DVWA (2/4) : Solutions, explications et étude des protections

Published by Ogma-sec on 17 août 2015

> **Note** : Les méthodes et procédés d'attaque expliqués dans ces articles ont pour objectif de vous faire comprendres 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 interdit et passible de peines d'emprisonnement ainsi que d'amendes lourdes. Plus d'informations ici :

Dans l'article précédent, nous avons présenté DVWA et vu son installation. Nous avons également étudié les attaques de type *brute force* et *command execution*. Dans cet article, nous verrons les failles *CSRF* et *File Inclusion*.

Failles de type CSRF

Nous commençons donc avec les failles de type CSRF.

Sécurité : Fonctionnement et impact d'une attaque CSRF

CSRF signifie *Cross-Site Request Forgery*. Cette attaque consiste en le fait de faire agir un utilisateur comme on le souhaite, par exemple en le faisant cliquer sur un lien d'un site amenant à une action sur un site en lui faisant faire une action que lui seul a la permission de faire, avec ses propres droits. La victime devient alors complice de l'attaque, à son insu bien évidemment.

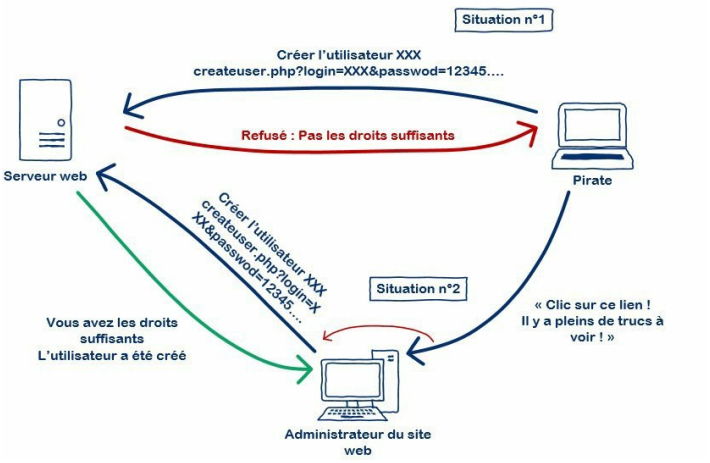
L'objectif même de l'attaque est généralement d'exécuter une action, exemple : Créer un utilisateur sur un site web.

Pour effectuer cette action, il faut disposer de certains droits, comme être administrateur de ce site web. La chose la plus simple reste alors de demander à notre administrateur de créer cet utilisateur, si nous ne sommes pas une personne de confiance, celui-ci refusera bien évidemment, **nous allons donc le forcer à le faire**.
Dans les contextes où nous sommes capable d'exploiter une faille de type *CSRF*, **nous pouvons savoir quelle URL et quels paramètres sont nécessaires pour créer un utilisateur sur le site web visé**, par exemple :

```
http://monserver.fr/createuser.php?login=patrick&password=123456&mail=patrick@mail.fr
```

Lorsque nous envoyons cette requête au serveur web depuis la position d'un client web standard (sans droits particuliers), le serveur nous rejette en nous annonçant que nous n'avons pas les droits nécessaires. En revanche si l'administrateur exécute la même requête alors qu'il est authentifié sur le site web, il va exécuter la création de cet utilisateur. **Il ne nous reste plus qu'à le piéger pour qu'il clique sur ce lien**. Voila ce qu'est une attaque de type *CSRF*.

Le concept de l'attaque peut ne pas être facile à comprendre. Voici un schéma qui présente ce type d'attaque :



Dans la situation n°1, **notre pirate essaie donc de créer un utilisateur mais se fait rejeter par manque de droits**. Dans la situation n°2, il envoie donc un piège à un administrateur de ce même site, un lien qui va lui faire exécuter une commande en direct du serveur visé et lui faire exécuter une action. **Cela peut être sur une page web de phishing ou un lien dans un conversation sur les réseaux sociaux/forum par exemple**.

