DVWA (3/4) : Solutions, explications et étude des protections

Published by Ogma-sec on 31 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 interdite et passible de peines d'emprisonnement ainsi que d'amendes lourdes. Plus d'informations ici :

Dans le précédent article, nous nous sommes intéressés aux failles de sécurité CSRF et File Inclusion. Dans cet article, nous allons traiter des failles de type SQL Injection et SQL Injection Blind.

Failles de type SQL Injection

Les failles de type SQL Injection sont parmi les failles les plus courantes et les plus dangereuses dans le domaine des applications web. **Les injections SQL sont notamment dangereuses car elles permettent d'accéder, dans certains contextes, à la quasi totalité des informations contenues dans la base de données d'une application web**, nous en entendons souvent parler lors de piratages ayant entraîné des exfiltrations de données, notamment des données utilisateurs (noms, mails, numéros de carte bleu, mots de passe).

Sécurité : Fonctionnement et impact d'une attaque SQL Injection

Nous allons ici essayer de comprendre quel est le principe global d'une injection SQL. Pour ceux ayant suivi cette suite d'articles depuis le début, on retrouve une idée similaire avec les failles de type *Command* ou *Code Exection*, mais cette fois-ci appliquée spécifiquement aux requêtes envoyées à la base de données par l'application web ciblée.

Lorsqu'une application web cherche des informations à traiter et/ou à afficher, elle va la plupart du temps les chercher dans une base de données, locale ou distante. Alors, elle est obligée de passer par une ou plusieurs requêtes SQL (Structured Query Language). Par exemple si, dans un formulaire de recherche, je cherche tous les articles de l'auteur « *bob*« , je vais envoyer une URL comme celle-ci au serveur web :

```
http://serveur.fr/search.php?s=bob
```

Le serveur web va alors utiliser ce paramètre afin d'effectuer une recherche dans une base de données, qui contient un ensemble d'informations sur les articles et leurs auteurs, une requête de ce type :

```
SELECT auteur, article FROM site.articles where auteur = "bob" .
```

La requête obtient alors un ensemble de données en retour, données qui vont être mises en forme par PHP et retournées sous forme de page HTML au client web.

Le principe d'une injection SQL va donc être d'**utiliser la requête SQL faite par le code PHP afin de lire ou d'écrire des informations en base de données**, cela n'étant évidemment pas prévu par le développeur de l'application web.

> **Note** : Pour découvrir et exploiter une injection SQL, il faut avoir certaines connaissances basiques du langage SQL.

DVWA — SQL Injection — level « low »

Bien, essayons de nous frotter au premier niveau de DVWA. On se trouve donc en face d'un formulaire de recherche des utilisateurs. Si l'on saisi « *1*« , nous obtenons en retour « *admin* » . **La première chose à faire va être de voir à quel point on peut agir sur la requête faite en base de données**. La plupart du temps, les requêtes SQL faites via PHP sont délimitées par des *guillemets* ou des *apostrophes*, également, les paramètres fournis par le client web sont parfois directement inclus dans la requête. Ici, si l'on saisi » *1'* » **on obtient une erreur SQL**, cela prouve donc que l'on peut agir sur la requête SQL faite au serveur de base de données, **en tout cas modifier sa structure pour qu'elle ne soit plus transmise de la même façon**. Voila plus exactement comment est construite la requête si l'on regarde le code source :

```
"SELECT first_name, last_name FROM users WHERE user_id = '$id'"
```

Ici, nous pouvons voir que le paramètre envoyé par les clients via le formulaire **est directement inclus, sans vérification, dans la requête SQL**. Voici donc ce que devient notre » 1' » :

```
"SELECT first_name, last_name FROM users WHERE user_id = '1''"
```

On voit donc ici qu'il y a effectivement un problème d'apostrophe, d'où l'erreur. Essayons de régler ce problème en rajoutant une apostrophe pour fermer celle que nous avons ouverte » 1 » » :

```
"SELECT first_name, last_name FROM users WHERE user_id = '1'' '"
```

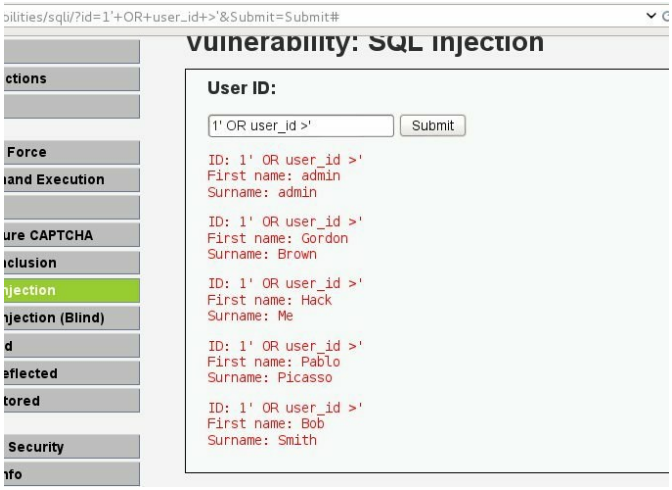
Nous remarquons alors qu'il n'y a plus d'erreur, ici, on peut donc espérer inclure du code dans la requête SQL, plus exactement rajouter des informations dans la requête SQL

```
"SELECT first_name, last_name FROM users WHERE user_id = '1' informations ajoutées dans
la requête qui crée nouvelles d'erreur'"
```

