POLYTECH NICE-SOPHIA

UNIVERSITÉ CÔTE D'AZUR

Code-Reuse Attacks for the Web: Breaking Cross-Site Scripting Mitigations via Script Gadgets

Quentin Dubois, Vinh Faucher,
Florian Latapie, Elisa Roux

github.com/FlorianLatapie/PNS-SI5-S9_Web_Apps_Security_Article_Review

Mardi 30 janvier 2024

Code-Reuse Attacks for the Web: Breaking Cross-Site Scripting Mitigations via Script Gadgets

Rédigé par

- Sebastian Lekies (Google)
- Krzysztof Kotowicz (Google)
- Samuel Groß (SAP)
- Eduardo A. Vela Nava (Google)
- Martin Johns (SAP)

Sujets abordés : XSS, Code-reuse attacks, script gadgets

Définition : XSS

XSS : Cross Site Scripting

- Attaque très répandue
- Faille de sécurité

Principe :

Injection de code (JS/HTML) côté client qui sera interprété coté client par la suite

Différents types :

- Stored : script stocké sur le serveur victime
- Reflected : chargement de paramètres de l'attaquant (lien)
- DOM XSS : injection dans un élément qui sera consommé

DOM (Document Object Model) : représentation dynamique de la page

Mesures d’atténuations (mitigations)

Pour lutter contre ces attaques, plusieurs solutions possibles :

- **HTML Sanitizers :** Bibliothèques de “nettoyage” du HTML, ils suppriment ou transforment les éléments non fiables
- **Browser XSS filters :** Mis en œuvre par le navigateur, tente de neutraliser les attaques XSS
- **Web Application Firewalls :** Logiciel côté serveur filtrant et bloquant les requêtes malveillantes
- **Content Security Policy :** Fonctionnalité permettant de spécifier une politique de sécurité, notamment en permettant de différencier les scripts légitimes des scripts malveillants

Définition : Gadget

Gadget :
Ensemble d’instructions de code contenues dans une application

Code Reuse Attacks :
Enchainement de gadgets afin de modifier l’exécution de l’application

Script Gadget :
Fragment de code JavaScript légitime

Exemple de Script Gadget

Code disponible dans notre dépôt : js_examples/script-gadget/min.html

```
<div data-role="payload" data-text="alert('DOM XSS')"></div>
<script>
    const htmlMarkUp = document.querySelector('div[data-role="payload"]');
    if (htmlMarkUp) {
        const container = document.createElement('script');
        container.innerHTML = htmlMarkUp.getAttribute('data-text');
        document.body.appendChild(container);
    }
</script>
```

Contient une DOM XSS

Exemple de Script Gadget

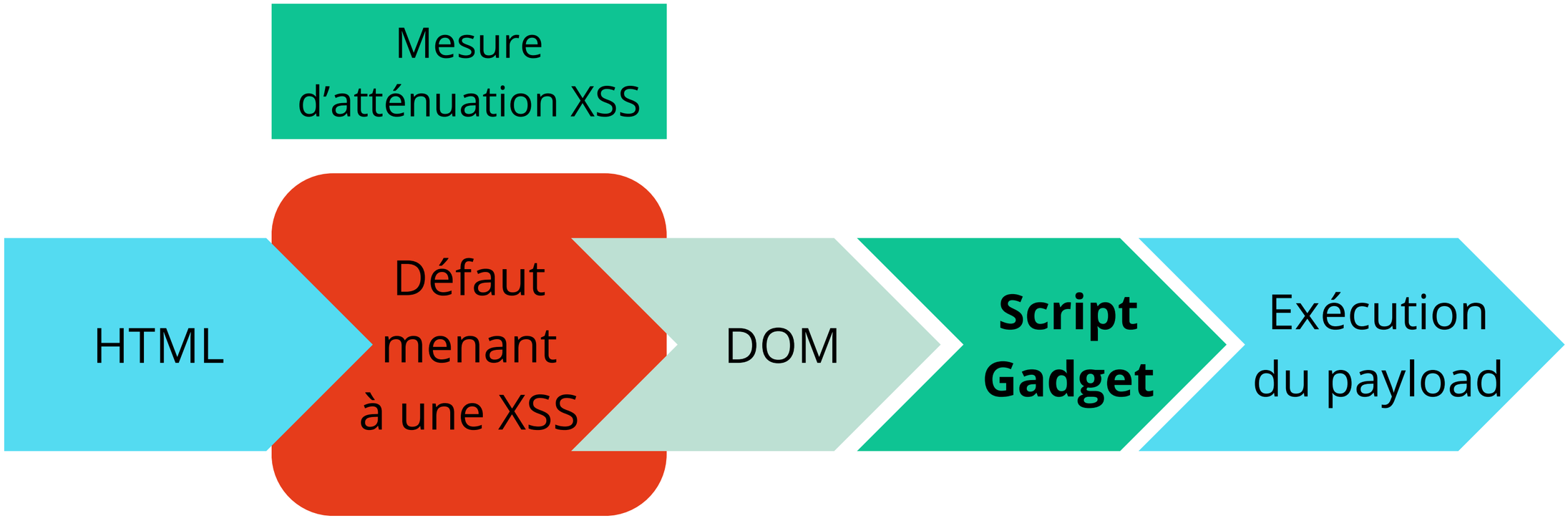
Code disponible dans notre dépôt : js_examples/script-gadget/min.html

```
<div data-role="pay                                                    /div>
<script>
    const htmlMarkU                                                    ole="payload"]');
    if (htmlMarkUp)
        const conta                                                    );
        container.i                                                    ta-text');
        document.body.
    }
</script>
```