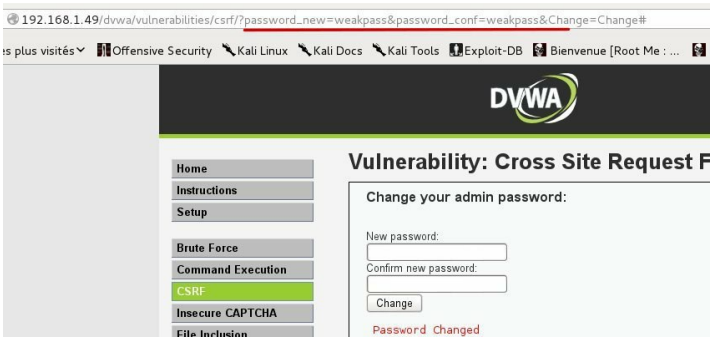
Une technique très utilisée en ce moment consiste à se servir **des raccourcisseurs de lien de type bit.ly pour dissimuler une URL** destinée à exécuter une action malveillante, en tant que victime, on ne peut alors absolument pas savoir où nous mènera ce lien et le fait de cliquer pour le savoir nous aura déjà fait exécuter l'action voulue par l'attaquant.
Voici quelques exemples des actions qu'il est généralement possible de faire via une faille CSRF :

- Créer, supprimer, modifier un utilisateur
- Changer un mot de passe
- Créer, modifier ou supprimer un article, un commentaire dans une application web
- Modifier les options d'une application web
- etc.

Si cela est encore obscure, rien ne vaut la pratique ! DVWA a été fait pour cela.

DVWA — CSRF — level « low »

On se retrouve donc devant un formulaire permettant de modifier le mot de passe de notre utilisateur. Si nous sommes connecté en tant que « *john* » et que nous remplissons ce formulaire, c'est le mot de passe de « *john* » qui sera changé, logique. On remarque, lorsque l'on saisi un mot de passe dans le formulaire et que l'on clique sur « *Change*« , **le mot de passe apparaît directement dans l'URL** :



Si l'on souhaite donc changer le mot de passe de l'utilisateur « *admin*« , il faut qu'admin exécute cette même requête. Nous allons donc chercher à le piéger pour qu'il exécute cette requête, croyant arriver sur un tout autre site, il atterrira en fait en cliquant sur notre lien sur son site web sur lequel il effectuera l'action du changement de mot de passe. **Cela car la requête est directement forgée pour effectuer cette action**. Une fois que l'admin aura lancé cette requête, nous pourrons, en tant que pirate, nous connecter avec son compte, cela car nous connaîtrons le nouveau mot de passe que nous lui avons fait changé : *weakpass* dans le cas de la capture d'écran.

Ici, nous pouvons profiter du fait que l'URL soit en clai et en « *GET* » afin de changer le mot de passe d'autres utilisateurs, sachez qu'une requête en POST ne poserait pas de difficulté majeure non plus :

```
http://192.168.1.49/dvwa/vulnerabilities/csrf/?
password_new=weakpass&password_conf=weakpass&Change=Change#
```

Si l'on arrive à faire cliquer un utilisateur sur ce lien, nous pourrons lui faire renouveler son mot de passe. Quelques exemples de code permettant de dissimuler un tel lien dans une page HTML :

```
<a href=http://192.168.1.49/dvwa/vulnerabilities/csrf/?
password_new=weakpass&password_conf=weakpass&Change=Change#> Voir ce lien pour plus
d'information </a>
<img src=http://192.168.1.49/dvwa/vulnerabilities/csrf/?
password_new=weakpass&password_conf=weakpass&Change=Change#>
```

