {
94        source: Object.assign({}, identifier, { name: identifier.source }),
95        type, sink, stackTrace: parsedStackTrace
96      };
97
98      // すでにチェックしたデータフローであればスキップ
99      const jsonPath = JSON.stringify(path);
100     if (visited.has(jsonPath)) {
```

```
101         break;
102       }
103       visited.add(jsonPath);
104
105       if (input.includes(identifier.value)) {
106         result.push(path);
107       }
108     }
109   });
110
111   // 解析対象のWebページが読み込まれた際にDefault policyとSourceの設定を行うスクリプトを仕込む
112   const script = generateInjectScript(reportFunction, browserType, identifiers);
113   await page.addInitScript(script);
114
115   // GETパラメータとフラグメント識別子もそれぞれ割り当てられた文字列を仕込む
116   let browsingUrl = url;
117
118   let queryParameters = identifiers.getIdentifiersBySource('getParameter');
119   queryParameters = queryParameters.map(param => `${param.key}=javascript:${param.value}`).join('&');
120
121   browsingUrl += '?' + queryParameters;
122   browsingUrl += `#javascript:${identifiers.getIdentifierBySource('hash').value}`
123
124   // リファラにも仕込んでおく
125   const { origin } = new URL(url);
126   const refererValue = `${origin}?test=${identifiers.getIdentifierBySource('referer').value}`;
127
128   // 準備が整ったのでWebブラウザに対象となるWebページにアクセスさせる
129   await page.goto(browsingUrl, {
130     timeout: options.timeout,
131     waitUntil: 'networkidle0',
132     referer: refererValue
133   }).catch(() => 'error');
134
```

```
135   // イベントの強制発火
136   if (browserType === 'chromium') {
137     await fireEvents(page);
138   }
139
140   await page.close();
141
142   return result;
143 };
```

ソースコード A.5: src/generateInjectScript.js

```
1  module.exports = generateInjectScript;
2
3  // Trusted Typesを使った, APIをフックする関数（Default policy）
4  const TRUSTED_TYPES_HOOK_FUNCTION = `
5  trustedTypes.createPolicy('default', {
6    createHTML: log,
7    createScript: log,
8    createScriptURL: log
9  });
10 `;
11
12 // Trusted Typesを使わない, APIをフックする関数
13 /*
14 使い方:
15 hookSink({
16   "name": "Element.prototype.innerHTML",
17   "type": "TrustedHTML",
18   "argumentIndex": 0,
19   "setTo": "set"
20 }, () => { console.log('hooked') })
21 */
22 const NON_TRUSTED_TYPES_HOOK_FUNCTION = `
23 // objectの深い階層にあるプロパティにアクセスする関数
24 let global = (0, eval)('this');
25 function lookupProperty(object, property) {
26   for (const part of property.split('.')) {
27     if (!(part in object)) {
28       return null;
29     }
30
```

```
31      object = object[part];
32    }
33    return object;
34  }
35
36  // 指定したsinkが利用されたときに，handlerを呼び出すようにする関数
37  function hookSink({ name, type, argumentIndex, setTo }, handler) {
38    let object = global;
39
40    const propertyName = name.slice(name.lastIndexOf('.') + 1);
41    if (name.includes('.')) {
42      const objectName = name.slice(0, name.lastIndexOf('.'));
43      object = lookupProperty(global, objectName);
44    }
45
46    if (object === null) {
47      console.warn('[object␣===␣null]', name);
48      return;
49    }
50
51    let propertyDescriptor = Object.getOwnPropertyDescriptor(object,
        propertyName)
52    if (propertyDescriptor === undefined) {
53      console.warn('[propertyDescriptor␣===␣undefined]', name);
54      return;
55    }
56
57    const func = propertyDescriptor[setTo];
58    propertyDescriptor[setTo] = function () {
59      handler(arguments[argumentIndex], type, name);
60      return func.apply(this, arguments);
61    };
62
63    if (propertyDescriptor.configurable) {
64      Object.defineProperty(object, propertyName, propertyDescriptor);
65    } else {
66      console.warn('[!configurable]', name);
67    }
68  }
69  `;
70
```

