Roman Mkrtchian
François Chapuis

SI5 - 2012/2013

Rapport de projet de WASP

Réalisation d'un site web sécurisé

Introduction

Nous avons choisi de coder un blog sécurisé. Nous avons notamment codé nous-mêmes la partie connexion des membres et création et gestion d'articles, et nous avons utilisé l'application externe **DISQUS** pour gérer les commentaires (mashup).

Nous avons utilisé dans notre projet l'ORM (Object Relational Mapping) **Redbean** pour faciliter les requêtes SQL ainsi que la librairie WYSIWYG **CKEditor**, pour une édition d'articles pratique et agréable. Nous avons également utilisé la librairie CSS **Bootstrap** pour la présentation. Pour la purification et la sécurisation du code html entré par le client, nous avons utilisé la librairie **HTML Purifier**.

Notre blog utilise HTML5 et CSS3, il est conseillé de l'exécuter sous un navigateur récent.

Tester l'application

Pour tester notre blog il faut créer une base de données mysql et configurer les variables suivantes dans le fichier includes/setup.php :

- ROOTPATH qui indique le chemin absolu du site depuis la racine des fichiers accessibles de l'extérieur (par défaut /WASP-blog/)
- DB_HOST le host (par défaut localhost)
- DB_NAME le nom de le base de données du projet (par défaut wasp)
- DB_USER le nom d'utilisateur mysql (par défaut root)
- DB_PASSWORD le mot de passe de l'utilisateur mysql (par défaut rien)

Ces informations se trouvent vers la fin du fichier.

Le blog ne contient pas d'informations (puisque la base de données est vide), il suffit alors de créer un nouvel utilisateur sur le site (avec adresse mail valide), d'activer son adresse mail par le lien reçu par mail, et de poster des articles avec son compte. Les tables seront crées toutes seules, il suffit initialement d'une base de données vide.

Cependant, pour vous faciliter le travail de test, nous avons préparé un dump de base de donnée avec un utilisateur déjà validé et des articles déjà créés.
Les identifiants de l'utilisateur sont :
login : user@yopmail.com
pass : password
Le fichier de dump est wasp.sql

Identification des failles potentielles

Dans notre application, les problèmes de sécurité sont essentiellement liés aux champs dans lesquels l'utilisateur peut entrer des données, qui permettent d'exploiter notamment des failles de type XSS, Injection SQL ou include. Nous avons donc commencé par les identifier dans le cadre de la sécurisation de notre site.

Les endroits faillibles sont donc essentiellement :

- Le formulaire d'inscription
- Le formulaire de connexion
- Le formulaire d'ajout/modification d'article
- Le mashup (commentaires)
- La pagination qui utilise des paramètres GET
- L'édition/suppression des articles qui utilise des paramètres GET

Il y a également à prendre en compte les failles connues.

Nous avons pris en compte la possibilité de failles CSRF qui permettent à un attaquant d'exploiter la confiance qu'un site a en un utilisateur, en forçant par exemple un utilisateur privilégié à réaliser une action critique.

Nous nous sommes assurés que l'accès aux pages sensibles soit restreint aux utilisateurs privilégiés. Pour assurer la confidentialité, nous avons traité également le cas du stockage des mots de passe chiffrés.

Nous avons également traité les problèmes de spam.

Nous ne nous sommes pas occupés des attaques du type « Homme du milieu » (Man in the middle) qui pourraient être solutionnés par l'utilisation du protocole SSL/TLS accompagné d'un certificat valide (payant). Nous avons seulement favorisé son utilisation grâce au flag « Secure » des cookies.

Il est à noter également qu'il peut exister des failles sur des couches différentes, et notamment des failles côté serveur. Nous ne nous sommes pas occupés de ces failles étant donné qu'elles sont hors de la portée de ce projet.

Solutions de sécurisation

1 – Injection SQL

Notre approche pour combler cette faille a consisté à utiliser l'ORM RedBean qui utilise l'extension PDO (PHP Data Objects) et permet d'effectuer des « requêtes préparées ». Celui-ci utilise prend de notre part du code pré-formaté de la forme :

```
R::findOne('user',' email =:mail AND password =:password',
        array(
        ':mail' => $_POST['mail'],
        ':password' => $_POST['password']
        ))
```

Ou encore :

```
R::findOne('user',' email = ? AND password = ?',
        array($_POST['mail'], $_POST['password'] )
        )
```

Par ailleurs RedBean assure l'échappement des caractères et permet de rendre inoffensif le code malicieux qui pourrait être entré par injection pour être mêlé aux requêtes de base de donnée.

2 – XSS (Cross Site Scripting)

La sécurisation des failles XSS est très difficile. Nous avons donc fait appel à des librairies externes reconnues pour faire cela. Déjà au niveau du client, la librairie Bootstrap et la librairie CKEditor permettent certaines vérifications et l'interdiction de code html dangereux avec du javascript côté client. Mais cela ne garantissant pas la sécurité, nous avons utilisé la librairie PHP : HTML Purifier. Nous avons utilisé sa fonction de purification sur l'ensemble des paramètres GET et POST que le client est susceptible de transmettre au site, ainsi, tout code malicieux sera bloqué (par exemple du javascript) et le html sera toujours bien formé et ses fonctionalités dangereuses seront neutralisées côté serveur.