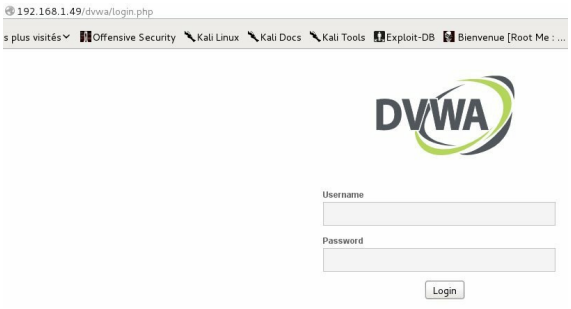
Dans le second cas (celui de l'image), le fait de simplement charger la page fera en sorte que le navigateur de la cible chargera le lien, croyant qu'il s'agit d'une image. Ajouter à cela le cas du raccourcisseur d'URL dans une conversation sur un réseau social.

Notez que dans notre cas, la faille peut être difficile à tester puisque nous n'avons personne à piéger. Pour vous rendre compte globalement de ce que cela fait, petite mise en situation :

- Déconnectez vous de *DVWA* via le bouton « *Logout* » puis rendez vous sur le site https://bitly.com/shorten/.
- À partir de là, nous pouvons générer une URL raccourcie à partir de l'URL utilisée pour piéger notre cible. On rentre donc dans le formulaire une URL comme celle-ci :

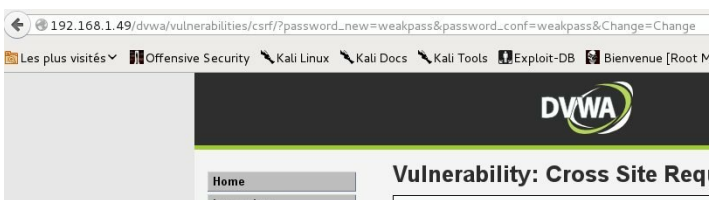
```
http://server_dvwa/dvwa/vulnerabilities/csrf/?
password_new=weakpass&password_conf=weakpass&Change=Change#
```

On se retrouve avec une URL raccourcie, par exemple :
http://bit.ly/1JIhxoB



En effet, nous n'avons pas les droits suffisants pour effectuer cette action.

- On se positionne maintenant dans la situation de l'administrateur, notre victime, et l'on se connecte à *DVWA*. Nous pourrons alors simuler le fait que nous nous soyons fait piéger en saisissant à nouveau le lien raccourci dans notre URL, on se retrouve alors avec cet affichage :

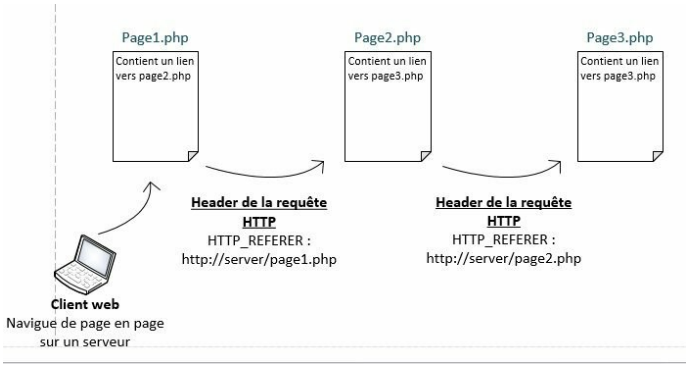


Ici, nous venons d'exécuter une action sur le site web : **Changer le mot de passe de l'administrateur**. Cela en cliquant sur un lien raccourci.

DVWA — CSRF — level « medium »

Passons au niveau supérieur, de nouveau, nous regardons que la technique utilisée dans le niveau « *Low* » ne fonctionne plus, des mesures de sécurité ont donc été prises.