Faisons simple pour l'instant en essayant d'afficher plus d'un utilisateur : 0' OR user_id >'

```
"SELECT first_name, last_name FROM users WHERE user_id = '0' OR user_id >'' "
```

Voici le résultat que l'on pourra voir :

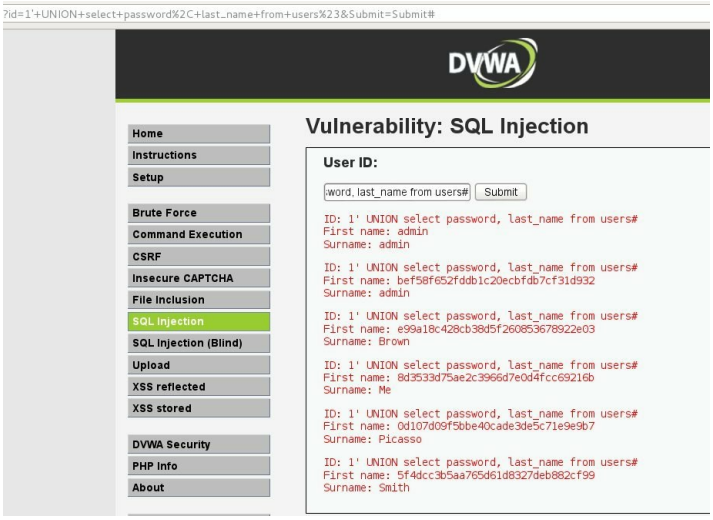


Ce que nous voyons ici est donc une modification de la requête SQL passée par le script PHP. Au lieu de simplement demander l'affichage des informations pour l'utilisateur ayant l'ID « *1* » , on a demandé l'affichage des informations pour l'utilisateur dont l'ID est 1 OU ID supérieur à rien (on aurai pu mettre 0), ce qui a pour effet d'afficher les informations de tous les utilisateurs.

Si l'on veut aller plus loin dans l'attaque, nous pouvons essayer de deviner le nom de la colonne contenant les mots de passe puis de les afficher grâce à une requête « *UNION* » permettant de joindre une autre requête *SELECT*, exemple : *1' UNION select password, last_name from users#*

```
"SELECT first_name, last_name FROM users WHERE user_id = '1' UNION select password,
last_name from users#"
```

Ici, nous arrivons donc à exploiter la faille SQL injection afin d'afficher directement les mots de passe des utilisateurs dans la page de réponse, ou plutôt le hash des mots de passe utilisateurs :



Comme je vous l'ai indiqué, il est nécessaire d'avoir quelques bases du langage SQL afin d'exploiter ce genre de faille. On peut généralement exfiltrer une très grande quantité de données lorsqu'une faille *SQL injection* est exploitée. Il est à noter que l'on retrouve également un principe de cloisonnement important. **Si l'application web requête la base de données avec un identifiant tel que « *root* » ou un utilisateur ayant des droits de lecture/écriture sur d'autres bases de données, il sera alors également possible de les requêter.**

Pour vous donner un exemple de jusqu'où nous pouvons aller en exploitant cette faille, on peut également très bien afficher un fichier local du serveur, notamment via « *load_file*« , qui est une syntaxe permettant la lecture de contenus locaux pour les intégrer à la réponse SQL : LOAD DATA INFILE Syntax

```
' union all select load_file('/etc/passwd'),null #
```

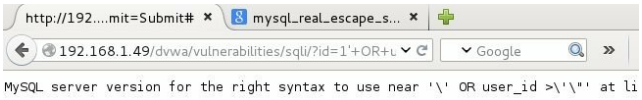
Résultat :

Ici, nous arrivons donc à lire certains fichiers du serveur web au travers l'exploitation d'une faille SQL injection.

DVWA — SQL Injection — level « medium »

Passons au niveau supérieur, des mesures de sécurité ont du être prises et nos méthodes d'exploitation utilisées précédemment ne fonctionnent plus. Si l'on cherche à les réutiliser, nous remarquons que les caractères spéciaux sont échappés.

