DVWA (4/4) : Solutions, explications et étude des protections

Published by Ogma-sec on 14 septembre 2015

> **Note** : Les méthodes et procédés d'attaque expliqués dans ces articles ont pour objectif de vous faire comprendre les enjeux de la sécurité et l'importance de la protection du système d'information. Pour rappel, l'utilisation de ces attaques sur des systèmes réels est strictement interdite et passible de peines d'emprisonnement ainsi que d'amendes lourdes. Plus d'informations ici :

Dans l'article précédent, nous nous sommes intéressés aux failles de sécurité SQL Injection et SQL Injection Blind. Dans cet article, nous allons traiter des failles de type File Upload, Rejected XSS et Stored XSS.

Failles de type File Upload

Sécurité : Fonctionnement et impact d'une faille File Upload

Les failles de sécurité de type File Upload se caractérisent généralement par la possibilité d'envoyer des fichiers de façon un peu trop libre sur l'application web visée. Généralement, un formulaire d'upload de fichier est mis en place pour différentes raisons. Cela peut par exemple être afin d'uploader une image de profil, de partager des fichiers ou des images avec des collaborateurs, etc. Quoi qu'il en soit, **une faille de type File Upload est exploitée lorsqu'un pirate se sert de la possibilité de charger un fichier sur le serveur à des fins non prévues**, par exemple :

- Saturer l'espace disque du serveur avec des fichiers trop lourds
- Charger des scripts lui permettant d'exécuter des actions sur le serveur

En effet, il faut savoir que par « fichier » , on entend aussi bien des images ou des documents standard, **mais également des scripts et bouts de code** (PHP par exemple). Cela est très dangereux, car un attaquant qui arrive à uploader un *webshell* sur le serveur web ciblé possèdera alors une possibilité d'action importante.

Un webshell est un shell (terminal) permettant d'envoyer des commandes et de voir leur sortie via le service web. Il existe beaucoup de possibilités permettant de construire un webshell, cela dépendra souvent du langage utilisé par l'application web (Python, PHP, etc.). Il faut par exemple savoir qu'en PHP, la fonction « *system()* » permet d'exécuter des commandes shell à partir d'un script PHP, nous avons notamment vu cela dans la partie détaillant les failles « *Command Execution*« . Il s'agit donc d'une façon supplémentaire que peut avoir un attaquant pour exécuter du code côté serveur.

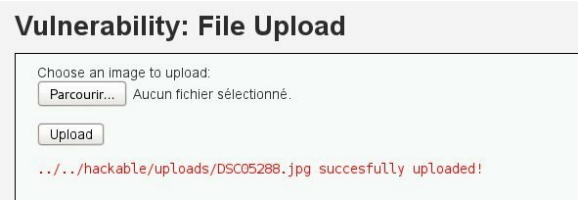
Quelles conséquences pour la sécurité ?

Un webshell sur un serveur web est très critique niveau compromission, en ayant la possibilité d'exécuter du code côté serveur, l'attaquant pourra **lire et extraire des données situées sur ce serveur et écrire d'autres fichiers, amenant alors à une exfiltration de données sensibles, souvent une élévation de privilège et parfois, une prise de contrôle totale du système**. Il existe nombre de sites web ayant été « défacés », on se rappelle notamment de pages provenant de groupes se revendiquant d'extrémistes religieux remplaçant les pages d'accueil des sites web par des textes religieux. **Dans d'autres cas, cela permet d'inclure des pages utilisées pour du Phishing, pour rendre le serveur membre d'un botnet ou pour diffuser des malwares.**

Il est donc important de se sécuriser et d'être très attentif à la sécurité concernant l'upload de fichier. Voyons à présent comment les formulaires d'upload de fichier peuvent être exploités.

DVWA — File Upload- level « low »

Le premier niveau est vraiment basique et va nous permettre de mieux comprendre l'enjeu en terme de sécurité lorsqu'un formulaire d'upload est vulnérable. Nous sommes donc en face d'un formulaire d'upload permettant, selon son auteur, d'uploader des images. En uploadant une image tout à fait banale, on se rend compte que celle-ci est stockée dans l'arborescence du serveur et plus précisément dans le dossier *../hackable/uploads/*:



Bien bien, essayons maintenant d'uploader quelque chose qui est tout sauf une image. **Un fichier texte standard par exemple !**

Cela fonctionne également. On peut donc s'apercevoir que, même si c'est une image qui est demandée, **le développeur nous fait confiance (à tord) et pense que nous allons uploader une image**. Cela nous fait

donc revenir au PHP, langage utilisé par l'application web sur laquelle nous travaillons. **Tous les fichiers en « *.php* » et contenant du code PHP valide seront exécutés par le serveur, que se passe-t-il si l'on décide d'uploader une nouvelle page web en .php sur le serveur ?** Essayons avec un fichier « *page1.php* » contenant le code suivant :

