Mini-Projet de Quantified Self (Qself)

Détection de passe au

Unihockey



Cédric Campos Carvalho, Thomas Frantzen

Et Henri Jaton

Responsable : **Andres Perez-Uribe**

Date : **19 novembre 2021**

# Introduction

## Contexte

Ce mini-projet a été réalisé dans le cadre du cours « Qself ». Il a pour but de valider le cours et d’appliquer les connaissances acquises durant le semestre.

## But

L’objectif du projet est de détecter les différents types de passes au unihockey grâce au machine learning et d’afficher des statistiques et résultats au travers d’une interface graphique (site web, application ou autre.

# Challenge

## Définitions

SQL Injection : La faille SQLi, abréviation de SQL Injection, soit injection SQL en français, est un groupe de méthodes d'exploitation de faille de sécurité d'une application interagissant avec une base de données. Elle permet d'injecter dans la requête SQL en cours un morceau de requête non prévu par le système et pouvant en compromettre la sécurité.

## Outils utilisés

* Metasploit
* WhatWeb
* dirbuster
* Burp
* Python
* PHP

## Détail de l’approche

### Instanciation du site

Pour lancer le site web, une instance est créée directement sur le site de hackthebox. L’avantage est cela permet de rapidement se lancer dans le challenge en se passant de toute installation. Je reviendrai sur les désavantages dans la conclusion.

### Fonctionnement

La première chose que j’ai faite est de comprendre comment le site fonctionne. Lorsque l’on débarque sur le site, on tombe sur la page suivante qui permet donc de paramétrer une configuration Nginx. Une fois la configuration crée, on la retrouve en bas avec l’id de l’utilisateur actuel et un id qui lui est général (un id différent par config, peu importe l’utilisateur) :

Graphical user interface

Description automatically generated

Si l’on clique sur la configuration, il est possible de l’afficher :

Text

Description automatically generated

### Approche : Console développeur du navigateur

Ensuite j’ai tenté de regarder dans la console du navigateurs les fichiers du site web accessibles, les différentes pages et les différentes requêtes émises et reçues par le site.

Je n’ai rien trouvé me permettant d’avancer excepté une chose étrange : Le cookie est mis à jour environ chaque deux secondes. Je garde cela de côté.

### Approche : Vulnérabilités Nginx

Le challenge s’appelant Nginxatsu et l’énoncé ne donnant aucun indice, je me suis dit qu’il y’avait de grandes chances pour que le site web passe lui-même par Nginx.

J’ai donc regardé l’output de la génération de configuration Nginx.

J’ai ensuite cherché sur google les vulnérabilités connues dans les configurations Nginx.

J’ai remarqué que la vulnérabilité « off by slash » est présente dans la configuration. Cette vulnérabilité, qui correspond à un slash manquant dans « location » dans « server » combinée à la directive « alias ». Cela permet potentiellement de récupérer une partie ou l’ensemble du code source de l’application web. Dans notre configuration, voici les lignes problématiques :

Text

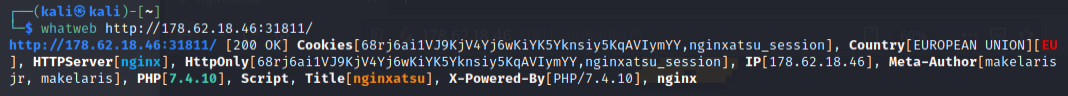
Description automatically generated

Cela est possible car étant donné le manquement du « / », le serveur Nginx va tenter de normaliser l’URL, par exemple avec http://IP:PORT/assets/xxxx il restera <http://privateadress/www/public//xxxx> , en manipulant l’adresse, il est potentiellement possible de se balader dans les fichiers sur site.

Mais maintenant le fait est que c’est juste l’application web qui est un créateur de configuration. Rien ne nous dit que l’application web elle-même comporte un Nginx. Comme mentionné précédemment, n’ayant pas d’autres indices et le challenge étant fortement lié à Nginx, j’ai ensuite cherché un moyen de savoir sur quoi tourne le serveur http.

J’ai trouvé que l’outil appelé « WhatWeb » permet d’obtenir cette information.

En effectuant un scan de mon adresse, j’obtiens ceci :



On peut donc voir que l’application est codée en PHP et passe par un serveur Nginx.

Je me demande donc si potentiellement avec ces indices, la configuration du Nginx est du même type que celles générées par l’application. C’est ce dont je vais commencer par exploiter.

Je commence par essayer d’accéder à IP:PORT/assets car c’est le chemin par défaut de la config Nginx à générer.

Graphical user interface

Description automatically generated

Cela me retourne ensuite un fichier « config » depuis http://IP:PORT/api/configs

Graphical user interface, text, application

Description automatically generated

Cela confirme le fait que la structure est similaire.