LIVE DEMO !

Contient une DOM XSS

Modèle d’attaque



Types de gadgets

Dans l’article, les chercheurs ont identifié 5 types de gadgets :

- **JavaScript execution sink gadgets**
- **Gadgets in expression parsers**
- String manipulation gadgets
- **Element construction gadgets**
- Function creation gadgets

JavaScript execution sink gadgets

Code disponible dans notre dépôt : js_examples/underscore/index.html

JavaScript execution sink gadget : utilisation dans un DOM XSS execution sink

```
<div id="template">
    <% alert("underscore.js XSS") %>
</div>

<div id="output"></div>

<script>
    var template = _.template(document.getElementById("template").innerText);
    document.getElementById("output").innerHTML = template();
</script>
```

Appel à la bibliothèque **underscore.js**

JavaScript execution sink gadgets

Code disponible dans notre dépôt : js_examples/underscore/index.html

JavaScript execution sink gadget : utilisation dans un DOM XSS execution sink

LIVE DEMO !

```
<div id="template">
    <% alert("undersc
</div>

<div id="output"></di

<script>
    var template = _.template(document.getElementById("template").innerText);
    document.getElementById("output").innerHTML = template();
</script>
```

erscore.js

Expression based gadgets

Les frameworks comme Aurelia permettent d’utiliser le templating coté client :
Les pages sont affichées comme une interpolation des données à l’intérieur d’un template

```
<td>
  ${customer.name}
</td>
```

```
AccessMember.prototype.evaluate =
  function(...) { // ...
    return /* ... */ instance[this.name];
  };
```

```
<button foo.call="sayHello()">
  Say Hello!
</button>
```

```
CallMember.prototype.evaluate =
function(...) { // ...
  return func.apply(instance, args);
};
```

Expression based gadgets

Code disponible dans notre dépôt : aurelia/app.html

Gadget in expression parsers : spécifique aux frameworks, exécute du code

```
<div
    ref="me"
    $.bind="$this.me.ownerDocument.defaultView.alert('Aurelia XSS')"
>
</div>
```

Expression based gadgets

Code disponible dans notre dépôt : aurelia/app.html

Gadget in expression parsers : spécifique aux frameworks, exécute du code

```
<div
    ref="me"
    $.bind="$this.me.ownerDocument.defaultView.alert('Aurelia XSS')"
>
</div>
```

Attribut HTML bénin — `document` — `window`

Expression based gadgets

Code disponible dans notre dépôt : aurelia/app.html

Etude empirique menée

Décomposée en 2 parties

- *Qualitative et manuelle*

  → **Les script gadgets dans les bibliothèques/frameworks Javascript**

- *Quantitative et automatisée*

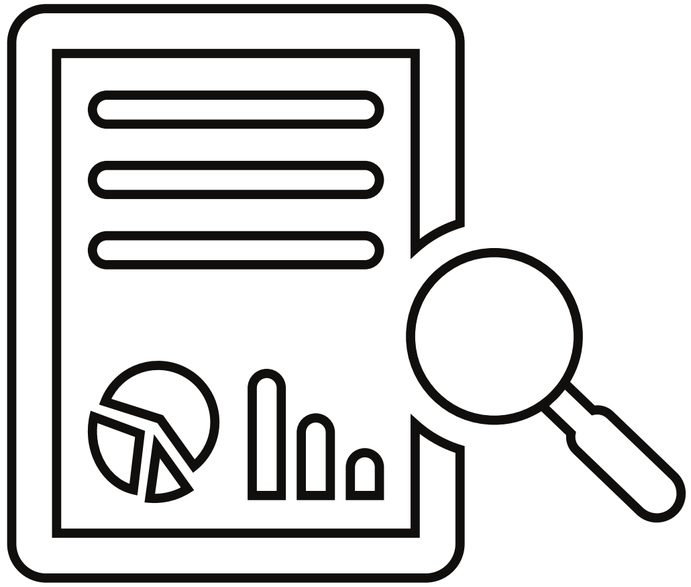
  → **Les script gadgets dans les bibliothèques/frameworks Javascript**