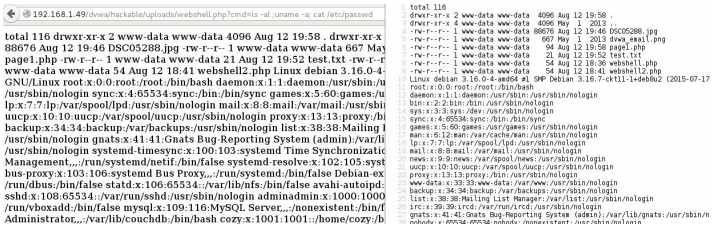
```
<?php
echo "Bonjour et bienvenue sur cette nouvelle page web ! Signé : un utilisateur ";
?>
```

On upload alors notre *page1.php* sur le serveur via le formulaire initialement destiné à uploader des images, puis l'on rend visite à notre fichier :

Nous voyons donc que notre page est bien présente et que notre code PHP est en plus exécuté. Cela nous permet de mettre en place un webshell.

> **Note** : Pour des raisons qui me sont propres, je ne délivrerai pas ici de code utilisable représentant un webshell PHP, vous en trouverez bien facilement ailleurs. Cela ne vous empêchera pas de comprendre l'enjeu et la dangerosité d'une faille de type File Upload.

Une fois un code de type webshell uploadé, voici une exemple de ce qu'il sera possible de faire :



On peut donc voir que, via mon webshell, qui est un script PHP au même titre que celui uploadé plus haut en démonstration, **je peux exécuter des commandes shell directement sur le système**. Cela permet à l'attaquant de parcourir l'arborescence du serveur, de lister ses utilisateurs et de commencer une phase d'élévation de privilège par exemple.

DVWA — File Upload- level « medium »

Passons au niveau supérieur, ici, l'upload d'un fichier avec l'extension « *.php* » est bloqué. On peut donc s'attendre à ce que le développeur, un peu plus prudent, ait pris quelques mesures afin d'éviter l'upload de fichiers autres que des images.

Si l'on essaie à nouveau d'uploader notre fichier texte, cela ne fonctionne pas et l'on obtient un « *Your image was not uploaded* » . Il faut savoir que, lorsque l'on envoie un fichier, plusieurs informations permettent de déterminer de quel type il est :

- Son extension, s'il s'agit d'un *.png*, d'un *.php* ou d'un *.txt* par exemple
- Le champ « *Content-type* » correspondant au type MIME et présent dans le header HTTP, il permet d'indiquer au serveur quel type de media est transporté. Côté serveur et code PHP, on peut récupérer ces informations sur le fichier transmis via le code suivant par exemple :

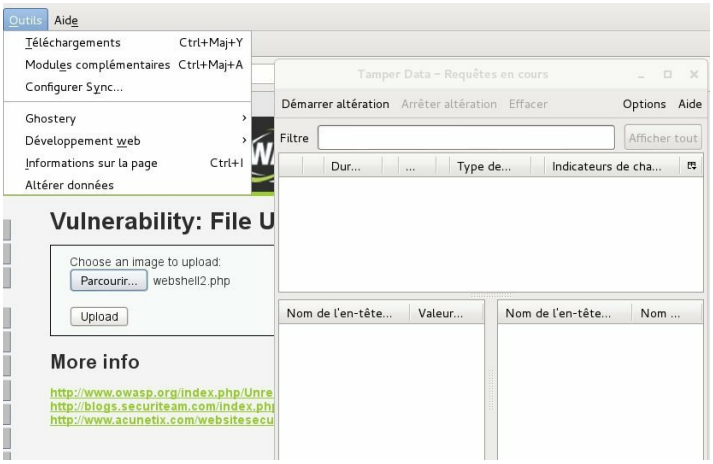
```
$uploaded_type = $_FILES['uploaded']['type'];
```

On peut alors indiquer que, si le type n'est pas précisément « image/jpeg » , on refusera le fichier en question. Il s'agit de la protection employée à ce niveau de DVWA.

Seulement, les champs de header HTTP d'une requête sont une donnée envoyée par le client, **cela est donc falsifiable**. Afin de vous montrer comment, nous allons utiliser le plug-in « *Tamper Data* » qui permet d'intercepter les requêtes émises par notre navigateur web pour nous donner la possibilité d'y changer des informations, comme les Header HTTP.