```
71 // Firefox向けのAPIをフックするスクリプト(APIの置き換え)
72 const FIREFOX_HOOK_SCRIPT = `
73 ${NON_TRUSTED_TYPES_HOOK_FUNCTION}
74
75 // 置き換えるSinkの一覧
76 const SINKS = [
77   {
78     "name": "Element.prototype.innerHTML",
79     "type": "TrustedHTML",
80     "argumentIndex": 0,
81     "setTo": "set"
82   },
83   {
84     "name": "Element.prototype.outerHTML",
85     "type": "TrustedHTML",
86     "argumentIndex": 0,
87     "setTo": "set"
88   },
89   {
90     "name": "HTMLIFrameElement.prototype.srcdoc",
91     "type": "TrustedHTML",
92     "argumentIndex": 0,
93     "setTo": "set"
94   },
95   {
96     "name": "HTMLScriptElement.prototype.text",
97     "type": "TrustedScript",
98     "argumentIndex": 0,
99     "setTo": "set"
100  },
101  {
102    "name": "HTMLScriptElement.prototype.src",
103    "type": "TrustedScriptURL",
104    "argumentIndex": 0,
105    "setTo": "set"
106  },
107  {
108    "name": "HTMLFormElement.prototype.action",
109    "type": "TrustedScriptURL",
110    "argumentIndex": 0,
111    "setTo": "set"
```

```
112   },
113   {
114     "name": "HTMLInputElement.prototype.formAction",
115     "type": "TrustedScriptURL",
116     "argumentIndex": 0,
117     "setTo": "set"
118   },
119   {
120     "name": "HTMLButtonElement.prototype.formAction",
121     "type": "TrustedScriptURL",
122     "argumentIndex": 0,
123     "setTo": "set"
124   },
125   {
126     "name": "HTMLAnchorElement.prototype.href",
127     "type": "TrustedScriptURL",
128     "argumentIndex": 0,
129     "setTo": "set"
130   },
131   {
132     "name": "HTMLIFrameElement.prototype.src",
133     "type": "TrustedHTML",
134     "argumentIndex": 0,
135     "setTo": "set"
136   },
137   {
138     "name": "HTMLFrameElement.prototype.src",
139     "type": "TrustedHTML",
140     "argumentIndex": 0,
141     "setTo": "set"
142   },
143   {
144     "name": "HTMLBaseElement.prototype.href",
145     "type": "TrustedScriptURL",
146     "argumentIndex": 0,
147     "setTo": "set"
148   },
149   {
150     "name": "HTMLLinkElement.prototype.href",
151     "type": "TrustedHTML",
152     "argumentIndex": 0,
```

```
153       "setTo": "set"
154     },
155     {
156       "name": "HTMLObjectElement.prototype.data",
157       "type": "TrustedScript",
158       "argumentIndex": 0,
159       "setTo": "set"
160     },
161     {
162       "name": "location.href",
163       "type": "TrustedScriptURL",
164       "argumentIndex": 0,
165       "setTo": "set"
166     },
167     {
168       "name": "location",
169       "type": "TrustedScriptURL",
170       "argumentIndex": 0,
171       "setTo": "set"
172     },
173     {
174       "name": "document.location",
175       "type": "TrustedScriptURL",
176       "argumentIndex": 0,
177       "setTo": "set"
178     },
179     {
180       "name": "Document.prototype.write",
181       "type": "TrustedHTML",
182       "argumentIndex": 0,
183       "setTo": "value"
184     },
185     {
186       "name": "Document.prototype.writeln",
187       "type": "TrustedHTML",
188       "argumentIndex": 0,
189       "setTo": "value"
190     },
191     {
192       "name": "Element.prototype.append",
193       "type": "TrustedHTML",
```

```
194       "argumentIndex": 0,
195       "setTo": "value"
196     },
197     {
198       "name": "Element.prototype.insertAdjacentHTML",
199       "type": "TrustedHTML",
200       "argumentIndex": 0,
201       "setTo": "value"
202     },
203     {
204       "name": "Range.prototype.createContextualFragment",
205       "type": "TrustedHTML",
206       "argumentIndex": 0,
207       "setTo": "value"
208     },
209     {
210       "name": "eval",
211       "type": "TrustedScript",
212       "argumentIndex": 0,
213       "setTo": "value"
214     },
215     {
216       "name": "Function",
217       "type": "TrustedScript",
218       "argumentIndex": 0,
219       "setTo": "value"
220     },
221     {
222       "name": "setTimeout",
223       "type": "TrustedScript",
224       "argumentIndex": 0,
225       "setTo": "value"
226     },
227     {
228       "name": "setInterval",
229       "type": "TrustedScript",
230       "argumentIndex": 0,
231       "setTo": "value"
232     },
233     {
234       "name": "setImmediate",
```