Dans le code source, on remarque que le *HTTP-REFERER* est ici analysé. ***HTTP-REFERER* est un champ présent dans les Header HTTP permettant d'identifier l'adresse de la page ayant amené à la ressource demandée**. Si la page « *index.php* » contient un formulaire qui envoie les données à « *login.php*« , le *HTTP-REFERER* de la requête du client demandant l'affichage de « *login.php* » sera donc « *index.php*« . Mis à part le nom de la page, *HTTP-REFERER* peut également contenir IP ou URL, ce qui permet d'identifier une domaine ou une machine :



Dans le code source, on remarque que, si l'*HTTP-REFERER* n'est pas la machine elle-même, nous obtenons un message d'erreur. **Cela est donc une protection évitant que le formulaire d'où provient l'utilisateur vienne d'une autre machine** (autre que 127.0.0.1). En effet, recevoir une requête avec un *HTTP-REFERER* qui n'est pas le serveur web lui-même signifie clairement que c'est un autre site qui a poussé l'utilisateur sur l'application web visée. La vérification de l'*HTTP-REFERER* est en soit une idée intéressante car elle permet de voir d'où provient la requête.

Cependant, comme les variables, **les champs présents dans les en-têtes HTTP (header) peuvent également être modifiés manuellement par le client**. Nous pouvons par exemple faire cela avec *Tamper Data* ou un proxy type *BurpSuite* ou *OWASP-ZAP* si nous sommes en position d'interception des données. Cependant, ce n'est pas forcément le cas ici puisque c'est à un tiers d'exécuter la requête.

CSRF — Des pistes de protection

Côté développement, plusieurs protections peuvent être mises en place afin de se protéger des failles de type CSRF.

La protection la plus basique va être de demander la validation de l'utilisateur, par exemple une fenêtre « *Souhaitez-vous vraiment faire cela ?* » comme l'on peut en trouver sur *WordPress*. **Cela peut également être la nécessité de se ré-authentifier pour aller dans la partie administration**, c'est le cas notamment dans *PHPbb* qui demande à nouveau le mot de passe de l'utilisateur souhaitant aller changer les informations de son profil même si celui-ci est déjà authentifié. Enfin, il s'agit de la protection que l'on trouve dans le niveau « *High*« . Celui-ci demande en effet, avant l'envoi de la requête de changement de mot de passe, la saisie du mot de passe actuel de l'utilisateur courant, **il s'agit donc de valider son identité (dans le contexte où son mot de passe n'a pas été volé) et également de marquer un temps d'arrêt avant d'effectuer l'action en question**.
On peut également utiliser un système de token (jeton) unique qui serait assigné à l'affichage d'un page et généré de façon aléatoire. Ce token étant différent à chaque requête, il sera difficile pour l'attaquant de le deviner.

Côté utilisateur et utilisation quotidienne d'une application web, différentes habitudes sont à prendre afin d'améliorer la sécurité :

Penser à se déconnecter d'une application et ne pas se contenter de la quitter en fermant l'onglet. Se déconnecter en cliquant sur les boutons du type « *déconnexion* » permet de mettre fin aux cookies afin de provoquer leur expiration. Ainsi, si vous retournez sur le même site, aucun cookie ne pourra vous authentifier automatiquement.
Il n'est généralement pas recommandé de sauvegarder login et password dans son navigateur. Cela est toujours possible, quand un navigateur voit que vous vous authentifiez, il demande généralement la permission avant de sauvegarder ces informations. Dans d'autres cas, et pour éviter d'avoir à se souvenir de plusieurs dizaines de mots de passe, il est possible d'utiliser des coffres-forts de mots de passe qui permettent d'avoir à saisir un mot de passe avant que les champs login et password ne se remplissent automatiquement. Cela permet de marquer un temps d'arrêt et une action utilisateur avant l'authentification et donc de se rendre compte d'un éventuel piège.
Dans certains cas, il est recommandé de totalement différencier les navigateurs (voire les machines) pour les accès confidentiels et les accès standards. Cette technique est d'ailleurs recommandée par l'ANSSI. Cela consiste donc à totalement séparer les connexions par type, à utiliser un navigateur ou une machine pour les accès critiques (application métier, banque, …) et un autre pour les accès standards (réseaux sociaux, forums, …). Cela permettra en effet de casser l'utilisation du CSRF car l'application web visée (si elle est critique) ne sera pas connue du navigateur utilisé pour aller sur le réseau social ou celui utilisé pour ouvrir par défaut les pages web sur le poste par exemple. Pour plus d'informations, je vous invite à consulter cette documentation de l'ANSSI : Recommandations pour le déploiement sécurisé du navigateur Mozilla Firefox sous Windows