Je tente donc maintenant d’utiliser l’exploit, je commence par essayer d’accéder au répertoire parent en utilisant http://IP:PORT/assets../ avec l’outil « dirbuster » présent sur Kali Linux. Ce logiciel permet, à partir d’une liste de sous répertoire et de nom de fichiers connus, de dumper un site si une faille est connue. Cela fonctionne très bien avec cette faille.

Table

Description automatically generated

Rapidement, j’obtiens un fichier intéressant : /public/index.php

Table

Description automatically generated

Je découvre que l’application est codée en Laravel, ce qui sera une piste pour la suite de l’exploit.

Je vais donc chercher les failles connues de Laravel.

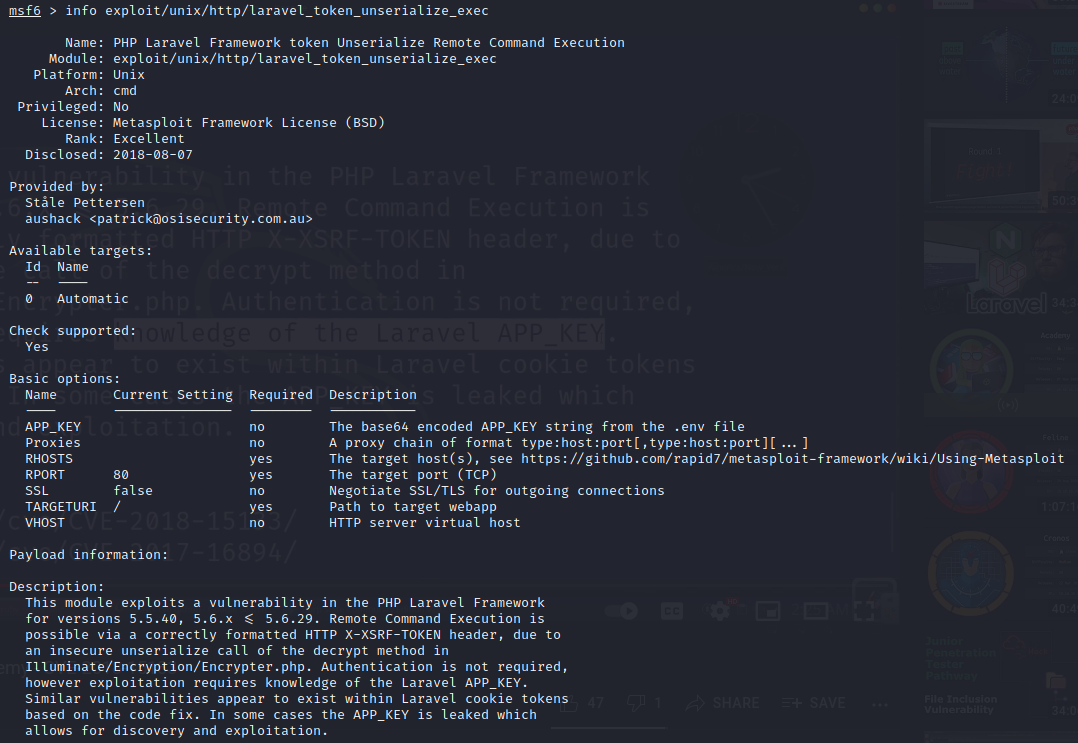
### Approche : Laravel avec Metasploit

Je commence par rechercher des failles avec Metasploit :

Graphical user interface, text, application

Description automatically generated

Une faille est connue. En tapant « info » + le nom, je peux trouver les prérequis pour l’utiliser.



Je vois donc qu’il faut avoir l’ APP\_KEY de l’application. Je vais donc voir dans mes fichiers accessibles si j’ai la possibilité d’extraire cette info. Malheureusement elle est introuvable.

Je cherche donc sur google comment la récupérer et je découvre qu’elle se trouve dans le « .env » de l’application (/../.env).

En essayant d’accéder à http://IP:PORT/assets../.env et je parviens à l’extraire :

Text

Description automatically generated with low confidence

J’obtiens une grande partie des informations sensibles de l’application dont ce fameux « APP\_KEY ».

Je tente donc l’exploit à l’aide de Metasploit. Je remarque que ça ne fonctionne pas. Je regarde donc la version de Laravel, l’exploit ne fonctionnant que jusqu’à la version 5.6.29.

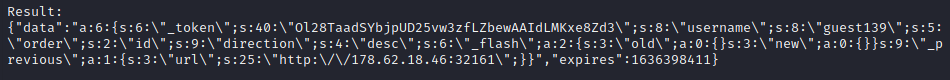
J’aurais dû faire cela avant car effectivement la version est la 5.8.10. J’ai trouvé l’information dans le fichier composer.json.

