Corrigé - Sécurité des applications

2019-2020

L'application fournie est un blog permettant à ses utilisateurs de publier des *articles* accessible ensuite aux visiteurs. Ce corrigé liste 14 vulnérabilités présentes dans l'application web.

1 Informations sensibles

Trois fichiers contiennent des informations sensibles en dur dans le code :

- `classes/Dal/Dao.php`, avec les identifiants pour se connecter à la base de donnée,
- `classes/Utils/Crypto.php`, avec la valeur par défaut d'un sel utilisé pour le hachage,
- `mysql/setup.sql`, avec les identifiants d'un utilisateur créé par défaut.

Même si ces fichiers ne sont pas accessibles via le Web, il s'agit d'une mauvaise pratique car elle augmente le risque de fuite de ces informations.

Si l'entreprise décide d'ouvrir son code et de le diffuser, n'importe qui a alors accès aux identifiants. De même, si l'entreprise fournis cette application à plusieurs clients, ils ont chacun accès à ces informations.

Recommandation : séparer le code source et les informations sensibles. Ceci facilitera d'ailleurs le déploiement continu et les mises à jours de l'application.

2 Identifiants faibles

Certains identifiants utilisés dans l'application sont trop simples :

- `classes/Dal/Dao.php`, le mot de passe est `azerty321`,
- `mysql/setup.sql`, le mot de passe par défaut est vide.

Recommandation : Supprimer les comptes par défaut (*i.e.* dans `setup.sql`) et lorsque des mots de passes doivent être définis, les construire pour avoir une complexité suffisante.

3 Injections SQL

Dans cette application, les requêtes SQL sont construite via une fonction *helper* dans `Dal\Dao::execute()`. Cette fonction effectue la préparation de la requête puis l'exécution avec les paramètres de celle-ci. Cette manière de faire permet d'éviter les injections SQL lorsqu'elle est bien utilisée.

Ce n'est malheureusement pas le cas dans les fonctions `read()`, `delete()` et `readlast()` de la classe `Post` (les autres requêtes de cette classe, ainsi que de la classe `User` sont, par contre, correctement construites).

Un attaquant peut utiliser l'affichage d'un post en particulier (via `html/posts/show.php`) pour effectuer une injection SQL, avec pour conséquence un contrôle de la base. La méthode `delete()` est atteignable via la page `html/posts/del.php` mais cette page fait d'abord à appel à `read()` (l'injection aura alors lieu deux fois). La méthode `readlast()` n'est par contre pas exploitable puisqu'elle n'utilise aucune donnée de l'utilisateur (mais mérite d'être corrigée tout de même au cas où son paramètre soit un jour fourni par l'utilisateur).

Recommandation : Ne pas insérer de valeur dans la requête directement mais plutôt insérer un nom de paramètre dont la valeur est passée en deuxième argument de la fonction `Dao::execute()`.

4 Injection de commande

Pour effectuer les commandes systèmes, une décoration est effectuée par la fonction `Utils\Exec::cmd()`. Avant d'exécuter la commande (via `shell_exec()`), un ensemble de caractères est filtré via `escapeshellcmd()`, interdisant d'y insérer des commandes bash.

Malheureusement, `escapeshellcmd()` ne sait pas faire la distinction entre des espaces *entre* des arguments et des espaces *dans* les arguments. Un attaquant peut donc injecter plusieurs arguments et modifier le comportement des commandes lancées.

Pour cette application, si un espace est inséré dans un mot de passe, il scindera ce dernier en deux arguments pour la commande `vernam`. La deuxième partie du mot de passe sera alors considérée par celle-ci comme étant le *sel*, ce qui modifiera complètement le fonctionnement de la commande.

Recommandation : Utiliser `escapeshellarg()` sur la commande ainsi que sur chacun de ses arguments.

5 Cross Site Scripting

Aucun contrôle n'est effectué sur les données fournies par les blogueurs lorsqu'ils créent un article (via la page `html/posts/add.php`) et ensuite affichées aux visiteurs (via les pages `html/index.php`, `html/posts/index.php` et `show.php`).

Un blogueur peut ainsi insérer du `html` ou `javascript` et exploiter les navigateurs des visiteurs.

Recommandation : Avant insertion des données, échapper tout caractère `html`. Si ceux-ci sont nécessaire, utiliser un filtre (i.e. `HTMLPurifier`).

6 Cross Site Request Forgery

Les pages effectuant des opérations sur des données ne sont jamais protégées contre une attaque de type CSRF.

Un attaquant pourrait utiliser des attaques XSS pour que ses victimes demandent l'exécution de ces pages à leur insu (*i.e.* créer et supprimer des pages, enregistrer de nouveaux utilisateurs, se déconnecter).

Recommandations : Ajouter des contrôles spécifiques (*e.g.* captcha, mots de passes, tokens).

7 Authentification

L'application ne pose aucune contrainte d'unicité du nom d'utilisateur. Ni dans la structure de la base (visible dans `mysql/setup.sql`), ni dans la gestion des utilisateurs (dans la classe `classes/Model/User.php`).

Il est possible de créer un deuxième utilisateur ayant le même nom qu'une victime (fonction `User::register()`). Puisque c'est le nom (et non l'objet ou son `id`) qui est stocké en session, l'application ne fera pas de différence entre ces utilisateurs.

Recommandation : Il est recommandé d'intégrer une contrainte d'unicité et d'utiliser le champ `id` plutôt que le nom d'utilisateur pour identifier ceux-ci.

8 Contrôle d'accès défectueux

Un contrôle d'accès est mis en place pour les pages nécessitant un compte sur le système (ajout et suppression d'un article).