Ne pas utiliser l'HTML dans la lecture des mails, qui peut entraîner l'exploitation d'une faille CSRF rien qu'en lisant le mail. L'ouverture du mail provoquant l'interprétation du code HTML.

Failles de type File Inclusion

Passons aux failles de type « *File Inclusion*« .

Sécurité : Fonctionnement et impact d'une attaque File Inclusion

Une faille de type « *File Inclusion* » consiste à l'inclusion de fichier. Avec ça, vous avez tout compris !
Plus précisément, **cela consiste en le fait d'utiliser l'application et ses fonctionnalités afin de charger des pages et des fichiers spécifiques**, des fichiers internes aux serveurs qui n'ont normalement pas à être lus par les clients du serveur web (faille *LFI* pour *Local File Inclusion*), mais également des pages web externes au serveur web si le serveur web est configuré pour (faille *RFI* pour *Remote File Inclusion*).

Outre le fait que l'application puisse alors être utilisée afin de charger des pages et fichiers qu'elle n'est pas censée traiter, une faille de type *File Inclusion* peut amener à :

- Une exécution de code côté serveur, un code maîtrisé par le pirate, tant qu'à faire
- Un déni de service, si l'on essaie par exemple de faire charger une ressource très importante sur une page web
- La fuite d'informations sensibles, ce qui est notamment le cas lorsque l'on utilise **une faille de type File Inclusion pour lire des fichiers internes aux serveurs**

Détaillons un peu le concept même de l'attaque. Certaines applications sont construites de façon à recevoir en paramètre un fichier pouvant être exécuté, par exemple :

```
http://serveur/affichage.php?fichier=index.php
```

Ici, nous avons donc un script php « *affichage.php* » qui attend comme paramètre un nom de fichier. On imagine donc que si on lui demande d'afficher « *index.php* » ou « *index*« , il va aller chercher dans l'arborescence actuelle le fichier « *index.php* » puis l'exécuter pour l'afficher dans la page.

Le fait est que si on demande maintenant au script « *affichage.php* » d'afficher un fichier disponible sur un répertoire au dessus de lui-même, par exemple *../fichier2.html* (en supposant qu'il existe) :

http://serveur/affichage.php?fichier=../fichier2.html

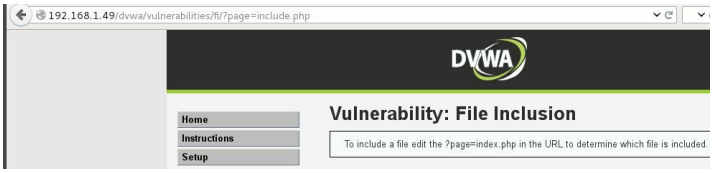
http://serveur/affichage.php?fichier=../../../../../etc/passwd

Par extension, une faille de type *RFI* consiste en la même opération sauf que l'on demande à « *affichage.php* » d'aller chercher une ressource (image, fichier texte ou page HTML) qui se situe sur un autre site. Il agira donc en tant que client web pour charger l'image et l'inclure dans son code de rendu.

Un peu de pratique pour mieux comprendre.

