

Attaques Web - XSS

Cross-Site Scripting

Cross-site Scripting

- Remonte à la fin des 90s
 - mais toujours *très* présente
 - on estime **deux tiers** des sites webs vulnérables
- Regroupe plusieurs types d'attaques toutes basées sur le même principe
 - \Rightarrow injection de code dans une page web

Un peu d'histoire

- Terme *XSS* introduit par les équipes de sécurité Microsoft en 2000
 - on peut voir ça plus simplement comme une *injection JS/HTML*
- Première attaques importantes:
 - 2002: *XSS worm* (Hotmail)
 - 2005: *Samy (MySpace worm)*

XSS - Principe

- Injection de code (principalement HTML et JS) dans le but de le faire exécuter par le navigateur d'un autre utilisateur
- *Cross-Site*: le code passe en quelque sorte *à travers votre site* pour atteindre l'utilisateur
- Rendue possible lorsque le site envoie des pages qui contiennent des entrées utilisateur sans les avoir assainies

Attaques XSS typiques

- Scripts typiques d'attaque XSS permettront des attaques très diverses :
 - altération de page
 - lecture/écriture de cookies (*session hijacking...*)
 - requêtes additionnelles
 - téléchargement de *payload...*

Pourquoi est-ce si dangereux?

- L'origine d'un code JavaScript est définie par l'origine de la page qui l'exécute
- Conséquence: tout code injecté dans une page web est exécuté avec les droits de cette page
- C'est ce qu'on appelle la politique *same origin*

Rappels - Exécution de code JS

```
<!--Element HTML <script>-->
```

```
<script>  
  alert("Coucou !");  
</script>
```

```
<script src="https://monsite.fr/un_script.js">  
</script>
```

```
<!-- Attribut HTML d'événements-->
```

```
<input  
  oninput="alert('Coucou !')"  
  type="text">
```

Classification OWASP XSS

- **1. XSS réfléchi**
 - *payload* (charge utile, code malicieux) est injecté dans la requête HTTP
 - serveur **renvoie** («réfléchi») le *payload* dans la réponse HTTP
 - navigateur exécute le *payload* dans le contexte de la page
- **2. XSS stocké**
 - attaquant réussi à stocker le *payload* dans la BDD du serveur
 - serveur envoie le *payload* à tous les clients qui demandent une page qui charge dynamiquement le *payload*
- **3. XSS basé sur le DOM**
 - pas de serveur impliqué, tout se passe dans le navigateur (SPA)

XSS réfléchi

XSS réfléchi - Exemple PHP

- Requête:
 - GET /inscrire.php?email=pg@mail.com
- Code template:
 - Email d'inscription : <?= \$_GET['email'] ?>
- HTML résultant:
 - Email d'inscription : pg@mail.com

Exploitation de l'exemple

- Requête (via un lien posté sur un forum par exemple):
 - GET /inscrire.php?email=<script>alert("Yo !");</script>
- Code template (idem):
 - Email d'inscription : <?= \$_GET['email'] ?>
- HTML résultant contient le script:
 - Email d'inscription : <script>alert("Yo !");</script>

XSS réfléchi - Principe

- Code HTML/JS envoyé au serveur (*url string*, donnée de formulaire...) et **réinjecté dans la réponse sans assainissement**
 - typiquement un script JS `<script> ... </script>`
 - le navigateur exécute alors le script lors de la réception
 - d'où le nom : *réfléchi* (~ miroir)
- Problème (point de vue attaquant) : comment injecter sur une machine cible ?
 - \Rightarrow recours à l'ingénierie sociale

Autre exemple

```
<input type="text" name="recherche" value="<?= $_GET['motcle'] ?>">
```

```
<input type="text" name="recherche" value="" onmouseover="alert(1)">
```