> **Note** : Échapper un caractère signifie le « *marquer* » afin que celui-ci ne soit plus interprété et affiché comme n'importe quel autre caractère. Le fait que les caractères spéciaux comme *les guillemets* ou *les apostrophes* soient interprétés nous a permis d'exploiter la faille SQL injection dans le niveau précédent.



Si l'on part à la recherche des méthodes et fonctions que l'on peut utiliser en PHP afin de se protéger de certains caractères, on peut croiser le chemin de la fonction PHP mysql_real_escape_string. Celle-ci est souvent utilisée pour protéger les requêtes SQL intégrant des paramètres envoyés par l'utilisateur. **Néanmoins nous allons voir que cette fonction peut être déjouée**, c'est d'ailleurs le but dans ce niveau DVWA.

Si l'on schématise, il faut donc faire passer des caractères comme des guillemets ou des apostrophes, sans inscrire dans l'URL les caractères en question. Un joyeux défi ! Pour ceux qui ont l'habitude de travailler sur les technos web, vous avez peut être déjà entendu parlé de l'encodage, **et plus spécifiquement de l'encodage HTTP ou « HTML URL Encoding ».** Pour ceux qui ne voient pas de quoi il s'agit, je vous conseille de faire un tour sur cette page : http://www.w3schools.com/tags/ref_urlencode.asp

L'HTML URL Encoding permet entre autre de faire passer au travers une URL des caractères spéciaux ou des lettres avec accent qui pourraient ne pas être interprétés correctement par les navigateurs ou les serveurs web. Ainsi, l'utilisation d'un encodage basé sur les caractères ASCII est utilisé (notamment la valeur « *Hex* » des lettres : http://table-ascii.com/)

En utilisant l'HTTP URL Encoding, un espace devient un %20 dans l'URL, un « ! » devient un %21, un « % » un %25, etc.

Cela nous permet donc ici de faire passer une guillemet ou une apostrophe de façon encodée pour ne pas qu'ils soient détectés et échappés par la fonction *mysql_real_escape_string*.

Autrement dit, au lieu d'envoyer la chaîne suivante :

```
1' OR user_id >'
```

On enverra son équivalent encodé :

```
1%20OR%20user_id%20%3E0
```

Pour cela, il faut directement insérer cela dans la valeur du paramètre « *id=* » dans l'URL, le saisir directement dans le formulaire ne fonctionnera pas.

Également, on peut effectuer la même chose avec les autres commandes SQL permettant d'exploiter cette faille :

```
1%20UNION%20select%20password%2C%20last_name%20from%20users%23
```

SQL Injection — Des pistes de protection

Nous étudierons les pistes de protection après le chapitre sur les failles de type SQL Injection « *Blind* » , rendez-vous en bas de cet article !

Failles de type SQL Injection (Blind)

Sécurité : Fonctionnement et impact d'une attaque SQL Injection Blind

Passons maintenant aux failles de type SQL Injection Blind. Dans le fond, le concept est le même qu'une injection SQL « *standard* » : **injecter des données pour agir sur la base de données de l'application web visée**. Dans la forme, il y a une différence majeure : **l'absence de l'affichage d'erreur permettant de « *guider* » une injection SQL standard**.

En effet, nous avons vu que dans le cadre de l'injection SQL, celle-ci est repérée lorsque l'on arrive à avoir comme retour à notre requête une erreur SQL. Techniquement, le non affichage des erreurs SQL est due à l'absence, dans le code source, du code PHP suivant :

```
or die('<pre>' . mysql_error() . '</pre>' );
```

Celui-ci permet de renvoyer l'erreur indiquée par la base de données « *mysql_error()* » directement dans la page HTML entre les balises d'affichage « *pre* » . On obtient donc un retour sur la nature de l'erreur. **Lorsque l'on parle d'injection SQL Blind, c'est que l'on cherche à exploiter une faille de type Injection SQL sans avoir la possibilité de voir les erreurs SQL car le code renvoyant les erreurs est absent dans la page PHP**, justement par mesure de sécurité et afin de compliquer le travail de l'attaquant.

On doit alors faire un ensemble de tests et de requêtes qui, selon leurs résultats, indiqueront si l'on est parvenu à modifier la requêtes SQL ou non.

DVWA — SQL Injection blind — level « low »

Dans le cas présent, **nous savons que la valeur à envoyer est un chiffre**. Envoyer « *1* » affiche les informations de l'utilisateur « *admin* » , envoyer « *2* » affiche les informations de l'utilisateur « *Gordon* » , etc. Il est ici nécessaire de savoir si seules certaines valeurs sont acceptées (Exemple : si l'on envoie pas précisément « *1* » ou « *2* » ou « *3* » , on se fait rejeter) ou si l'on est un peu plus libre que cela.