DVWA — File Inclusion — level « low »

On se rend donc dans le premier niveau « *File Inclusion* » de DVWA pour analyser ce qui se présente à nous, on voit donc un formulaire et une URL comme celle présente dans l'explication que j'ai faite plus haut :



Nous remarquons facilement une faille de type *File inclusion* comme celle-ci par la présence d'un nom de page ou de fichier en paramètre (après le « = »). Parfois, on peut trouver des implémentations permettant d'omettre le « *.php* » après le nom du fichier.
Ici, on peut donc essayer de demander à la page de charger un fichier qui se situe hors de l'application web :

```
http://192.168.1.49/dvwa/vulnerabilities/fi/?page=../../../../.../../../../etc/passwd
```

Comme je l'ai indiqué, certains serveurs permettent également l'inclusion du fichier distant, on parle alors de faille RFI. Pour cela, il faut que le serveur, et notamment PHP, permette le « requettage » de fichiers distants dans l'URL. Ce sont les options « *allow_url_fopen* » et « *allow_url_include* » qui permettent respectivement de prendre en charge le traitement des URLS contenant (*http://* ou *ftp://*) comme des fichiers (qui doivent donc être téléchargés) et leur inclusion dans l'ouverture de l'URL. Certaines applications peuvent avoir besoin de l'activation de ces options, mais ce n'est pas toujours le cas.

Quoi qu'il en soit, je vous laisse tester une faille RFI avec une page Google page exemple :

Puis une faille RFI demandant le chargement d'un très gros fichier, par exemple une image ISO, cela va avoir pour effet de bloquer la communication avec DVWA le temps que la requête soit traitée. On parle alors de déni de service, mais cela pourrait être expliqué plus en détail :

```
http://192.168.1.49/dvwa/vulnerabilities/fi/?page=http://cdimage.debian.org/debian-cd/8.1.0/amd64/iso-cd/debian-8.1.0-amd64-CD-2.iso
```

Dans des exploitations plus avancées, une faille LFI/RFI peut permettre l'exécution de code côté serveur. Nous n'irons pas jusque là dans nos exemples.

DVWA — File Inclusion — level « medium »

On enchaîne ! On passe maintenant au niveau supérieur dans lequel des mesures de sécurité ont été prises. On peut directement voir que si l'on tente l'exploitation de la faille LFI (chargement de fichiers internes au serveur), cela fonctionne toujours. Cependant si l'on essaie à nouveau d'exploiter la faille RFI, cela ne fonctionne plus. C'est donc ici que doit se situer la mesure de sécurité mise en place.

Analysons un peu la chose, un paramètre « *../../../etc/passwd* » est accepté et non un « *http://google.fr*« . Ils ont en commun des « / », des « . » et des lettres, on peut donc présumer que c'est « *http://* » qui alarme le système de protection. On peut d'ailleurs vérifier cela en lisant le code source. **Je vous rappelle qu'un attaquant n'aura normalement pas accès au code source PHP de votre application web**. Cependant, si vous utilisez des outils libres (open-source) et connus tel que WordPress, l'attaquant peut très bien faire de même et auditer l'outil en question en interne afin d'y trouver des failles qu'il exploitera ensuite sur votre application web.

On remarque donc que la fonction « *str_replace* » est utilisée pour remplacer « *http://* » par du vide, puis « *https://* » par du vide également. Voici le déroulement normal du processus :

- En entrée, on donne le paramètre : *http://google.fr*
- Il est traité une première fois et la ligne *$file = str_replace(« http:// », « », $file);* le transforme donc en « *google.fr*«
- Il est traité une deuxième fois mais rien ne change car « *https://* » n'est pas présent dans le paramètre donné
- On se retrouve donc avec « *google.fr*«

On peut casser ce mécanisme par le fait que la chaîne n'est vérifiée qu'une seule fois. Si j'envoie « *12http://34*« , j'aurai après les traitements de sécurité fait par *str_replace* la chaîne « *1234*« . Voyez plutôt :