Cet exploit permettant d’accéder au shell de la cible à l’aide l’APP\_KEY et des cookies, malheureusement je ne pourrais donc pas l’utiliser.

### Approche : Laravel par injection SQL dans le cookie

Malgré l’échec de Metasploit, je me dis en ayant pu récupérer l’API\_KEY et que les cookies sont étrangement mis à jour toutes les 1-2 secondes qu’il y’a définitivement quelque chose à faire avec ceux-ci.

Je vais donc regarder sous quelle forme se présente le cookie en le décryptant à l’aide d’un code python trouvé sur le net. La valeur contient les informations suivantes :



Je vois que dans le cookie sont passées plusieurs informations telles que « username », « order », « direction » etc.

Je décide donc de trouver dans les fichiers que j’ai pu dumper grâce à la première faille ou ces infos sont récoltées/utilisées.

Après de longues recherches je parviens à tomber sur le fichier « ConfigController.php » se trouvant dans « /assets../app/Http/Controllers/API/ConfigController.php ».

Ce fichier contient ceci :

Text

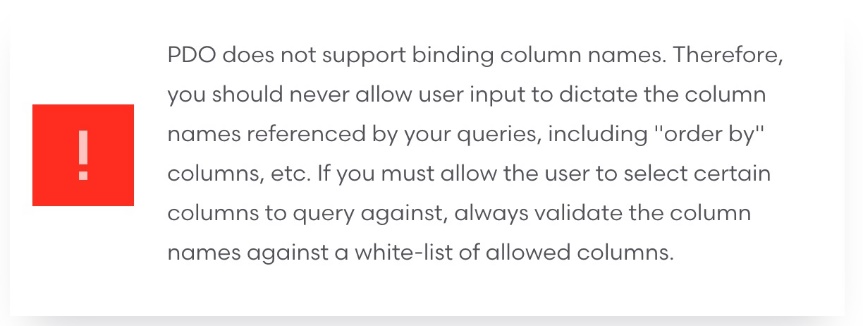
Description automatically generated

En l’observant un peu, je remarque cette fonction :

A picture containing text

Description automatically generated

J’avais lu lors de mes recherches sur les potentielles failles de Laravel qu’il était possible d’effectuer des injections SQL via le nom de colonne. Cela est dû au fait que celui-ci n’échappe pas les paramètres passés au Query Builder

.

On peut voir que le contrôleur utilise « orderBy » avec des paramètres provenant directement du cookie. Cela nous permet donc possiblement d’effectuer une injection en remplaçant les paramètres de notre cookie décrypté par une commande SQL et de le réencrypter avant de le renvoyer à l’application.

Étant un novice en SQL et surtout en injections, j’ai dû demander de l’aide sur un discord de Ethical Hacking pour la construction de celles-ci ainsi que pour tester la possibilité d’effectuer des injections.

Si nous revenons à notre cookie, il est donc possible d’effectuer une injection à partir de ces informations :

s:5:\"order\";s:2:\"id\"

Nous savons grâce au fichier « .env » que la base de données est de type MySQL. Cela nous permet de chercher les différentes requêtes possibles sur celle-ci.

J’ai tenté d’abord d’utiliser Burp pour changer le cookie avant d’envoyer la requête mais malheureusement je n’ai jamais réussi à avoir de réponse satisfaisante. J’imagine que cela est dû au fait que le cookie change très souvent. J’ai donc dû partir sur une solution scriptée.

Après une fastidieuse écriture d’un script python en s’aidant de plusieurs fichiers trouvés sur internet (décryptage, sérialisation etc.) j’ai enfin réussi à tester deux injections SQL. Il faut ici utiliser des blinds injections parce qu’il est impossible d’extraire directement les informations telles qu’elles.

Nous allons donc chercher à manipuler cette partie pour tester une injection.

L’aide qu’on m’a fourni a permis d’arriver à ceci :

s:5:\"order\";s:2:\"id->\"’)), ({SQL QUERY}) #"

J’avais le choix entre des injections dites « times based » qui consistent à envoyer une requête qui induira un « sleep » , ce qui permet en mesurant le temps avant le retour de la requête, de savoir si cela a fonctionné Le désavantage est que c’est passablement lent. L’autre solution est d’utiliser des « error-based » injections, qui celles-ci retourneront 200 en cas de succès ou 500 en cas d’échec. C’est ce type d’injections que je vais utiliser ici.