Malheureusement, ce contrôle est défectueux lors de la suppression des articles. Cette tâche s'effectue dans le fichier `html/posts/del.php` et le contrôle est fait dans la ligne 5 que voici (ainsi que les 3 lignes suivantes)

```
if (\Model\User::get() != $post->author && isset($_POST["confirm"])) {
    $post->delete() ;
    \View\Page::redirect("/post/index.php");
}
```

Le comparateur `!=` utilisé ici signifie que seul l'auteur de l'article ne peut pas supprimer l'article. N'importe qui d'autre, même sans être authentifié, peut supprimer un article.

Recommandation : Corriger la comparaison et remplacer le `!=` par un `===`.

9 Fixation de session

La session de l'utilisateur est correctement stockée côté serveur via `$_SESSION` mais elle n'est générée qu'une fois pour toute. Un attaquant pouvant forcer un identifiant de session chez une victime pourra ensuite lui voler sa session.

Recommandation : Pour éviter une attaque, il est nécessaire d'ajouter l'appel à la fonction PHP `session_regenerate_id()` après toute authentification (fin des fonctions `User::register()` et `User::login()`).

10 Redirection arbitraire

Lors de l'exécution de certaines pages (*i.e.* `html/users/login.php`), l'application envoie une réponse de type *redirection* en utilisant l'en-tête REFERER de la requête sans la contrôler.

Un attaquant exploitant une XSS pourrait forcer une victime à naviguer sur cette page et, après avoir forgé l'en-tête, forcer une redirection sur un site de son choix.

Recommandations : Ne pas rediriger les visiteurs sans contrôler les adresses cibles.

11 Cryptographie défectueuse

Le code `vernam.c` est sensé effectuer un chiffrement *de Vernam*. Ce chiffrement est effectivement *incassable* (et prouvé comme tel en 1940) mais nécessite pour cela le respect de trois conditions dont aucune n'est ici satisfaite.

1. **La clé doit être aussi longue que le message,** mais les mots de passes n'ayant aucune contrainte de taille, certains pourraient être plus long que la clé,
2. **La clé doit être choisie aléatoirement,** ici, c'est en fait le résultat, en hexadécimal, de l'empreinte `MD5` de la chaine *42*,
3. **La clé doit être à usage unique,** or ici, elle est réutilisée systématiquement.

Rien que ce troisième point est ici problématique car un attaquant connaissant un mot de passe pourra rétro-concevoir la clé et ensuite déchiffrer l'ensemble des mots de passes.

On peut également noter que l'implémentation n'est pas conforme aux spécification. L'opération pour combiner la clé et le message n'étant pas bijective, plusieurs mots de passes, une fois chiffrés, pourront avoir le même résultat.

```
((data[i] + mask[i%strlen(mask)]) % 26 ) + 'A'
```

Recommandation : Vérifier les implémentations et surtout les contraintes d'utilisation des algorithmes cryptographiques afin de les utiliser à bon escient.

12 Stockage du mot de passe

En admettant que la primitive *vernam* soit correctement implémentée, elle n'est pas adaptée au cas d'usage du stockage des mots de passes.

Le but étant de stocker les *chiffrés* pour les déchiffrer ensuite, la clé utilisée (même si elle était de la bonne taille, aléatoire et unique), devrait également être stockée. Un attaquant ayant accès à la base de donnée pourra alors déchiffrer les mots de passe.

Recommandation : Lors du stockage des mots de passes, utiliser des algorithmes de Hachage (et non de chiffrement) ainsi qu'un sel choisi aléatoirement. Par simplicité, il est conseillé d'utiliser les fonctions `password_hash()` et `password_verify()` de PHP.

13 Débordement dans le tas

Le code `vernam.c` effectue deux copies de chaînes via `strcpy()` aux lignes 28 et 31 sans vérifier que les zones mémoires d'arrivées sont suffisamment grandes pour accueillir les données.

Un attaquant pourrait, via un mot de passe bien choisi, écraser les structures internes de `malloc` pour détourner le flux d'exécution.

Recommandation : Utiliser `strncpy()` qui évitera de déborder et/ou dimensionner correctement les zones mémoires.

14 Double *free*

Le code `vernam.c` effectue deux `free()` du même pointeur.

En l'état, le code n'est pas exploitable. D'une part les deux *free* sont consécutifs et seront détectés mais les zones n'étant pas ensuite ré-allouée, la corruption des structures de `malloc` ne sera pas *utilisée*.

Recommandation : Libérer une seule fois chaque zone mémoire. Utiliser des outils d'analyses statiques et/ou dynamique peut détecter ces erreurs.

15 Variables non initialisées

Le code `vernam.c` déclare deux variables (`i` et `j` en ligne 15) sans les initialiser.

En l'état, cette erreur n'est pas exploitable (i est initialisé avant d'être utilisé et j n'est pas utilisé du tout). Sa correction est néanmoins importante pour éviter des régressions éventuelles.

Recommandation : toujours initialiser les variables.

16 Bugs divers

Les bugs listés ici ne remettent pas en cause la sécurité de l'application web mais empêchent certaines page de fonctionner correctement.

Erreur d'inclusion. Le fichier `html/index.php` en ligne 3 inclus `../autoload.php` au lieu d'inclure `./autoload.php`. Cette erreur empêche la page d'accueil de fonctionner.

Erreur de formulaire. Le fichier `html/posts/del.php` contient le même formulaire que le fichier `html/posts/add.php` (erreur de duplication de code). Le formulaire n'étant pas cohérent avec les données attendue, cette page ne fonctionnera pas pour les visiteurs (mais elle pourra être utilisée via des scripts).

Erreur de nom. La méthode `hash` de la classe `Crypto` est appelée avec une majuscule et définie en minuscule. Le `PHP` n'étant pas sensible à la casse, le code s'exécutera comme attendu, mais peut poser problème pour la relecture du code par les développeurs.