Bug Bounty basé sur PortSwigger Web Security Academy.

Table des matières

Authentification

0x01 - Bruteforce

Pour le premier lab, on va utiliser l'attaque bruteforce (énumération de mots de passe) pour trouver le mot de passe de l'utilisateur. En utilisant Hydra, on détermine assez rapidement les creds :

```
login : jeremy
password: letmein
```

0x02 - Bypass MFA

Pour ce lab, on va devoir bypass le MFA (Multi-Factor Authentication) pour accéder au compte de l'utilisateur.
Une règle d'or quand on fait du client-serveur est : "Ne jamais faire confiance au client".
Le problème ici est que le serveur fait confiance au client pour le nom d'utilisateur.

Ainsi, on part de :

```
username2=jesamy&mfa=610829
```

on modifie le nom d'utilisateur :

```
username2=jeremy&mfa=610829
```

On a donc accès au compte de l'utilisateur.

0x03 - IDOR

de Vaadata.com : `Une vulnérabilité de type IDOR est un problème de contrôle de droits, qui apparait lorsqu'une référence directe`

`à un objet (fichiers, informations personnelles, etc.) peut être contrôlée par un utilisateur.`.

C'est donc ce qu'on va utiliser pour accéder à un compte qui ne nous appartient pas.

on part donc de l'url de base : `http://0.0.0.0/labs/e0x02.php?account=1009`

on modifie les informations du compte : `http://localhost/labs/e0x02.php?account=1008`

On a donc accès, sans posséder les droits.

0x04 - Json Web Token (JWT)

On peut signer un JWT... ou pas.
Et si on ne le signe pas, un simple passage sur cyberchef nous permet de modifier le contenu du JWT et de nous connecter en tant qu'admin.
En l'occurence :

On a juste à modifier le payload du JWT pour avoir les droits d'admin : rôle de `staff` à `admin`, requête `PUT` et le tour est jouée.

Inclusion de fichiers

0x01 - Et si on incluait des fichiers ?

Ça ressemble un peu à une vulnérabilité IDOR. Mais ça n'en est pas une.
En effet, on peut inclure des fichiers en utilisant le paramètre `file` de l'url.
Exemple : `http://0.0.0.0/labs/fi0x01.php?filename=../fi0x01.php` nous permet de voir le contenu du fichier en question.

Plutôt embêtant, mais nous permet de solve le lab.

0x02 - Inclusion de fichiers : le retour

Dans ce lab-ci, il semble qu'il y ait un filtre implémenté pour empêcher l'inclusion de fichiers.
Cependant, on peut toujours le contourner en utilisant un caractère encodé comme `%00` pour passer outre le filtre.

XSS

0x01

On tente avec des requêtes basiques, ça n'a pas l'air de fonctionner. Il doit y avoir un filtre.
Toutes ? Non, pas toutes. on s'aperçoit que l'event `onerror` n'est pas filtré.
On contourne donc le filtre en utilisant cet event pour exécuter notre payload.

```
<img src="x" onerror="alert(1)">
```

On peut donc résoudre le lab comme ça.

0x02

Le deuxième lab est encore plus simple;
On remarque directement que les champs de commentaire ont l'air vulnérables, on execute un payload de base et le tour est joué.

```
<script>alert(1)</script>
```

0x03

Ce lab ressemble un peu à ce qu'on peut voir sur rootme avec le bot qui refresh la page, dont on doit voler le cookie.
En effet, on voit qu'il y a un formulaire de soumission de cookie, ce qui nous permet d'envoyer un payload qui sera exécuté par l'administrateur dès qu'il le verra.

Upload de fichiers

0x03

L'application a l'air d'accepter les fichiers avec une extension `.phtml`.
On essaye donc d'uploader un fichier avec une extension `.phtml`, donc avec du code php dedans, et on voit que l'application l'accepte... Mais surtout l'execute !

exemple de payload :

```
<?php echo "Ça marche vraiment, ça ?"; ?>
```

On a donc pu résoudre le lab.

