DVWA (1/4) : Solutions, explications et étude des protections

Published by Ogma-sec on 11 août 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 interdit et passible de peines d'emprisonnement ainsi que d'amendes lourdes. Plus d'informations ici :

Concernant les challenges présents sur DVWA, il en existe plusieurs types et chacun d'entre eux proposent plusieurs « *niveaux* » avec des protections plus ou moins efficaces. À noter que **le dernier niveau « *High* » de chacune des failles n'est en l'état pas exploitable**. J'entends par là que les niveaux « *High* » présentent les cas où la sécurité est correctement implémentée, il ne sert donc à rien de passer des heures dessus, hormis si vous avez en tête de trouver une faille Zero-Day 🙂. **Nous nous appuierons entre autres sur ces exemples d'implémentations sécurisées pour présenter les pistes de sécurité à mettre en place**.

Voici les principales failles de sécurité présentent dans DVWA:

- Brute Force
- Commande Execution
- CSRF
- Insecure CAPTCHA (non traité)
- File Inclusion
- SQL Injection
- SQL Injection (blind)
- Upload
- XSS reflected
- XSS stored

Failles de type Brute Force

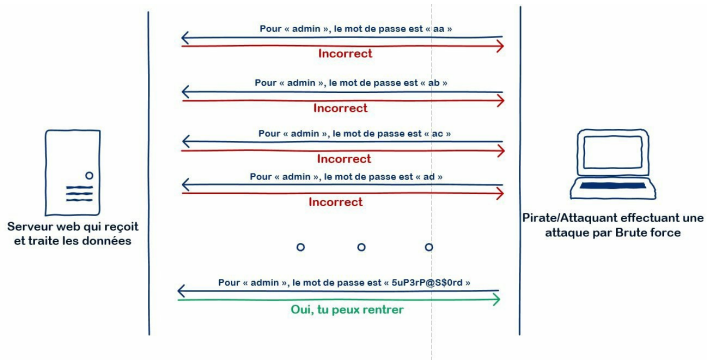
Attaquons nous tout de suite à la première étape : les failles de type *brute force*. Ici, il n'y a que les niveaux « *Low* » et « *High* » de disponibles.

Sécurité : Fonctionnement et impact d'une attaque par brute force

Les attaques de type « *brute force* » portent bien leur nom, **il s'agit de tester une très grande quantité de mots de passe pour un utilisateur donné**, en présumant (avec certitude si l'on dispose d'énormément de temps) que l'on tombera un jour sur le bon. La plupart des processus d'authentification aujourd'hui demandent au minimum deux informations :

- Un login
- Un mot de passe

Les logins peuvent être faciles à trouver, la plupart des applications et logiciels disposent souvent d'un compte d'administration « *admin*« , « *root* » ou « *administrateur*« . Dans d'autres cas, les logins des utilisateurs peuvent être trouvés en faisant quelques recherches, un nom de session Windows ou le nom des auteurs d'articles sur un blog (type *CMS, WordPress*), etc. **Une attaque par brute force est très simple à mettre en place lorsqu'il n'y a pas de protectoin**, on peut la schématiser comme suivant :



L'idée est donc que, si l'on peut proposer un nombre infini de réponse, on tombera forcément sur la bonne à un moment donné.

Il faut savoir qu'il existe deux types d'attaque par brute force :

- **Avec dictionnaire** : Dans ce cas, on se base sur une liste de mots, appelée dictionnaire, pour générer notre attaque. Plus précisément, nous allons par exemple prendre la liste des 100 mots de passe les plus utilisés et mettre cela dans un fichier. Pour chaque login connu, nous allons tenter de nous authentifier avec ces 100 mots de passe (un par un) en supposant qu'un utilisateur sera susceptible d'avoir mis un de ces mots de passe. Les dictionnaires peuvent être variés, être composés de noms propres à une application, un contexte, une entreprise ou au contraire être composés de tous les mots de passe dit « faibles » que les utilisateurs sont susceptibles d'utiliser (Exemple : Love, God, Soleil, password123, …).

- **Sans dictionnaire** : Ici, nous n'avons aucune information, il faut donc tester une à une toutes les possibilités, mélanger les caractères alphanumériques, les caractères spéciaux, etc. Par exemple, on va tester « *aa*« , puis « *ab*« , etc., puis passer à « *aaa*« , « *aab*« , une fois que toutes les chaînes à deux caractères auront été tentées. Cela fait bien entendu un nombre infini de possibilités qu'il est

possible de cadrer en fonction des informations dont on dispose, par exemple si l'on sait que l'application force l'utilisation de mot de passe entre 8 et 10 caractères sans caractères spéciaux.