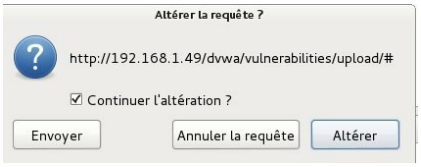
Vous trouverez ce plugin à cette adresse : https://addons.mozilla.org/fr/firefox/addon/tamper-data/

Rendez-vous dans « *Outils* » sur Firefox puis « *Altérer données* » , vous pourrez alors cliquer sur « *Démarrer altération* » et tenter d'uploader votre script PHP :

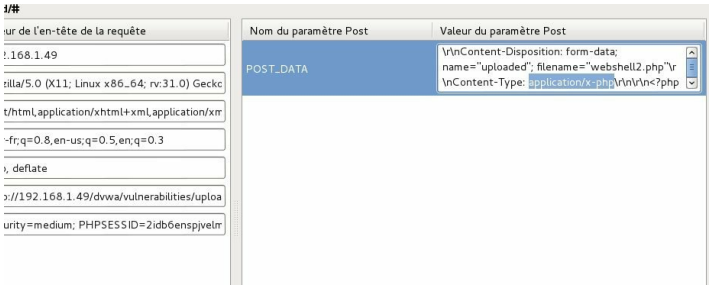


Une fois que vous aurez cliqué sur « *Upload* » , Tamper Data vous demandera si la requête est à envoyer, à

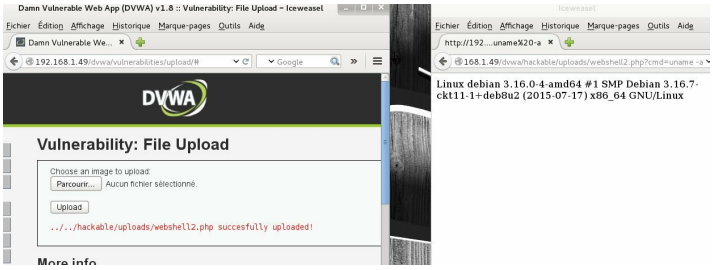
bloquer ou à altérer, on choisira cette dernière option et nous verrons alors les différents champs de notre requête, ceux-ci étant modifiables :



Entre autre, vous trouverez dans « *POST-DATA* » le contenu des données envoyées et parmi elles, le champ « *Content-Type* » :



Il suffira alors de changer « *application/x-php* » par « *image/jpeg* » afin de tromper le serveur web visé, on pourra alors cliquer sur « *Envoyer* » et notre webshell s'uploadera correctement.



Ici, nous avons tout simplement fait croire au serveur que le type de fichier envoyé était une image alors que c'était un script PHP, cela car il a fait confiance au champ « *Content-Type* » qui est un champ créé et envoyé par l'utilisateur (par son navigateur plus précisément).

Il existe d'autres moyens de contourner cette protection ainsi que d'autres outils utilisables comme *BurpSuite*, mais les enjeux restent les mêmes.

File Upload- Des pistes de protection

Plusieurs protections peuvent être mises en place pour protéger vos applications web de ce genre de faille.

Dans un premier temps, **il convient de ne pas proposer un formulaire d'upload si cela n'est pas nécessaire**. Dans les forums par exemple, il est courant de ne pas avoir la possibilité d'uploader d'image pour l'illustration dans les topics. D'une part car cela serait lourd, d'autres part car les utilisateurs ont appris à faire autrement (utiliser des plate-formes tierces). Le principe étant de mettre des formulaires d'upload uniquement si nécessaire et de ne pas en laisser traîner inutilement.

Il peut être intéressant de **désactiver l'exécution de script et l'interprétation de ceux-ci dans le répertoire où seront stockés les fichiers uploadés**. Ainsi, même si un fichier PHP est uploadé, il ne sera pas interprété par le serveur (son code ne sera pas exécuté). Également, il faut veiller à ce qu'un attaquant **ne puisse pas positionner son fichier uploadé dans un autre répertoire que celui indiqué**. Que ce passe-t-il si le pirate upload un fichier portant le nom « *../fichier.php* » dans un serveur Linux par exemple ? Pourra-t-il remonter l'arborescence afin de positionner son script un répertoire au dessus ?
Il convient également de vérifier un ensemble de données en provenance du fichier, le type MIME (header HTTP « *Content-type*«), extension et double extension, en tête du fichier sont généralement insuffisant pour faire confiance à un fichier mais méritent tout de même d'être vérifiés.