```
235       "type": "TrustedScript",
236       "argumentIndex": 0,
237       "setTo": "value"
238     },
239     {
240       "name": "location.assign",
241       "type": "TrustedScriptURL",
242       "argumentIndex": 0,
243       "setTo": "value"
244     },
245     {
246       "name": "location.replace",
247       "type": "TrustedScriptURL",
248       "argumentIndex": 0,
249       "setTo": "value"
250     },
251     {
252       "name": "window.open",
253       "type": "TrustedScriptURL",
254       "argumentIndex": 0,
255       "setTo": "value"
256     },
257     {
258       "name": "fetch",
259       "type": "Others",
260       "argumentIndex": 0,
261       "setTo": "value"
262     },
263     {
264       "name": "XMLHttpRequest.prototype.open",
265       "type": "Others",
266       "argumentIndex": 1,
267       "setTo": "value"
268     },
269     {
270       "name": "XMLHttpRequest.prototype.send",
271       "type": "Others",
272       "argumentIndex": 0,
273       "setTo": "value"
274     }
275 ];
```

```
276
277 for (const sink of SINKS) {
278   hookSink(sink, log);
279 }
280 `;
281
282 // Chromium向けのAPIをフックするスクリプト（Trusted Types + APIの置き換え）
283 const CHROMIUM_HOOK_SCRIPT = `
284   ${TRUSTED_TYPES_HOOK_FUNCTION}
285   ${NON_TRUSTED_TYPES_HOOK_FUNCTION}
286
287   hookSink({
288     "name": "fetch",
289     "type": "Others",
290     "argumentIndex": 0,
291     "setTo": "value"
292   }, log);
293   hookSink({
294     "name": "XMLHttpRequest.prototype.open",
295     "type": "Others",
296     "argumentIndex": 1,
297     "setTo": "value"
298   }, log);
299   hookSink({
300     "name": "XMLHttpRequest.prototype.send",
301     "type": "Others",
302     "argumentIndex": 0,
303     "setTo": "value"
304   }, log);
305 `;
306
307 const hookScripts = {
308   chromium: CHROMIUM_HOOK_SCRIPT,
309   firefox: FIREFOX_HOOK_SCRIPT
310 };
311
312 // 解析対象のページに挿入されるスクリプトを生成
313 // (Default policyの登録とSourceの設定を行う)
314 function generateInjectScript(report, browserType, identifiers) {
315   let hookScript = hookScripts[browserType];
316
```

```
317   // Cookieを設定するコード
318   let setCookie = '';
319   for (const cookie of identifiers.getIdentifiersBySource('cookie')) {
320     setCookie += `document.cookie = '${cookie.key}=${cookie.value}';\n`;
321   }
322
323   // localStorageを設定するコード
324   let setLocalStorage = '';
325   for (const storage of identifiers.getIdentifiersBySource('localStorage'))
          {
326     setLocalStorage += `localStorage.setItem('${storage.key}', '${storage.
          value}');\n`;
327   }
328
329   // sessionStorageを設定するコード
330   let setSessionStorage = '';
331   for (const storage of identifiers.getIdentifiersBySource('sessionStorage'
        )) {
332     setSessionStorage += `sessionStorage.setItem('${storage.key}', '${
          storage.value}');\n`;
333   }
334
335   // ウィンドウ名を設定するコード
336   let setWindowName = `window.name = '${identifiers.getIdentifierBySource('
        windowName').value}';\n`;
337
338   return `
339 (() => {
340   function log(input, type, sink) {
341     const e = new Error('a');
342     ${report}(input, type, sink, e.stack);
343     return input;
344   }
345
346   ${hookScript}
347
348   ${setCookie}
349   ${setLocalStorage}
350   ${setSessionStorage}
351   ${setWindowName}
352 })();
```

```
353   `;
354 };
```

ソースコード A.6: src/fireEvents.js

```
1  const { generateRandomString } = require('./util');
2
3  module.exports = {
4    fireEvents
5  };
6
7  async function fireEvents(page) {
8    const client = await page.context().newCDPSession(page);
9
10   // 解析対象のWebページ上に存在する要素の一覧を取得
11   const varName = `window.x${generateRandomString()}`;
12   const object = await client.send('Runtime.evaluate', {
13     expression: `${varName} = document.querySelectorAll("*")`
14   });
15   const elements = await client.send('Runtime.getProperties', {
16     objectId: object.result.objectId,
17     ownProperties: true
18   });
19
20   for (const element of elements.result) {
21     // 要素に登録されているイベントリスナの情報を取得
22     const { listeners } = await client.send('DOMDebugger.getEventListeners'
         , {
23       objectId: element.value.objectId
24     });
25
26     // 取得できた情報から, すべてのイベントを発火
27     for (const { type } of listeners) {
28       await client.send('Runtime.evaluate', {
29         expression: `
30         (() => {
31           const event = new Event('${type}');
32           ${varName}[${element.name}].dispatchEvent(event);
33         })();
34         `
35       });
36     }
```