SSRF

de vaaadata.com : “A partir d’une application web vulnérable, les SSRF permettent d’interagir avec le serveur, afin d’en extraire des fichiers et de trouver ses autres services actifs. Mais cela ne s’arrête pas là. Il est également possible de scanner le réseau interne afin d’en cartographier les IP et Ports ouverts.”

Le SSRF est une vulnérabilité qui permet à un attaquant de forcer une application à effectuer des requêtes HTTP vers des ressources arbitraires. SSRF signifie Server-Side Request Forgery, ce qui signifie que l’attaquant peut forcer le serveur à effectuer des requêtes vers des ressources arbitraires.

0x01

Dans ce lab, on peut intercepter une requête qui est faite par l’application, et en modifiant l’url de la requête, on peut accéder à des ressources internes.

exemple :

```
{"url":"http://0.0.0.0/labs/api/thirdparty/amazoom.php"}
```

On la modifie :

```
{"url":"http://0.0.0.0/index.php"}
```

Et on a accès à la page d’accueil de l’application.

0x02

On s’aperçoit qu’à présent il y a un filtre qui empêche l’accès à des ressources internes.
En effet, si on essaie d’accéder à quoi que ce soit d’autre que `localhost`, on se fait bloquer.

On s’aperçoit assez vite qu’il doit s’agir d’une regex assez simple, on peut donc contourner cette protection en utilisant une URL du type `http://` :

```
{"url":"http://localhost.monserveur.com/scriptmalveillant.php"}
```

CSRF

Le CSRF (Cross-Site Request Forgery) est une attaque qui force un utilisateur à exécuter des actions non désirées sur une application web dans laquelle il est authentifié.

0x01

Ce lab est assez simple. On s'aperçoit que le cookie peut être reforgé en utilisant un autre utilisateur.

Le cookie ressemble à ça :

`csrf0x01=jeremy`

On a juste à modifier `Jeremy` par l'utilisateur qu'on veut, `admin` par exemple, et le tour est joué.

0x02

Cette fois, on a deux cookies CSRF, mais il semble qu'aucune vérification ne soit implémentée pour vérifier que les deux cookies sont cohérents.

avec ces cookies (`csrf0x01=jessamy` et `csrf0x02=jessamy`) on peut envoyer une demande de changement de mot de passe pour l'utilisateur `jessamy` en utilisant le cookie `csrf0x02` :

Il est assez évident qu'une validation (server-side) est nécessaire pour vérifier que les deux cookies sont cohérents.

Open Redirect

Une vulnérabilité d'Open Redirect est une faille de sécurité qui permet à un attaquant de rediriger un utilisateur vers un site malveillant.

0x01

si on requests le paramètre `return_url`, on peut effectuer une redirection arbitraire.

Exemple concret :

`http://0.0.0.0/labs/r0x01_script.php?id=1&return_url=https://monsite.com/monscriptevil.php`

SQLI

Pour les SQLI, la méthode est la même. Seule la valeur du dictionnaire change.

On utilise le dictionnaire de payloards de portswigger, chaque exercice se fait en capturant le site sur burpsuite, puis en envoyant vers le repeater, et enfin en

modifiant la catégorie par le dictionnaire de payloads donné. Les labs sont donc tous très simples à faire.

Méthode générale de résolution :

Là encore, il y a une sorte de "règle d'or" lorsqu'on tente une injection SQL : on essaye toujours avec un payload de base, et on voit si ça marche.
En l'occurence, là on va utiliser `x' OR '1'='1` pour voir si l'injection est faisable.

Ça nous permet de déterminer si une condition toujours vraie est possible, et donc si l'injection est possible.

Ensuite, on peut utiliser un script python pour voir combien de colonnes sont présentes dans la table, en utilisant la valeur `null`, puis on extrait le nom des tables, puis des colonnes, puis des données.

Conclusion

Bien que ce rapport soit loin d'être exhaustif compte tenu du nombre de challenges proposés par PortSwigger, il permet de donner un aperçu des vulnérabilités les plus courantes et de leur résolution.
Il est important de noter que ces vulnérabilités sont très courantes et peuvent être exploitées par des attaquants pour compromettre des systèmes.

Auteurs :

R.T.
M.J.R.A.
S.L.
N.M.
T.M.