HTML Purifier s'utilise simplement, avec une fonction comme suit :

```
$clean_html = $purifier->purify($dirty_html);
```

3 – CSRF (Cross Site Request Forgery)

Lorsque l'utilisateur se connecte, un token unique est généré et stocké dans la variable de session.
Lorsqu'un formulaire est généré, un champ input « hidden » contenant ce token y est ajouté. Ainsi, lorsque l'utilisateur soumet le formulaire, le

token est accessible dans la variable $_POST et peut être comparé à celui de la variable $_SESSION, afin de vérifier l'authenticité de l'utilisateur.

Si jamais un attaquant vole le cookie de session de l'utilisateur, il ne pourra pas exécuter d'actions critiques avec sa session, et de même il ne pourra pas envoyer de « lien piégé » sur du contenu critique à l'utilisateur (par exemple via une balise img, script ou iframe) car il faudrait que ce lien contienne une requête POST avec la valeur du token unique.

4 – Cookies

Nos cookies ne contiennent pas d'information importante de sécurité, ils servent uniquement à identifier l'utilisateur, et à vérifier si l'utilisateur est connecté ou non en utilisant un identifiant stocké dans le cookie sous forme hashé (SHA1) pour identifier l'utilisateur de manière unique. Par ailleurs, il expirent au bout d'un certain temps limité.

Nous avons placé les flags « HttpOnly » et « Secure » dans la méthode setcookie(). HttpOnly permet d'apporter une protection contre des attaques XSS en sécurisant l'accès au cookie côté client (cela marche que si le navigateur le supporte), en obligeant d'utiliser http pour accéder au cookie, et Secure permet de forcer l'utilisation de HTTPS lorsque c'est possible.

Si un cookie fait référence à une session non-existante, il s'agit alors probablement d'une tentative de « session fixation » (https://www.owasp.org/index.php/Session_Fixation), il faut donc remplacer sa valeur immédiatement.

5 – Failles Include

Nous avons architecturé notre site de manière à ne pas avoir de faille include. En effet, à aucun moment nous ne passons en paramètre des pages php, et donc l'attaquant ne peut pas en inclure non plus.

Nous avons une page php différente par fonctionnalité (notre blog contient peu de fonctionnalités au final) et nous incluons au sein de celles-ci un header et un footer, mais nous n'incluons pas de pages à partir de données dynamiques.

6 – Spam

Pour prévenir le Spam, nous avons utilisé d'abord un CAPTCHA (reCAPTCHA) pour prévenir le spam sur la soumission d'articles, pour éviter qu'un bot déjà inscrit puisse poster massivement des articles.

Nous avons également mis en place une confirmation par mail lors de l'inscription pour éviter qu'un bot puisse s'inscrire automatiquement. Nous avons crée une table *accountvalidation* dans la base de données, qui associe un utilisateur à un token unique (sous forme de hash sha1) et contient une date d'expiration. Lors de l'inscription d'un utilisateur, ce token est généré et à la fois envoyé par mail (en paramètre GET dans un lien) et stocké dans la base de données. Lorsque l'utilisateur clique sur le lien du mail, le token est comparé à celui stocké dans la base, si ils correspondent, le compte de l'utilisateur est validé.

7 – Mashup

Nous utilisons le mashup DISQUS pour les commentaires. Nous n'avons rien fait (et ne pouvons rien faire) pour sa sécurisation en terme d'injection SQL, de failles XSS etc... c'est lui qui gère sa propre sécurisation.

Pour sécuriser quand même cette partie, nous l'avons simplement placé dans un iFrame, de manière à ce que le navigateur interprète son code comme étant séparé et comme provenant d'un site externe. Si jamais il comporte des failles de sécurité, cela n'affectera pas notre site et notre base de données, mais seulement nos commentaires.

8 – Confidentialité

Nous ne stockons pas les mots de passe en clair dans la base de données pour des raisons de confidentialité. Nous utilisons un hash SHA1 avec salt, c'est à dire que le mot de passe ne pourra normalement pas être retrouvé, même si un attaquant réalise un dump de la base de données.cookie HttpOnly

Nous utilisons par ailleurs un .htaccess pour protéger nos fichiers sensibles.

Nous utilisons par exemple la règle « LIMIT GET POST » :

```
<Limit GET POST>
Deny from all
</Limit>
```

Pour éviter qu'un attaquant demande une page protégée, autrement que par GET et POST (par exemple par une requête

http avec netcat).

Nous avons par ailleurs placé systématiquement un fichier index dans chaque répertoire pour éviter le *directory listing*.

Conclusion

Nous avons mis en place plusieurs mécanismes pour sécuriser au maximum notre blog. N'ayant pas accès à la configuration du serveur, nous n'avons pu traiter que les failles liées au code du blog lui-même.

Un point important est aussi que nous avons dû sécuriser notre application sans compter sur le client. Par exemple la librairie Bootstrap permet de vérifier par du javascript que le champ email contient bien une adresse email, et de même, CKEditor filtre le code html côté client avec du javascript, mais c'est insuffisant au niveau de la sécurité, même si c'est pratique car rapide. Nous avons donc aussi géré la vérification côté serveur en PHP (avec notamment la librairie HTML Purifier).

Nous aurions pu également mettre en place un mécanisme empêchant un utilisateur de tester trop de fois de se connecter sur un même compte. Ainsi, les tentatives de bruteforce de mot de passe auraient été empêchées. Par manque de temps nous n'avons pas implémenté cette fonctionnalité.