Au delà d'utiliser un très grand nombre de possibilités, il est également courant que les mots de passe utilisés soient ceux par défaut. Alors, il est inutile de déployer les grands moyens, et tester les mots de passe par défaut les plus répandus suffit. Exemple : « *admin*« , « *password*« , « *root*« , « *toor*« , ou le nom de l'application.

On ne parle plus forcément d'attaque de type brute force dans ce cas, mais plus de « vulnérabilité » due à un mot de passe (très) faible.

Quel impact peut avoir une attaque de type brute force

Bien entendu, l'impact est **la découverte du mot de passe d'un utilisateur, cela peut donc entraîner une élévation de privilège pour l'attaquant**. Celui-ci passe en effet d'un statut où il n'a aucun droit à celui où il a les droits du compte dont il a pris le contrôle, cela est plus ou moins grave en fonction des droits assignés au propriétaire initial du compte, s'il s'agit d'un utilisateur « standard » ou d'un administrateur. **L'accès à un compte utilisateur est souvent une première base depuis laquelle est ensuite effectuée une élévation de privilège** durant laquelle un pirate va chercher à avoir toujours plus de contrôle sur la machine qu'il cible.

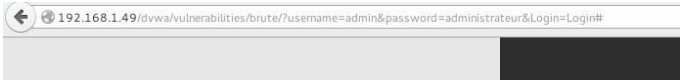
De plus, cela peut avoir des effets de bords importants comme le fait d'**utiliser le mot de passe trouvé dans d'autres applications** dans lesquelles le propriétaire initial aurait un compte ou **de repérer une logique dans la construction du mot de passe** (informations relatives à l'utilisateur, date de création du compte, …) pour reconstruire les mots de passe d'autres utilisateurs… Parmi les éléments les plus vulnérables aux attaques par brute force, on trouve bien entendu les applications et sites web, mais également certains protocoles comme SSH et FTP qui sont particulièrement exposés sur le net.

DVWA — Brute Force — level « low »

Comme je l'ai dit plus haut, nous pouvons dans un premier temps tester les mots de passe les plus basiques comme « *administrateur*« , « *admin*« , « *root*« , etc. Pour l'utilisateur, nous pouvons viser « *admin* » dans un premier temps. Nous pouvons faire cela à la main, mais cela ne vous apprendrait pas grand chose et il est plus intéressant de voir comment utiliser un outil pouvant également s'utiliser pour des tests beaucoup plus importants, j'ai nommé *Wfuzz* ! J'ai déjà rédigé un article sur cet outil : https://ogma-sec.fr/scan-dapplication-web-wfuzz/

Sur une machine *KaliLinux*, *Wfuzz* est présent par défaut, en fonction des paramètres qu'on lui donne, il va être capable de faire un grand nombre de tests de mot de passe à notre place. Pour cela, regardons un peu les informations que nous pouvons lui founir.

Dans un premier temps, nous pouvons voir que dans *DVWA*, les informations que nous saisissons atterrissent dans l'URL, nous savons donc que les paramètres sont transmis via la méthode GET (et non POST), ce qui rend encore plus simple une automatisation de l'envoi d'URL :



Nous savons donc que nous allons envoyer cette URL en modifiant le champ « *password* » pour tomber un jour sur le bon. Également, il nous faut un fichier dictionnaire composé des mots de passe les plus connus. Nous allons en créer un nommé « *password.txt* » par exemple, contenant les mots de passe suivants :

- pass
- pass123
- admin
- administrateur
- password

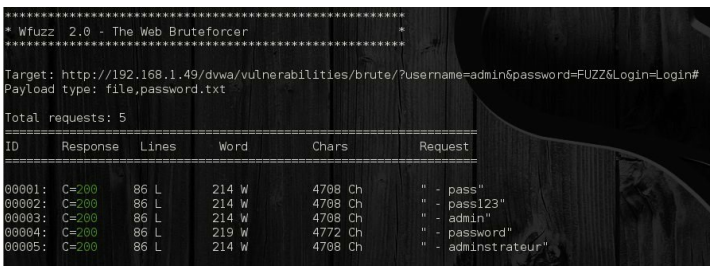
Vous pouvez bien entendu en ajouter autant que vous le souhaitez ou directement télécharger un dictionnaire de mots de passe sur internet. Une dernière subtilité est le fait d'avoir à préciser notre cookie dans la commande *wfuzz* et plus précisément dans le *Header HTTP* des requêtes que va passer *Wfuzz*. Voici la commande à utiliser :

```
wfuzz -H Cookie:PHPSESSID=htpbj0oeedtqdfituum9l33hk5 -c -z file,password.txt
"http://192.168.1.49/dvwa/vulnerabilities/brute/?
username=admin&password=FUZZ&Login=Login#"
```