XSS stocké

XSS stocké - Principe

- Idem mais cette fois le code est stocké au lieu d'être retourné immédiatement dans la réponse
 - BDD, fichiers, cookies, sessions...
 - le serveur est alors « miné »
 - potentiellement, il peut se passer des mois avant que l'attaque ne se produise finalement
- Une victime finit par récupérer le code infectée en accédant à la page minée
- Conséquences: beaucoup plus puissant car ne nécessite pas le recours à l'ingénierie sociale, et peut toucher toutes les personnes qui vont consulter une page minée

```
<?php
    $commentaire = $bdd->getCommentaire($idComm);
?>

<div class="commentaire">
    <p><?= $commentaire->texte ?></p>
    <p>Posté par <?= $commentaire->auteur ?></p>
</div>
```

Mon texte de commentaire inoffensif
Posté par <script>alert(1)</script>

XSS stocké - Patterns

- Rendu possible en général lorsqu'un service permet de «rentrer des informations» sur le serveur
 - Enregistrement d'un compte
 - Modification d'un profil utilisateur
 - Création d'une entité quelconque (commentaire, commande, etc.)
- Cela peut provenir d'une requête HTTP, d'un service REST (API), d'un import de BDD...

Les patterns de code injecté sont les mêmes qu'avec le XSS réfléchi. C'est juste que ce code a été entré dans le système par un biais qui lui permet de **rester** sur ce système.

XSS basé sur le DOM

DOM - Rappels

- DOM = Document Object Model
- Représentation du document HTML chargé dans le navigateur
- Permet de manipuler le document HTML avec JavaScript via une API

```
document.getElementById('monId').innerHTML = 'Nouveau texte';
```

XSS basé sur le DOM - Principe

- Le code malveillant (venant toujours d'une entrée quelconque) est interprété par un script JS **côté client**
 - faille côté code client (JS) et non côté serveur
 - évidemment, nécessite toujours que l'entrée ne soit pas assainie

Relation avec les autres types de XSS

- *XSS réfléchi*: la victime accède à un *lien* contenant le *code infecté* qui est renvoyé par le serveur
- *XSS stocké*: le *code infecté* est *stocké sur le serveur*, et renvoyé à la victime lorsqu'elle accède à une page
- *XSS basé sur le DOM*: le *code vulnérable* est *stocké dans la page*, souvent une application de type SPA chargée en une seule fois

Autre type de XSS - *mXSS*

- *mXSS*: *mutation-based XSS*, « XSS muté »
- Repose sur plusieurs constations:
 - du code HTML n'a pas à être 100 % correct pour être interprété (navigateurs « compensent »)
 - les navigateurs ont des interprétations différentes de ce qui est acceptable ou pas, de comment réagir à du code non standard
- Ce type d'attaque XSS peut être efficace contre les filtres

mXSS - Exemple

```
<img ="><script>alert('Coucou !')</script>">
```

XSS - Contre-mesures

- Validation/**assainissement** des entrées
- Utilisation de liste blanche pour les entrées autorisées quand c'est possible

Emplacement privilégiés pour attaques XSS

- Contenu HTML: `$entree`
- Attribut HTML: `<input value="$entree">`
- JavaScript: `var truc = '$entree';`
- Propriété CSS: `color: $entree;`
- URL: `https://lesite.com/lapage.php?truc=$entree`
- Surveiller les méthodes qui reçoivent en paramètres des données issues d'entrées utilisateur: est-ce la responsabilité de cette méthode de s'assurer de l'assainissement de ces entrées?

Protection des cookies

Cookies HTTPOnly

- Les cookies HTTPOnly sont inaccessibles au code JS client
 - un code JS malveillant ne pourra pas les voler
 - *Header* HTTP: Set-Cookie: user_id=1234; HttpOnly
 - les frameworks web permettent en général de mettre en place facilement un cookie HttpOnly

CSP - *Content Security Policy*

- Une CSP décrit quels types de ressources peuvent être utilisés ainsi que leurs sources autorisées
 - La CSP est envoyée dans le *header* (entête) de la réponse HTTP
 - ou bien dans le HTML HEAD (<head>) avec la balise <meta>
 - et donc à destination du navigateur

CSP - examples

- *Header HTTP*

```
Content-Security-Policy: script-src 'self'; object-src 'self';
```

```
Content-Security-Policy: script-src 'self' autresitedeconfiance.fr; object-src
```