Il faut savoir que les requêtes SQL acceptent assez bien les opérations mathématiques. En règle générale dans un requête SQL, si l'on envoie « *1+1"* , cela sera automatiquement traduit par « *2* » . Cela peut dans notre cas être utile pour voir à quel point l'on peut modifier la requête SQL envoyée au serveur de base de données.

Si l'on essai d'envoyer la valeur « *1+1* » dans le formulaire, nous avons en réponse les informations de l'utilisateur « *Gordon* » :

Si l'on envoie « *2-1* » , nous avons les informations de l'utilisateur « *admin* » :

Cela signifie deux choses :

- Il n'y a pas de contrôle précis dans le code PHP, en tout cas pas de contrôle vérifiant que la donnée envoyée est purement un nombre sans caractère spécial par exemple
- On peut modifier la requête SQL avec les données que l'on envoie

On peut alors tenter d'aller plus loin en reprenant les exploits utilisés dans le niveau « *low* » et « *medium* » d'Injection SQL :

On arrive donc à modifier la requête en y ajoutant des données supplémentaires, signe qu'une Injection SQL est possible

DVWA — SQL Injection blind — level « medium »

Même chose que l'exercice précédent, le mode de détection étant le même et la méthode d'exploitation est la même que pour l'exercice « SQL injection medium » .

SQL Injection Blind — Des pistes de protection

Les failles de type SQL Injection sont très répandues, toutefois il possible de s'en protéger, voyons ensemble quelques conseils permettant d'éviter qu'un attaquant puisse profiter de l'exploitation d'une faille SQL Injection.

- **Avoir une phase d'épuration des données** (« *sanitization* ») pour chaque paramètre que le client web peut avoir à envoyer, notamment grâce à un ensemble de fonctions PHP permettant le traitement sécurisé de données : *mysql_real_escape_string*(), addcslashes(), htmlentities() et htmlspecialchars(). Je vous conseille d'ailleurs la lecture de ce tutoriel sur OpenClassRooms : Sécurité PHP, sécuriser les flux de données

- **Utiliser un firewall Applicatif de type mod_security**, nous avons vu que la syntaxe globale d'une Injection SQL contient des éléments qu'il est possible de récupérer. Par exemple le fameux *OR 1=1* ou encore l'utilisation de requêtes *UNION* (qui peuvent parfois se trouver dans le comportement des applications web). Un système de firewall applicatif, se positionnant entre le client web et le serveur web, **permet alors d'intercepter, d'analyser les requêtes clients puis de filtrer celles jugées suspectes**. Cela permet de se couvrir de bon nombre de failles de sécurité dont certaine déjà vues dans cette suite d'article. Je vous invite à consulter mon article sur l'installation et la configuration de *mod_security* publié sur IT-Connect : Installation de mod_security devant un serveur web Apache

- **Limiter les droits au niveau de la base de données**, comme je l'ai souligné, il est primordial d'avoir une bonne segmentation des droits au sein de sa base de données. **Le fait de donner accès à plusieurs bases de données à un utilisateur est généralement à proscrire**. Principalement car cela permet à un attaquant ayant la main sur cet utilisateur depuis une des applications web, de récupérer les données de plusieurs autres applications web. Également, il est tout à fait déconseillé de faire tourner une application web avec les droits « *root* » d'une base de données, cela pour les mêmes raisons.

- **Désactiver l'option Load File** : Nous avons vu que, si l'utilisation de « *load_file* » était possible dans la base de données MySQL, il était alors permis de charger des fichiers locaux du serveur pour les afficher dans la réponse faite au client. Cela amène notamment au fait de pouvoir lire le fichier */etc/passwd* ou d'autres fichiers contenant des informations sensibles. Il est donc conseillé de désactiver la possibilité d'utiliser la syntaxe *load_file* au sein de MySQL, rares sont les applications web en ayant besoin.

- **Éviter la construction de requêtes SQL directement avec les informations saisies et envoyées par le client web**, préférer les requêtes préparées, comme décrit dans ce cours OpenClassRooms : Requêtes préparées PHP

- Pour les applications web connues, **il est recommandé de changer le préfixe de base de données utilisé par défaut**, cela évite entre autre le fait que l'attaquant puisse facilement retrouver la façon dont sont construites les requêtes SQL. La difficulté qu'on lui ajoute lorsque l'on change le suffixe est que le nom des tables nécessaire à la formation de la requête change.

Il s'agit là de quelques conseils qui vous permettront de vous protéger contre les failles de type SQL Injection, il en existe beaucoup d'autres.

Dans le prochain et dernier article sur DVWA, nous nous attaquerons aux Failles File Upload et XSS.

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

Partager :

Published in Challenges et CTFs

DVWA