Pour détailler rapidement les options utilisées :

- **-H** : Permet de spécifier le contenu du *Header HTTP* des requêtes envoyées, ici on précise donc le contenu de notre Cookie qui nous permettra de passer la page *login.php*. Le contenu du cookie est bien entendu à adapter
- **-c** : Permet d'avoir une sortie en couleur, optionnel
- **-z** : Permet de préciser la source de données, ici, il s'agit donc d'un fichier que l'on précise ensuite : « *password.txt*«
- Enfin, on spécifie notre URL avec les paramètres *GET*. Le mot *« FUZZ »* est celui qui sera remplacé à chaque requête par le contenu de notre fichier source, il s'agit donc ici d'une attaque par dictionnaire

Voici le retour que j'ai :



Il faut savoir que dans *Wfuzz*, les colonnes « *Lines*« , « *Word*« , et « *Chars* » sont respectivement le nombre

de lignes, mots et caractères contenus dans la réponse faite à notre requête. Au niveau des réponses attendues, il n'y a que deux possibilités :

- On nous indique que le mot de passe est bon : Le serveur nous renverra un certain message, toujours le même
- On nous indique que le mot de passe n'est pas bon : Le serveur nous renverra un autre message

Nous remarquons alors que les valeurs pour la ligne « *password* » ne sont pas les mêmes, on peut donc imaginer qu'il s'agit du mot de passe pour le compte « *admin*« . Chose que l'on pourra valider via un essai manuel.

Plus basiquement, **des essais « manuels » auraient également été rapidement concluants**. Il reste intéressant de voir comment *Wfuzz* peut nous aider à faire cela.

D'autres outils peuvent également être utilisés, tel que *BurpSuite*, qui agit comme un proxy entre le navigateur de l'attaquant et la page web visée afin de modifier le contenu de la requête ou de la réponse ou encore de rejouer plusieurs fois une requête en y modifiant des données (ce qui est appelé le *Fuzzing*).

> **Note** : Outre la vulnérabilité du mot de passe faible supposément présente, on remarque que l'URL utilisée pour s'authentifier est en GET, qui plus est en HTTP, ce qui rend très facile la capture des identifiants transmis sur le réseau via une attaque Man In The Middle et/ou une écoute réseau.

Brute Force — Des pistes de protection

Bien ! Nous avons vu ce qu'était une attaque par Brute Force, dont la nature et l'exécution peut prendre des formes diverses et variées, il ne s'agit pas toujours d'utiliser *Wfuzz* sur une page web. **Nous allons maintenant réfléchir aux outils et méthodes qui pourraient être utilisés pour améliorer la sécurité d'un processus d'authentification :**

- **Nous avons ici vu que le mot de passe trouvé durant notre attaque était un mot de passe faible**. Un mot de passe faible est soit un mot de passe contenant très peu de caractères, ce qui le rend rapide à trouver en cas d'attaque par brute force sans dictionnaire, soit un mot de passe très répandu tel que « *admin*« , « *password123*« , etc. Il est donc important de forcer les utilisateurs et administrateurs à utiliser des mots de passe forts, qui répondent à certains standards comme ceux que vous pouvez trouver ici : Recommandations de sécurité relatives aux mots de passe

- Également, il devient facile pour l'attaquant, à partir du moment où il n'a aucune contrainte à part le temps, de tenter un nombre infini de possibilités. Ainsi, il devient intéressant de **mettre un processus de sécurité en place bloquant toute possibilité d'authentification durant quelques minutes si plus de X échecs sont enregistrés pour un compte**. En effet, il est courant de voir cette sécurité en place car elle permet de ralentir grandement le travail de l'attaquant. Si l'on autorise que trois échecs puis que l'on bloque le compte pour 5 minutes, l'attaquant ne pourra tenter que trois mots de passe toutes les cinq minutes, ce qui nuit grandement au succès d'une attaque de type brute force, sauf en cas de mot de passe faible utilisé (« *password*« , « *admin*« , etc.).