```
37   }
38 }
```

ソースコード A.7: src/simulateAttack.js

```
1  const util = require('./util');
2  const { fireEvents } = require('./fireEvents');
3
4  module.exports = {
5    simulateAttack
6  };
7
8  // Sinkの種類に応じてペイロードを生成
9  function generatePayload(uniqueString, { source, type }) {
10   const payload = `alert('${uniqueString}')`;
11
12   // eval等
13   if (type === 'TrustedScript') {
14     return `javascript:${payload}`;
15   }
16
17   // innerHTML等
18   if (type === 'TrustedHTML') {
19     return `<img src="xxx" onerror="${payload}">`
20   }
21
22   // scriptのsrc等
23   if (type === 'TrustedScriptURL') {
24     return `data:,${payload}`
25   }
26
27   return '';
28 };
29
30 // 攻撃対象のページに挿入されるスクリプトを生成
31 // (Sourceにペイロードを仕込む)
32 function generateAttackScript(payload, { source }) {
33   let result = '';
34
35   if (source.name === 'cookie') {
36     result += `document.cookie = \`${source.key}=${payload}\``;
37   }
```

```
38
39    if (source.name === 'localStorage') {
40      result += `localStorage.setItem('${source.key}', \`${payload}\`)`;
41    }
42
43    if (source.name === 'sessionStorage') {
44      result += `sessionStorage.setItem('${source.key}', \`${payload}\`)`;
45    }
46
47    if (source.name === 'windowName') {
48      result += `window.name = \`${payload}\``;
49    }
50
51    return result;
52  };
53
54  // 攻撃シミュレーションを行う
55  async function simulateAttack(browser, browserType, url, path, timeout) {
56    const { source } = path;
57
58    // alert関数を呼び出すスクリプトをペイロードとする
59    // alert関数が攻撃によって呼び出された場合と,
60    // 正常な動作で呼び出された場合を判別するため,
61    // 引数に特定のランダムな文字列が含まれるようなペイロードを生成する
62    const uniqueString = util.generateRandomString();
63    const payload = generatePayload(uniqueString, path);
64    const script = generateAttackScript(payload, path);
65
66    const page = await browser.newPage();
67    await page.addInitScript(script);
68
69    // GETパラメータとフラグメント識別子にもペイロードを仕込む
70    let browsingUrl = url;
71    if (source.name === 'getParameter') {
72      browsingUrl += `?${source.key}=${encodeURI(payload)}`;
73    }
74    if (source.name === 'hash') {
75      browsingUrl += `#${encodeURI(payload)}`;
76    }
77
78    // alert関数が呼び出された場合の処理, 引数が特定の文字列でなければスルー
```

```
79    let isVulnerable = false;
80    page.on('dialog', async (dialog) => {
81      const message = dialog.message();
82
83      if (message === uniqueString) {
84        isVulnerable = true;
85      }
86
87      await dialog.accept();
88    });
89
90    const refererValue = 'http://example.com/?' + payload;
91
92    await page.goto(browsingUrl, {
93      timeout,
94      waitUntil: 'networkidle0',
95      referer: source.name === 'referer' ? refererValue : ''
96    }).catch(() => 'error');
97
98    // イベントの強制発火
99    if (browserType === 'chromium') {
100     await fireEvents(page);
101   }
102
103   await page.close();
104
105   return isVulnerable;
106 };
```

ソースコード A.8: src/printResults.js

```
1  const chalk = require('chalk');
2
3  module.exports = {
4    printPaths
5  };
6
7  function printPath(path, browserType) {
8    const { source, sink, type, stackTrace } = path;
9
10   if (source.key === null) {
11     process.stdout.write(`    Path\t${source.name} \u2192 ${sink}\n`)
```