En entrée, on donne le paramètre : *htthttp://p://google.fr*

- Il est traité une première fois et la ligne *$file = str_replace(« http:// », « », $file);* le transforme donc en « *http://google.fr* » car il a trouvé une fois « *http://* » en entier et le remplace donc par du vide, laissant « *htt* » *et p://* » former « *http://*« .
- Il est traité une deuxième fois mais rien ne change car « *https://* » n'est pas présent dans le paramètre donné
- Ici, on se retrouve alors avec « *http://google.fr*«

En somme, voici le paramètre à fournir pour contourner cette sécurité, le plus dur ici étant de détecter la sécurité mise en place sans visualiser le code source et d'ensuite comprendre le traitement des données qui est fait:

```
http://192.168.1.49/dvwa/vulnerabilities/fi/?page=htthttp://p://google.fr
```

Avec cette URL, la page de Google va également se charger, nous avons donc à nouveau l'exploitation d'une faille de type RFI : *Remote File Inclusion*

File Inclusion — Des pistes de protection

Après avoir vu le principe de fonctionnement et d'exploitation de ces failles *LFI* et *RFI*, partons à la recherche de méthodes de protection efficaces à mettre en place.
Si l'on va faire un tour du côté du niveau « *High* » qui est censé représenter un environnement non exploitable et sécurisé, on peut voir le code suivant :

File Inclusion Source

```
<?php

    $file = $_GET['page']; //The page we wish to display

    // Only allow include.php
    if ( $file != "include.php" ) {
        echo "ERROR: File not found!";
        exit;
    }

?>
```

Ici, nous remarquons donc que si le paramètre n'est pas exactement « *include.php*« , on obtient un message d'erreur « *ERROR : File not found!*« , il s'agit donc de ne laisser aucune possibilité quant au passage de paramètre. On peut d'ailleurs noter que dans ce cas, mettre la valeur directement dans le code (sans passer par un paramètre) aurait été plus rapide.

Si l'on doit quand même utiliser un tel fonctionnement mais avec plusieurs fichiers, on peut alors effectuer le même procédé avec plusieurs noms de fichier. **On opte alors pour une sécurité par liste blanche dans laquelle tout ce qui n'est pas explicitement autorisé est interdit**.

Nous avons également vu que, dans le cas de l'exploitation de la faille RFI, il fallait que les options « *allow_url_fopen* » et « *allow_url_include* » soient activées. Selon les applications web utilisées, **il peut parfois être inutile que celles-ci soient activées, il vaut mieux donc dans ces cas là les désactiver**.

Comme nous l'avons vu dans les protections envisageables contre les attaques de types *code execution*, **il peut être intéressant de décomposer, analyser puis recomposer tous les paramètres fournis par le client web**. Ainsi, on peut par exemple décomposer la chaîne sur laquelle nous avons travaillé en exemple afin de vérifier s'il n'y a bien qu'un seul point dans le nom du fichier, ou alors si la chaîne de caractère juste après le premier point est bien « php » est rien d'autre.

Concernant les failles de types *LFI*, il est possible de cloisonner les droits et les accès d'un serveur web afin que celui-ci ne puisse pas remonter une arborescence web au delà d'un certain point. Je décris notamment la mise en place de la solution « *apache-itk* » dans ce tutoriel technique, fait sur IT-Connect : Serveur web mutualisé (5/5) : Des pistes de sécurisation

Également, l'option PHP OpenBaseDir peut être utile afin de restreindre les accès d'un serveur web ou d'un vhost sur un serveur web.

En plus des protections qu'il est possible de mettre en place au niveau de la configuration du serveur web et des vhosts, il faut également être vigilant dès le développement de l'application web. **L'OWASP Testing Guide propose à destination des développeurs un ensemble d'ouvrages visant à aider à un développement sécurisé des applications web** : https://www.owasp.org/index.php/OWASP_Testing_Guide_v4_Table_of_Contents

C'est tout pour cet article, dans le prochain billet, nous parlerons des exercices concernant les failles SQL Injection.

N'hésitez pas à me notifier de toute remarque, amélioration, note ou correction dans les commentaires !

Published in Challenges et CTFs