Source intéressante couvrant un ensemble de cas : https://www.acunetix.com/websitesecurity/upload-forms-threat/

Failles de type Reflected XSS

Sécurité : Fonctionnement et impact d'une faille Reflected XSS

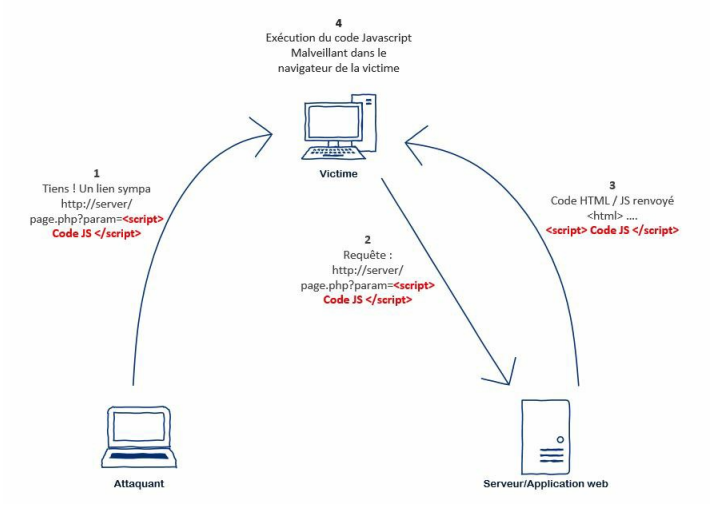
Les failles de type XSS sont parmi les failles les plus répandues dans les applications et sites web, elles sont d'ailleurs en troisième position dans l'OWASP Top Ten Vulnerability 2013 . **Une faille XSS va permettre à un attaquant d'exécuter du code JavaScript dans les navigateurs des utilisateurs des applications et sites web visés**, cela en jouant sur les données renvoyées par l'application web aux clients.

L'exécution et les possibilités de code *Javascript* permettent d'effectuer des actions malveillantes chez les victimes, qui ne sont dans ce cas pas les applications et les serveurs web, mais leurs visiteurs. Parmi ces possibilités, on retrouve le vol de cookies / sessions, le défacement de site web ou encore la redirection des utilisateurs vers des URL malveillantes (téléchargement de virus, phishing).

Il est toutefois à noter qu'il en existe deux types, les failles XSS réfléchies (comme un miroir) et les failles XSS stockées.

La particularité des failles XSS réfléchies est qu'elles ne sont pas permanentes, **cela signifie que le code malveillant que l'attaquant souhaite exécuter sur les navigateurs des victimes n'est pas stocké dans l'application web mais est propre à la requête/réponse entre le serveur web et la victime**. Nous avons déjà vu certains contextes où l'application web nous renvoie un mot ou une phrase que nous avons saisi dans un formulaire.

Le cas le plus classique étant un formulaire de recherche. Si l'on recherche tous les articles de « *John*« , la page retour contiendra quelque chose comme « *Voici les articles de John* » . On voit alors que la page affichée est propre à la requête (la recherche faite par l'utilisateur). Pour les failles XSS réfléchies, on retrouve ce fonctionnement car le code JavaScript à exécuter va être positionné par l'attaquant dans l'URL ou les paramètres que la victime va passer en requête au serveur. Voici un schéma de principe reprenant le fonctionnement d'une attaque *XSS réfléchie* :

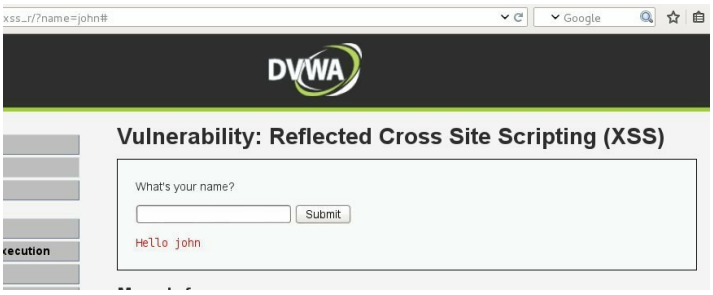


Ici, nous voyons donc que le pirate arrive à persuader sa victime d'aller cliquer sur un lien, ce lien est une URL exploitant une faille XSS sur l'application web en question et va permettre l'exécution de code JavaScript afin de, par exemple, récupérer son cookie de connexion afin de le renvoyer vers le pirate, ce qui est une des possibilités du JavaScript.

Ok, mais techniquement, comment ça marche ? C'est ce que nous allons voir dans les différents niveaux que propose DVWA.

DVWA — Reflected XSS — level « low »

On se retrouve donc avec un formulaire, on pourrait y insérer notre nom de façon tout à fait standard comme la plupart des utilisateurs le feraient. Un attaquant lui, **souhaitera profiter de ce système afin d'injecter du code, d'exploiter une faille XSS**, etc. On remarque que tout ce que nous saisissons dans le formulaire nous est renvoyé, de plus on remarque la présence de la chaîne saisie dans l'URL. Ici, j'ai saisie « john » dans le formulaire :



Que ce passerait-il si l'attaquant injecte des balises HTML dans sa saisie, par exemple afin de mettre en gras une certaine partie de la chaîne via les balises « *strong* » : « *john pas John* » . Voici le résultat obtenu :

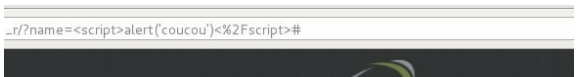
On remarque alors que la partie mise en gras via les balises HTML « *strong* » apparaît bien en gras. Le code (HTML) envoyé par l'utilisateur est donc interprété par l'application web.