```
12    } else {
13      process.stdout.write(`    Path\t${source.name} ${source.key} \u2192 ${
          sink}\n`)
14    }
15
16    //
        APIフックの実装の差から，FirefoxではChromiumよりコールスタックが深くなる
17    if ((browserType === 'chromium' && stackTrace.length > 1) || stackTrace.
        length > 2) {
18      const sinkLocation = browserType === 'chromium' ? stackTrace[1] :
          stackTrace[2];
19      const { lineNumber, columnNumber } = sinkLocation;
20
21      let filePath;
22
23      try {
24        const url = new URL(sinkLocation.fileName);
25        filePath = url.origin + url.pathname;
26      } catch (e) {
27        filePath = sinkLocation.fileName;
28      }
29
30      process.stdout.write(`    Location\t${filePath}:${lineNumber}:${
          columnNumber}\n`)
31    }
32
33    process.stdout.write('\n')
34  }
35
36  // 解析結果をいい感じに出力
37  function printPaths(vulnerabilities, possibleVulnerabilities, browserType)
      {
38    if (vulnerabilities.length === 0) {
39      process.stdout.write(`  [+] ${chalk.green('0')} vulnerabilities found\n
          `);
40    } else {
41      process.stdout.write(`  [+] ${chalk.red(vulnerabilities.length)}
          vulnerabilit${vulnerabilities.length === 1 ? 'y' : 'ies'} found\n`);
42
43      for (const path of vulnerabilities) {
```

```
44        printPath(path, browserType);
45      }
46    }
47
48    if (possibleVulnerabilities.length === 0) {
49      process.stdout.write(`  [+] ${chalk.green('0')} possible
            vulnerabilities found\n`);
50    } else {
51      process.stdout.write(`  [+] ${chalk.yellow(possibleVulnerabilities.
            length)} possible vulnerabilit${possibleVulnerabilities.length === 1
            ? 'y' : 'ies'} found\n`);
52
53      for (const path of possibleVulnerabilities) {
54        printPath(path, browserType);
55      }
56    }
57  }
```

ソースコード A.9: src/util.js

```
1  const crypto = require('crypto');
2
3  module.exports = {
4    generateRandomString,
5    parseCallStack
6  };
7
8  function generateRandomString() {
9    const bytes = crypto.randomBytes(32);
10   return bytes.toString('hex');
11 };
12
13 function parseChromeCallStack(error) {
14   const stack = error.split('\n').slice(1);
15
16   const parsedStack = stack.map(e => {
17     // トップレベルの呼び出しであれば, at test.html:11:9 のようになる(関数
         名がない)
18     // そうでなければ at g (test.html:11:9) のようになるので, カッコの有無
         でトップレベルの呼び出しか判断
19     const functionName = e.includes('␣(')
20       ? e.match(/at (.+?) \(/)[1]
```

```
21        : null;
22
23      let fileName, lineNumber, columnNumber;
24      if (functionName === 'eval') {
25        // evalからの呼び出しであれば，ファイル名は <anonmymous>ということにする
26        [fileName, lineNumber, columnNumber] = ['<anonymous>', 1, 1];
27      } else {
28        // トップレベルの呼び出しであれば，ファイル名などはカッコでなく at の直後に来る
29        [fileName, lineNumber, columnNumber] = e.includes('(')
30          ? e.match(/\(([^(]+):(\d+):(\d+)\)$/).slice(1)
31          : e.match(/at (.+):(\d+):(\d+)$/).slice(1);
32      }
33
34      lineNumber = parseInt(lineNumber, 10);
35      columnNumber = parseInt(columnNumber, 10);
36
37      return {
38        functionName, fileName, lineNumber, columnNumber
39      };
40    });
41
42    return parsedStack;
43  }
44
45  function parseFirefoxCallStack(error) {
46    const stack = error.trimEnd().split('\n');
47
48    const parsedStack = stack.map(e => {
49      // f@http://… のように，各行の先頭に関数名が来る
50      // トップレベルの呼び出しであれば，@http://… のようになる(関数名がない)
51      let [functionName, callLocation] = e.split('@');
52      functionName = functionName.length > 0 ? functionName : null;
53
54      // ファイル名，行番号，列番号は a.js:13:9 のようにコロン区切りで与えられる
55      // ファイル名には http://localhost:8000 のようにコロンが含まれる可能性もあるから，右側から区切る
56      let [fileName, lineNumber, columnNumber] = callLocation.match
```

```
        (/(.+):(.+):(.+)$/).slice(1)
57      lineNumber = parseInt(lineNumber, 10);
58      columnNumber = parseInt(columnNumber, 10);
59
60      return {
61        functionName, fileName, lineNumber, columnNumber
62      };
63    });
64
65    return parsedStack;
66  }
67
68  function parseCallStack(error, browser='chromium') {
69    if (browser === 'chrome' || browser === 'chromium') {
70      return parseChromeCallStack(error);
71    }
72
73    if (browser === 'firefox') {
74      return parseFirefoxCallStack(error);
75    }
76
77    throw new Error('unimplemented␣browser');
78  }
```