- HTML <head>

```
<meta http-equiv="Content-Security-Policy" content="script-scr 'self'; object-s
```

CSP - Conclusion

- Les CSP peuvent contrôler l'accès à d'autres ressources, comme CSS, images, polices, audio, vidéo...
- Plus d'informations:
- <https://content-security-policy.com>
- Outil Google qui examine la CSP d'un site et propose des recommandations:
- <https://csp-evaluator.withgoogle.com>

Trouver des failles XSS dans le code
Analyse statique

Analyse dynamique vs. statique

- **Analyse dynamique** (par ex. avec *Burp Suite*):
 - audit de la page/application **en cours d'exécution**
- Il existe aussi l'**analyse statique**
 - audit du **code source** de l'application

Échappement en PHP

- En PHP, il existe une fonction `htmlspecialchars` qui permet d'*échapper* les caractères spéciaux HTML

```
htmlspecialchars($input);
```

```
// ex: `<script>alert(1)</script>` => `&lt;script&gt;alert(1)&lt;/script&gt;`
```

Avertissement

- Il faut bien étudier les fonctions d'échappement dans le langage utilisé et ne pas se contenter de «solutions de surface»
- Ex: Avant PHP 8.1, `htmlspecialchars` n'échappe pas les guillemets par défaut
 - cela peut conduire à des failles XSS si on utilise des guillemets simples dans les attributs HTML
 - il fallait utiliser `htmlspecialchars($input, ENT_QUOTES);`
- Cet avertissement vaut pour tous les autres mécanismes de sécurisation du code en général, et ce quel que soit le langage

Échappement en Symfony

- En Symfony, il existe une fonction `escape` qui permet d'*échapper* les caractères spéciaux HTML
- Le moteur de *templating* de Symfony (Twig) échappe automatiquement les variables injectées dans les templates
 - ex: `{{ valeur }}`
- Cependant, il existe un moyen de désactiver cet échappement (parce qu'on veut volontairement afficher du code HTML)
 - ex: `{{ valeur | raw }}`
 - dans ce cas, il faut s'assurer que la variable `valeur` ne contient pas de code malicieux (et notamment qu'elle ne provient pas d'une entrée utilisateur)

Échappement en Laravel

- Laravel/Blade se comportent exactement comme Symfony/Twig, sauf pour la syntaxe
 - `{{ valeur }}` (échappement automatique)
 - `{!! valeur !!}` (pas d'échappement, donc ce type de code est à auditer très soigneusement)

Échappement en ASP.NET Core

- ASP.NET/Razor
 - `@Model.UserName` (échappement automatique)
 - `@HttpUtility.HtmlEncode(Model.UserName)` (échappement manuel)
 - `@Html.Raw(Model.UserName)` (pas d'échappement)
- .NET propose également une méthode pour échapper les strings JS:
 - `@HttpUtility.JavaScriptStringEncode(Model.UserName)`

Échappement en Angular

- `<p>{{ valeur }}</p>` (échappement automatique)
- `<p [innerHTML]="valeur"></p>` (pas d'échappement)
- `bypassSecurityTrustHtml` et d'autres `bypassSecurity*` (pas d'échappement)
- `nativeElement.innerHTML` (pas d'échappement)

Échappement en React

- `<p> {this.valeur} </p>` (échappement automatique)
- `<p dangerouslySetInnerHTML={{ __html: this.valeur }}></p>` (pas d'échappement)
- `XSS` (échappement mais valeur à valider, peut contenir un lien malveillant)
- `ReactDOM.findDOMNode(n).innerHTML = "...";` (pas d'échappement)

Échappement en Vue.js

- `<p>{{ valeur }}</p>` (échappement automatique)
- `<p v-html="valeur"></p>` (pas d'échappement)
- `<p innerHTML=this.valeur></p>` (pas d'échappement)
- `<a :href="valeur">XSS` (pas d'échappement)

Attaques XSS - Conclusion

- Combinaison gagnante contre XSS:
- validation des entrées
- assainissement des entrées
- cookies HTTPOnly
- définition d'une *Content Security Policy*
- revue de code cyber et utilisation d'outils d'analyse statique