> **Attention** : Ce processus peut toutefois être utilisé par l'attaquant afin de bloquer un compte volontairement, c'est pourquoi il faut privilégier les protections bloquant l'accès à l'application web (par exemple en bloquant l'IP de l'attaquant comme le fait Fail2ban) plutôt que ceux bloquant l'accès au compte.

- Nous avons également vu que **notre attaque était facilitée par le fait que nous connaissions déjà le login du compte à brute-forcer.** Il est alors judicieux de ne pas rendre les logins des utilisateurs visibles depuis l'extérieur. Dans certaines applications web comme *WordPress* pour citer la plus connue, **on peut par exemple différencier le nom d'affichage des auteurs d'articles de leur login de connexion**. Le fait d'avoir à trouver un login ET un mot de passe rend l'attaque encore plus complexe.

- **Une authentification à double facteur** : Ici, nous voyons bien que c'est le couple « login:mot de passe » qui fait apparaître une vulnérabilité. Par pure logique, tester toutes les possibilités existantes aboutira forcément un jour sur une combinaison correcte. Il faut donc réfléchir à utiliser, dans le processus d'authentification, **une information qui ne peut pas être brute forcée**. Pour rappel, l'authentification d'un individu peut se baser sur différents critères : ce que je possède (une clé spécifique, une carte magnétique), ce que je suis (mes attributs physiques, empreintes, iris de l'œil), ce que je sais (un mot de passe), ce que je sais faire (signature).

Il existe bien entendu quantité d'autres protections qu'il est possible de mettre en place afin de se protéger des attaques de type brute force, mais les plus basiques sont là.

Failles de type Command Execution

Passons maintenant aux attaques de type « *Command Execution* » ou « *exécution de commande*« , également appelées « r*emote code execution* » pour « *exécution de code à distance*« . En effet, c'est tout le principe de ce type d'attaque qui consiste à exécuter du code (PHP ou bash par exemple) tout en étant dans la position d'un client web : à distance.

Sécurité : Fonctionnement et impact d'une attaque Command Execution

Les failles de type « command execution » consistent en le fait qu'un attaquant puisse exécuter des commandes et plus précisément du code depuis l'application web ciblée. Le principe est donc que l'attaquant va utiliser les composants de l'application pour exécuter du code sur le serveur, code qu'elle n'est normalement pas censée exécuter. Cela est généralement intéressant pour un attaquant car il peut alors exécuter des commandes directement sur le serveur et utiliser cela comme un premier point d'entrée.

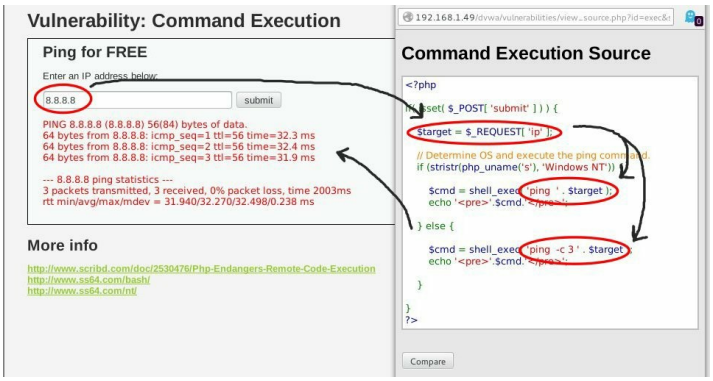
Il est à noter que dans le cas d'une telle attaque, les commandes sont exécutées avec les droits et les privilèges de l'application web qui les exécute. On voit alors grandement l'intérêt de faire tourner un serveur web avec des droits autres que ceux de « *root*« . L'exécution de commande à distance dans une application web tournant avec les droits de « *root* » permettrait à l'attaquant d'avoir directement tous les droits sur la machine cible. Si un compte avec des droits plus restreints est utilisé (exemple : *apache* ou *www-data*), l'attaquant n'aura que ces droits restreint via l'exécution des commandes.

L'exécution de commande et de code fait partie des failles citées dans le Top Ten Web Application Security Risk de l'OWASP. De manière générale, ces attaques sont exploitables car l'application web visée accepte de traiter des caractères spécifiquement forgés pour exécuter des commandes. Si le concept

est un peu obscure pour le moment, le premier niveau de DVWA devrait vous éclairer.