Les deux payload de test se présentent de la façon suivante :

* "id->\"')), (SELECT (CASE WHEN (SELECT 1)=1 THEN 'SUCCESS' ELSE (select exp(~(SELECT \* FROM (select user())x))) END)) #"

Retourne 200 OK

* "id->\"')), (SELECT (CASE WHEN (SELECT 1)=2 THEN 'SUCCESS' ELSE (select exp(~(SELECT \* FROM (select user())x))) END)) #"

Retourne 500

Cela consistera à tester caractère après caractère le nom des colonnes, ce qui nous retournera 200 en cas de caractère correct et 500 en cas de caractère incorrect.

Text

Description automatically generated

Pour commencer, étant en MySQL, il faut extraire les infos de l’information\_schema.tables.

J’ai pu, en recherchant la notion « fl », extraire de l’information\_schema.tables la table « definitely\_not\_a\_flaag ».

* "id->\"')), (SELECT (CASE WHEN (SELECT ASCII(SUBSTRING(table\_name,{pos},1)) FROM information\_schema.tables WHERE table\_name LIKE '%fl%' LIMIT 1) = {ch} THEN 'SUCCESS' ELSE (select exp(~(SELECT \* FROM (select user())x))) END)) #"

En cherchant la colonne « flag » dans celle-ci, j’ai découvert une fausse colonne nommée « flag » (comme si ce n’était pas déjà compliqué).

En ajustant la requête, il est possible trouver la bonne colonne flag : « flag\_XXXXX » change à chaque instance de l’application.

- "id->\"')), (SELECT (CASE WHEN (SELECT ASCII(SUBSTRING(table\_name,{pos},1)) FROM information\_schema.tables WHERE table\_name LIKE '%fl%' LIMIT 1) = {ch} THEN 'SUCCESS' ELSE (select exp(~(SELECT \* FROM (select user())x))) END)) #"

Calendar

Description automatically generated with low confidence

Finalement, à l’aide d’une nouvelle requête, j’ai pu extraire le flag correspondant :

* "id->\"')), (SELECT (CASE WHEN (SELECT ASCII(SUBSTRING(flag\_xxxxx,{pos},1)) FROM nginxatsu.definitely\_not\_a\_flaaag) = {ch} THEN 'SUCCESS' ELSE (select exp(~(SELECT \* FROM (select user())x))) END)) #"

A picture containing calendar

Description automatically generated

Cela me permet d’obtenir le flag :

{HTB\_XXXXXXXXXXXXXXX}

## Mitigation

### Nginx :

Il suffit d’enlever le « off-by-slash » en ajoutant un / à la fin de assets

A picture containing text

Description automatically generated

### Laravel

Il suffit de d’échapper les données passées au contrôleur manuellement. Il ne faut pas passer des noms de colonne user-controlled au query builder sans whitelister les potentielles séquences problématiques.

# Conclusion

Le challenge était très intéressant, il m’a permis de découvrir beaucoup d’outils, de me familiariser avec ceux-ci et d’apprendre beaucoup de choses sur SQL, Laravel et Nginx. Cependant le challenge était plus compliqué que je pensais en étant également passablement long. Mon manque d’expertise dans le web ainsi que SQL m’ont fait perdre énormément de temps.

De plus, les instances de hackthebox ont été très souvent coupé en plein milieu de test ou extractions, ce qui m’a également fait perdre énormément de temps. Le problème étant que les tables , l’API\_KEY et d’autres variables sont générées aléatoirement à chaque instance. Pour moi ce challenge peut être qualifié de « Hard » voire « Very Hard » par le nombre de pièges et le nombre d’exploit à effectuer.

Je recommande toute de même ce challenge qui se veut passionnant, c’est plus le manque de temps qui m’as vraiment pénalisé dans mon cas.

# Références

*Error blind payloads*. (s.d.). Récupéré sur https://www.neuralegion.com/blog/sql-injection-payloads/#error-blind-payloads

Hamdan, M. (s.d.). *Youtube - Laravel PHP Vulnerabilities - CVE-2018-15133*. Récupéré sur https://www.youtube.com/watch?v=egBkzDJmL7k

*Laravel Tricks - HackTricks*. (s.d.). Récupéré sur https://book.hacktricks.xyz/pentesting/pentesting-web/laravel

nbah22. (s.d.). *laravel\_encryption.py.* Récupéré sur https://gist.github.com/nbah22/1fd7da3deb5b416a0524

*SQL Injections In Laravel*. (s.d.). Récupéré sur https://cyberpanda.la/blog/laravel-sql-injections

to, C. N. (s.d.). Récupéré sur https://blog.detectify.com/2020/11/10/common-nginx-misconfigurations/

*Vulmon - Know Laravel Vulnerabilities*. (s.d.). Récupéré sur https://vulmon.com/searchpage?q=laravel

# Annexes

* **Code source de la solution**
* **Présentation**