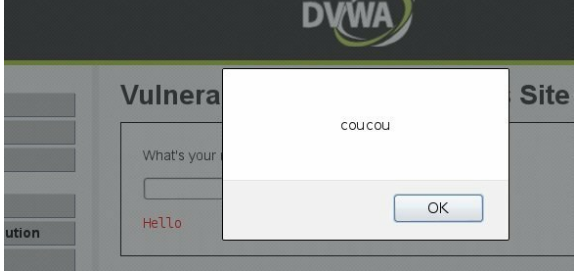
```
        <input type="submit" value="Submit">
    </form>

    <pre>Hello <strong>john</strong> pas John</pre>

</div>
```

Après le code HTML, on peut essayer du code JavaScript, on pourra alors effectivement parler de faille XSS, essayons avec le code suivant :

```
<script>alert('coucou')</script>
```

Voici ce que l'on pourra voir :





Nous voyons donc qu'ici, comme le code HTML, le code Javascript a été interprété et la fenêtre d'alerte JavaScript s'inscrit dans le navigateur de la victime. Afin de faire exécuter ce script à un autre utilisateur authentifié sur la même application web, il faudra arriver à le faire cliquer sur le lien correspondant à la génération de notre fenêtre JavaScript. L'intérêt d'une faille XSS n'est pas simplement d'arriver à afficher une pop-up, **il s'agit en réalité d'arriver à faire exécuter du code JavaScript exécutant un code écrit par le pirate dans le navigateur des clients.**

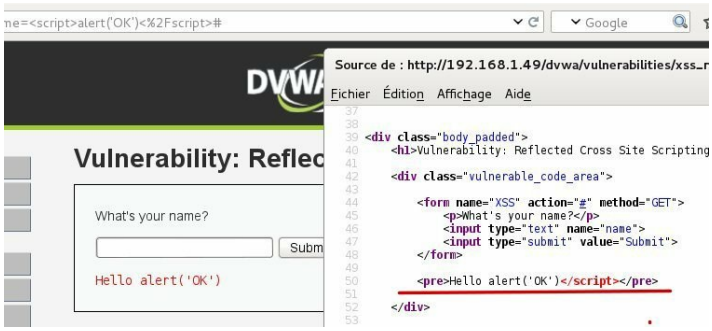
L'exécution de code sur la machine d'un utilisateur permet à l'attaquant d'avoir une possibilité d'action comme le vol de données. **Bien que ce soit ici l'application web qui soit vulnérable, ce sont tous ses visiteurs qui sont directement impactés.**

À noter que la manière la plus courante et la plus visible afin de tester la présence et l'exploitation d'une faille XSS est d'utiliser la fonction « *alert* » de JavaScript permettant la génération d'un pop-up comme sur la capture d'écran vue plus haut. Néanmoins, lors des phases réelles d'exploitation de ces failles, **le code JavaScript contient d'autres directives et est en général beaucoup plus discret, passant ainsi inaperçu aux yeux des utilisateurs et des administrateurs.**

Ce niveau était facile car aucune protection n'a été mise en place, il nous a néanmoins permis de voir concrètement le contexte d'exploitation d'une faille XSS.

DVWA — Reflected XSS- level « medium »

Comme à notre habitude dans DVWA, on passe au niveau au dessus et la méthode utilisée dans le niveau « *Low* » ne fonctionne plus. Essayons de comprendre pourquoi. Si l'on regarde dans le code source après avoir tenté un exploitation, nous pourrons voir cela :



Il semble que quelque chose ait disparu par rapport aux informations que nous avons envoyé, et plus précisément le « *< script >* » qui permet d'introduire du code JavaScript. Nous avons déjà vu un comportement similaire dans DVWA, notamment dans les protections contre les attaques « *File Inclusion* » . **La protection ici consiste à repérer la chaîne de caractère « *< script >* » puis à la remplacer par du vide si trouvée.** Nous allons donc opter pour la même technique de contournement : couper *< script >* en deux par un « *< script >* » qui sera supprimé, recomposant ainsi le premier entièrement avant de l'envoyer au serveur :

```
<scr<script>ipt>alert('OK')</script>
```

Ici, le « *script* » complet sera donc remplacé par du vide par le système de protection, recomposant ainsi le « *script* » coupé en deux. Voici ce que l'on obtient alors :

Si l'on regarde le code source, nous y retrouvons notre code JavaScript intacte :

```
<pre>Hello <script>alert('OK')</script></pre>
```

Nous voyons alors que la même technique de contournement a été utilisée pour deux contextes d'attaques différents : *File Inclusion* et *XSS*. Il est intéressant de noter ici que la protection en place peut sembler efficace au premier abord mais est en réalité contournable.