DVWA — Command Execution — level « low »

Ici, on nous propose donc un formulaire dans lequel on peut saisir une IP qui sera ensuite pingée par le système. La sortie du ping nous sera retournée. On peut donc deviner qu'une commande « *ping* » est lancée par un script PHP et que **le paramètre que nous saisissons dans le formulaire va être ajouté à cette commande**. Le tout sera exécuté sur le serveur et le retour de la commande affiché dans la page web. Cela nous est confirmé par la vue du code source qui est possible dans DVWA afin de mieux comprendre ce que nous faisons :



Nous voyons donc que le paramètre saisie est récupéré dans une variable appelée « *$target* » et que **cette variable est ensuite directement mise à la suite de la commande « *ping*«** (qui diffère légèrement sous Linux et sous Windows), exécutée avec la fonction PHP « *shell_exec*« .

Sous Linux, il faut savoir qu'il est possible d'enchaîner plusieurs commandes en une ligne, notamment avec des caractères spéciaux comme « *;*« , « *&&* » ou « *||*« . Je détaille cela dans un de mes articles sur IT-Connect : L'enchaînement des commandes sous Linux

Ici, nous pouvons exécuter n'importe quelle commande. On peut rapidement comprendre que ce que l'on met dans le formulaire est ensuite intégré dans une commande de type shell. Il faut alors jouer d'une certaine syntaxe possible afin de passer plusieurs commandes en une seule ligne. Par exemple :

```
8.8.8.8
```

Deviendra dans la fonction *shell_exec* (fonction PHP exécutant des commandes systèmes) :

```
ping -c 3 8.8.8.8
```

Ainsi si l'on décide d'ajouter une commande avec un caractère d'enchaînement tel que « *;* » :

```
8.8.8.8 ; uname -a
```

Deviendra

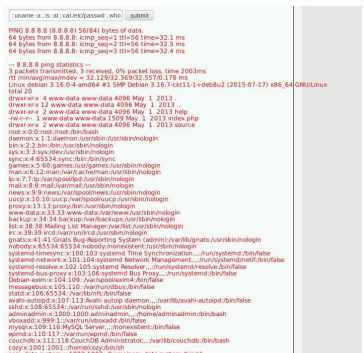
```
ping -c 3 8.8.8.8 ; uname -a
```

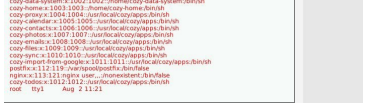
Et voilà, ici **le pirate a donc réussi à utiliser une fonctionnalité de l'application web pour exécuter une commande système qui n'était pas prévue au départ**. À noter qu'ici, nous pouvons exécuter n'importe quelle commande que l'utilisateur « *www-data*« , qui fait tourner l'application web, est capable d'exécuter :

À noter que l'on peut très bien enchaîner plusieurs dizaines de commandes, tant qu'elles restent sur la même ligne :

```
8.8.8.8 ; uname -a ; ls -al ; cat /etc/passwd ; who
```

Résultat :





DVWA – Command Execution – level « medium »

Passons au niveau supérieur, on peut s'attendre à ce que des mesures de sécurité supplémentaires aient été mises en place, et effectivement, les commandes passées dans le niveau *« Low »* ne fonctionnent plus. Si l'on est un peu curieux et que l'on s'intéresse à tous les caractères d'enchaînement de commande sous Linux que sont « ; », « && » et « || », on remarque que le dernier est un peu spécifique. En effet, **il permet d'exécuter une commande uniquement si la première commande retourne une erreur**. Exemple :

```
cd /dossier1 || mkdir /dossier1
```

Ici, si le dossier */dossier1* n'existe pas, je le crée. S'il existe, alors la seconde commande n'est pas exécutée.

Dans le code source, on remarque qu'il manque l'exclusion du caractère d'enchaînement des commandes « || » qui permet d'exécuter une commande que si la précédente ne se déroule pas correctement. On peut donc volontairement faire échouer le *ping* pour passer à l'exécution de notre commande tierce, cela en indiquant une adresse IP faussée par exemple :

```
8888.8.8.8 || ls
```

Voici le résultat :

Nous avons donc contourné la sécurité mise en place dans ce niveau.