Script Gadgets dans les bibliothèques JS

Analyse des bibliothèques JS

16 bibliothèques populaires :

- AngularJS 1.x, Aurelia,
- Bootstrap,
- Closure,
- Dojo Toolkit,
- Emberjs, Knockout,
- Polymer 1.x,
- Ractive, React, RequireJS,
- Underscore / Backbone,
- Vue.js,
- jQuery, jQuery Mobile, jQuery UI



Script Gadgets dans les bibliothèques JS

Mesures d’atténuation testées :

- **CSP** : whitelists, nonces, unsafe-eval, strict-dynamiuc
- **WAF** : ModSecurity
- **Sanitizer HTML** : Closure, DOMPurify
- **Filtres XSS** : Chrome, Edge, NoScript

Résultats

CSP				XSS Filter			Sanitizers		WAFs
whitelists	nonces	unsafe-eval	strict-dynamic	Chrome	Edge	NoScript	DOMPurify	Closure	ModSecurity CRS
3/16	4/16	10/16	13/16	13/16	9/16	9/16	9/16	6/16	9/16

Script Gadgets dans le code utilisateur

Méthodologie

- **Dataset** : Top 5000 sites sur Alexa
- Utilisation d'un **taint tracking engine** créé par les chercheurs
  - Détection de flux de données
- Vérification par **simulation d'une Reflected XSS**
- **Générateur d'exploits** sous forme non exécutable

```
<div id="button"
    data-text="&lt;svg onload=verify()&gt;">
</div>
```

Résultats

Les techniques de “mitigation” XSS modernes ne sont pas à la hauteur

- Gadgets présents dans **82%** du dataset
- 285 894 gadgets vérifiés sur 906 domaines **(19,88 %)**
- Détection & Vérification conservatrice **(sans faux négatifs)**
- Le vrai nombre est bien plus haut

Résumé

- Mesures d’atténuation fonctionnent en **bloquant les attaques**
- Gadgets peuvent **contourner** ces mesures d’atténuations
- Gadgets sont **omniprésents** dans les **bibliothèques JS**
- Gadgets **existent** aussi dans le code source des **sites web**

Conclusion

La sécurité est complexe à mettre en œuvre:

- Rétro compatibilité pour ne pas casser les anciens sites
  → La sécurité n’est pas présente par défaut
- Un programmeur novice peut facilement ajouter des failles

Les fondations du web sont vulnérables

→ Réel besoin de changement au lieu d’ajouter des sécurité à postériori

  - Primitives d’isolation
  - DOM API Sécurisé : Trusted Type API

Trusted Types API

Force la sanitization avant l’ajout d’un élément dans le DOM (.innerHTML)

- Si la chaine n’est pas sanitized
  - Le développeur reçoit une *TypeError*
  - et l’élément n’est pas inséré
- Sinon
  - Il est ajouté

Cela permet d’éviter les erreurs d’oubli de nettoyage des données

La fonction de nettoyage à appliquer est à la charge du développeur

Demo Trusted Types API

```
    <meta http-equiv="Content-Security-Policy" content="require-trusted-types-for 'script'">
</head>
<body>
    <div id="myDiv"></div>

    <script>
        const escapeHTMLPolicy = trustedTypes.createPolicy("myEscapePolicy", {
            createHTML: (string) => DOMPurify.sanitize(string)
        });

        let el = document.getElementById("myDiv");
        const escaped = escapeHTMLPolicy.createHTML("<img src=x onerror=alert(1)>");
        //const escaped = "<img src=x onerror=alert(1)>";
        console.log(escaped instanceof TrustedHTML);
        el.innerHTML = escaped;
```

Toujours le même problème : CSP n’est pas activé par défaut

Demo Trusted Types API

```
    <meta http-equiv="Content-Security-Policy" content="require-trusted-types-for 'script'">
</head>
<body>
    <div id="myDiv"></div>

    <script>
        const escapeHTMLP                                                    olicy", {
            createHTML: (
        });

        let el = document
        const escaped = esc                                                  or=alert(1)>");
        //const escaped = "<img src=x onerror=alert(1)>";
        console.log(escaped instanceof TrustedHTML);
        el.innerHTML = escaped;
```

LIVE DEMO !

Toujours le même problème : CSP n’est pas activé par défaut