Reflected XSS — Des pistes de protection

Voyons ensemble quelques pistes de sécurisation contre les failles XSS, à noter que celles-ci sont valables

pour les deux types de failles XSS (*Reflected* et *Stored*).

Côté développement, il est encore une fois important de vérifier très précisément les données provenant des utilisateurs et clients web. Nous pouvons notamment voir que dans le niveau « *High* » , qui représente une implémentation sécurisée, on trouve quatre fonctions qui vont vérifier et sécuriser les entrées :

- trim : Prend une chaîne de caractères en entrée et y supprime tous les caractères invisibles (type espace, tabulation, saut de ligne).
- stripslashes : Prend une chaîne de caractères en entrée et y supprime les antislashs, qui permettent parfois de contourner une politique de sécurité.
- mysql_real_escape : Nous en avons déjà parlé dans les précédents articles, prend une chaîne de caractères en entrée et y échappe tous les caractères spéciaux comme les guillemets, apostrophes, caractère NULL, etc.
- htmlspecialchars : Nous en avons déjà parlé également, prend une chaîne de caractères en entrée et convertit tout ce qui ressemble à des caractères spéciaux en entités HTML, un chevron > deviendra &lt et ne sera alors plus interprété dans le code HTML, par exemple.

Ces fonctions sont donc utilisées sur le paramètre envoyé par le client avant tout affichage. **On parle d'une phase d'épuration (sanitization en anglais) des données.**

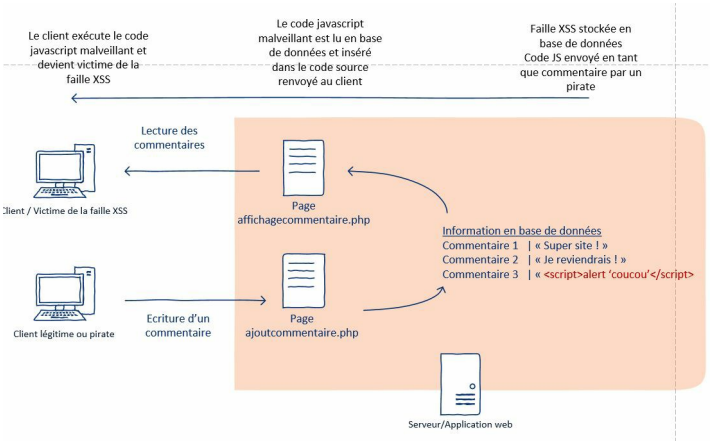
Côté utilisateur, il convient également de se protéger, par exemple, **on peut interdire toute exécution du JavaScript** grâce à des plugin comme NoScript ,bien que cela puisse perturber le rendu de certains sites web. Il peut également être intéressant d'utiliser des outils de réputation de site comme WoT (Web Of trust) qui pourra alors vous aider à repérer les sites de mauvaise réputation (donc potentiellement infectés).

Failles de type Stored XSS

Sécurité : Fonctionnement et impact d'une faille Stored XSS

Les failles de type XSS stockées possèdent un principe d'exploitation qui diffère légèrement des XSS réfléchies. **En effet, ici, le code JavaScript sera stocké dans l'application web, le plus souvent en base de données**. Cette faille est également appelée « *Faille du livre d'or*« , les commentaires ou livres d'or permettent en effet à des utilisateurs sans permissions spécifiques de stocker des informations dans la base de données du serveur, en temps normal des commentaires tout à fait standards.

Un commentaire envoyé à l'application web par un utilisateur sera donc stocké en base de données et affiché par la suite dans la page web. Pour illustrer tout ce fonctionnement, rien ne vaut un schéma :

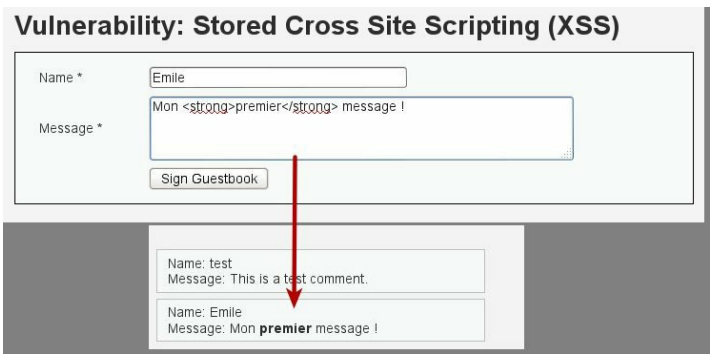


Nous avons donc au départ une page avec un formulaire permettant d'**ajouter des commentaires qui seront stockés en base de données**, une autre permettant de **lire les commentaires stockés en base de données**. Un attaquant a eu la joyeuse idée de stocker du code Javascript en base de données via ce formulaire, ce qui est possible quand aucune protection n'est mise en place. De ce fait, les commentaires affichés seront « *Super site !* » , « *Je reviendrai* » puis le code JavaScript qui sera lui interprété pour ce qu'il est dans le navigateur des visiteurs du site web : **du code exécutable**.