Command Execution — Des pistes de protection

Si l'on regarde le second code source, on remarque que des vérifications ont été faites au niveau du paramètre envoyé par l'utilisateur :

Command Execution Source

```
<?php

if( isset( $_POST[ 'submit' ] ) ) {

    $target = $_REQUEST[ 'ip' ];

    // Remove any of the charactars in the array (blacklist).
    $substitutions = array(
        '&&' => '',
        ';' => '',
    );

    $target = str_replace( array_keys( $substitutions ), $substitutions, $target );

    // Determine OS and execute the ping command.
    if (stristr(php_uname('s'), 'Windows NT')) {

        $cmd = shell_exec( 'ping  ' . $target );
        echo '<pre>'.$cmd.'</pre>';

    } else {

        $cmd = shell_exec( 'ping  -c 3 ' . $target );
        echo '<pre>'.$cmd.'</pre>';

    }
}

?>
```

En effet, la fonction « *str_replace* » est ici présente afin de remplacer les caractères « *&&* » ou « *;* » par du vide. **La conséquence est donc que l'on ne peut plus utiliser ces caractères pour enchaîner les commandes**. Le principe est donc de créer une « *blacklist* » de caractères qui ne devront absolument pas arriver jusqu'à la phase d'exécution. Cependant, **il est relativement difficile de prendre en compte toutes les possibilités**. On voit que pour ce niveau, le développeur a oublié le caractère « *|*|« , par manque de connaissance par exemple. Le principe étant que l'on trouve toujours plus malin que nous.

Il est préférable dans de tels cas, où l'entrée (que l'on nomme « *input »)* est utilisée pour exécuter des commandes systèmes, de décomposer la chaîne pour vérifier chacun de ces éléments. Nous savons que la seule valeur attendue est une adresse IP, qui pour le coup est composée d'éléments simples :

- 4 nombres entre 0 et 255
- 3 points qui les séparent

Le principe utilisé dans le niveau « *High* » est donc le bon, il consiste à **découper l'input et séparer chacun des composants du paramètre fournis par le client web pour vérifier chacun de ces composants individuellement, puis de recomposer le tout avant exécution**. Cela permet donc de simplifier la vérification, mais également de casser toute tentative incluant des caractères non prévus. Le fait de vérifier si un paramètre est un nombre entre 0 et 255 étant relativement facile, voici le code source sécurisé proposé :

```
<?php
if( isset( $_POST[ 'submit' ] ) ) {
    $target = $_REQUEST["ip"];
    $target = stripslashes( $target );

    // Split the IP into 4 octects
    $octet = explode(".", $target);
```

```
    // Check IF each octet is an integer
    if ((is_numeric($octet[0])) && (is_numeric($octet[1])) && (is_numeric($octet[2])) && (is_numeric($octet[3])) && (sizeof($octet) == 4)  ) {

    // If all 4 octets are int's put the IP back together.
    $target = $octet[0].'.'.$octet[1].'.'.$octet[2].'.'.$octet[3];

        // Determine OS and execute the ping command.
        if (stristr(php_uname('s'), 'Windows NT')) {

            $cmd = shell_exec( 'ping  ' . $target );
            echo '<pre>'.$cmd.'</pre>';

        } else {

            $cmd = shell_exec( 'ping  -c 3 ' . $target );
            echo '<pre>'.$cmd.'</pre>';

        }

    }

    else {
        echo '<pre>ERROR: You have entered an invalid IP</pre>';
    }


}

?>
```

Pour empêcher les failles de type « *Command Execution*« , il est important de prendre un maximum de précautions en sécurisant les informations et données envoyées par les clients web. Également, **il est important de faire en sorte que l'application tourne avec seulement les droits nécessaires**, réduisant ainsi l'impact d'un éventuel contournement de la sécurité pouvant amener à l'exécution de commande.

Enfin, les commandes PHP permettant l'exécution de commande *bash* sur un système sont généralement à bannir. Il existe une valeur dans la configuration *php.ini* permettant de lister des fonctions qui deviendront non utilisables : *disable_functions*

Voici une document donnant quelques exemples d'utilisation de cette option PHP : Disable PHP Functions Using disable_function

Ce point est d'ailleurs traité à la référence « *OTG-INPVAL-013* » de l'OWASP Testing Guide, la version en ligne est disponible ici : Testing for Command Injection

À noter que pour certaines applications, la désactivation de certaines fonctions PHP peut poser problème, il est alors utile de faire quelques tests avant mise en production.

La suite dans la prochaine partie ! Nous traiterons des autres failles présentées dans DVWA

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

Published in Challenges et CTFs

DVWA