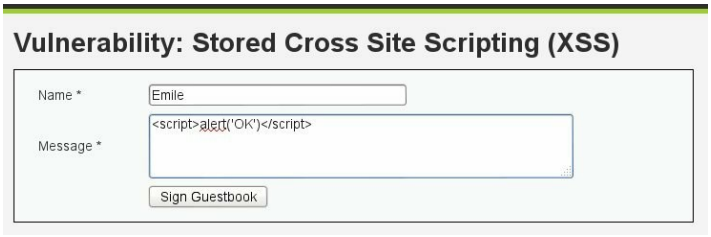
Cela permettra alors à l'attaquant d'exécuter du code JavaScript sur les navigateurs de tous les visiteurs du site web, on voit alors qu'une faille XSS stockée est plus impactante en terme de nombre de victimes qu'une faille XSS réfléchie.

DVWA — Stored XSS — level « low »

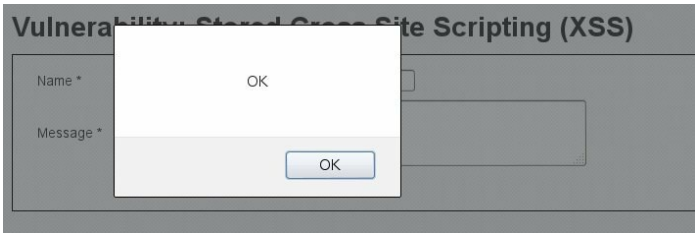
Ici, on se retrouve donc avec un formulaire permettant d'ajouter un commentaire, notre fameux livre d'or. Si l'on essaie de comprendre globalement le fonctionnement de la page, on peut voir que ce qui est saisie est ensuite stocké dans l'application web pour être ré affiché ensuite. **Nous avons donc la capacité de stocker des informations en base de données alors que nous ne sommes qu'un simple utilisateur**. Nous pouvons, comme la faille précédente, essayer d'afficher du texte en gras dans un premier temps afin de voir si le code HTML et plus précisément les chevrons et balises, sont interdits ou non :



Cela fonctionne, on passe maintenant au code JavaScript :



La subtilité ici est que le code JavaScript est maintenant stocké directement dans la base de données et sera réaffiché à chaque fois qu'un utilisateur viendra sur cette page, exécutant donc le code JavaScript malveillant.



On voit d'ailleurs notre code JavaScript dans le code source de la page renvoyée.

```
ts">Name: Emile <br />Message: <script>alert('OK')</script> <br /></div>
```

Petite astuce, pour vider la liste de commentaires, rendez vous dans « *Setup* » puis cliquez sur le bouton afin de réinitialiser la base de données.

DVWA — Stored XSS — level « medium »

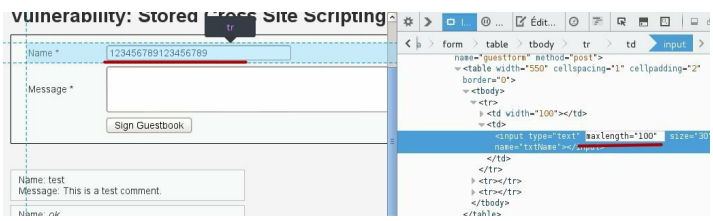
Passons au niveau supérieur, si l'on essaie à nouveau de réinjecter un code JavaScript comme celui déjà utilisé, cela ne fonctionne pas. Les balises injectées en question ne sont plus présentes lors de l'affichage. On peut donc supposer l'utilisation de la fonction PHP *htmlspecialchar()* qui est une protection efficace.

Si l'on est un peu curieux, on voit qu'il y a deux champs à saisir, ces deux champs doivent être à vérifier par le développeur. Précédemment, nous avons exploité le champ « *Message* » , intéressons nous au champ « *Name* » , celui-ci est peut être moins bien sécurisé par la phase « *sanitization* » établie par les développeurs.

La premier chose que l'on remarque si l'on essaie de rendre un code JavaScript dans le champ « *Name*« , c'est que l'on a pas assez de place. En effet, au bout de 10 caractères, nous ne pouvons plus écrire, nous n'avons donc pas assez de place pour injecter notre code. **Pour faire plus facilement notre test avec les balises HTML, on peut utiliser la mise en italique** via la balise « *i* » plutôt que la balise « *strong*« , trop grande pour entrer dans le champ « *Name* » , et l'on voit que cela marche, contrairement au champ « *Message* » :

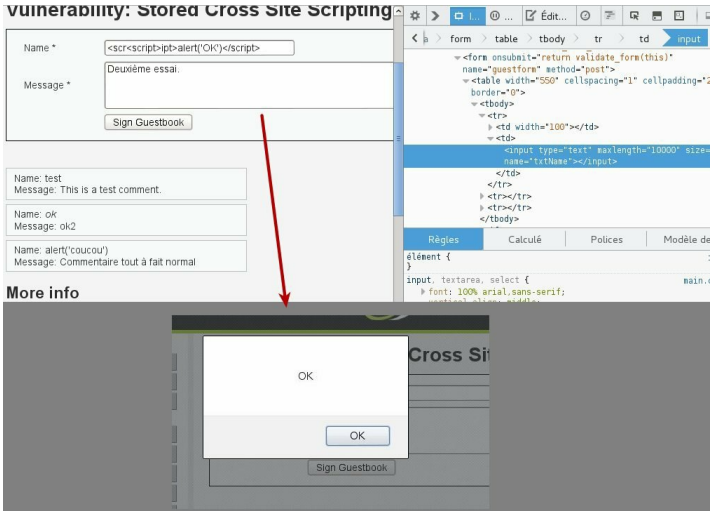
Cela valide donc bien le fait que le champ « *Name* » est moins bien protégé que le champ « *Message* » et que nous pouvons tenter d'y insérer un code JavaScript. Il ne nous reste plus qu'à trouver comment y saisir plus de 10 caractères.

Pour ceux qui ont déjà fait du HTML, cela ne devrait pas être dur à trouver. **La limitation interdisant de saisir plus de dix caractères est ici effectuée en HTML, dans le code source de la page, modifiable localement par n'importe quel client**. Il suffit donc d'accéder au mode développeur « *Ctrl Shift i* » sur les navigateurs Chrome et Firefox pour aller modifier localement le code source de la page, on pourra alors passer la limitation de 10 à 100 par exemple, ce qui devrait nous laisser une bonne marge :



Nous avons donc la main, en tant qu'utilisateur, sur cette limitation. Il ne nous reste maintenant plus qu'à insérer notre code JavaScript !

Et là, surprise, ça ne fonctionne toujours pas. En regardant le code source, ce qui est un bon réflexe à avoir lorsque l'on teste une faille XSS, on remarque que le premier « *script* » est de nouveau parti, comme dans le second niveau « *XSS reflected* » , on opte donc pour le même contournement :



Nous avons à nouveau réussi à contourner la sécurité en place concernant cette faille XSS.

Stored XSS — Des pistes de protection

Comme pour les failles de type XSS Reflected, **une protection efficace côté développement passe par l'utilisation de plusieurs fonctions PHP**. On retrouve d'ailleurs ces fonctions dans le code source du niveau « *Medium* » pour la protection du paramètre « *$message* » . L'erreur du développeur a été d'oublier de faire la même chose pour le paramètre « *$name* » qui n'est pas traité par la fonction *htmlspecialchar()*.

```
if(isset($_POST['btnSign']))
{
    $message = trim($_POST['mtxMessage']);
    $name    = trim($_POST['txtName']);

    // Sanitize message input
    $message = trim(strip_tags(addslashes($message)));
    $message = mysql_real_escape_string($message);
    $message = htmlspecialchars($message);

    // Sanitize name input
    $name = str_replace('<script>', '', $name);
    $name = mysql_real_escape_string($name);

    $query = "INSERT INTO guestbook (comment,name) VALUES ('$message','$name');";

    $result = mysql_query($query) or die('<pre>' . mysql_error() . '</pre>' );

}
```

Dans le cas du traitement de plusieurs variables, et afin d'éviter de les oublier, on peut passer par la création d'une fonction PHP. Ainsi, il ne suffira qu'à donner en paramètre les données envoyées par le client à cette fonction. On s'assurera donc qu'elles soient toutes protégées de la même façon.

Dans l'ensemble, on peut donc utiliser les mêmes protections pour les failles de type XSS stored et XSS reflected.

C'est la fin de ces quatre articles ! **Je félicite les courageux et/ou curieux qui les auront lu du début à la fin et j'espère que ceux-ci vous auront aidé à mieux comprendre les enjeux de la sécurité et l'importance de celle-ci dans le développement et la mise en place des applications web**. Je rappelle que les conseils concernant les protections à mettre en place ne sont pas complets, il existe certainement de meilleures méthodes. Également, les attaques et contournements de sécurité exposés ici sont tout à fait basiques et un attaquant déterminé pourra se montrer beaucoup plus ingénieux que cela.

Comme d'habitude, n'hésitez pas à me notifier de toute remarque, amélioration, note ou correction dans les commentaires !

Published in Challenges et CTFs

DVWA