Corrigé - Sécurité des applications

Thibaut HENIN INSA - Centre Val de Loire

2018/2019

1 Contexte et mission

L'entreprise *Speed-e-dev* a développé un moteur de blog en `PHP` utilisant `SQL` pour le stockage et un binaire fait maison en `C` pour une primitive cryptographique. Il nous est demandé d'analyser le code source de l'application à la recherche de vulnérabilités.

2 Vulnérabilités découvertes

Cette application contient 13 vulnérabilités détaillées ci-après.

2.1 Sessions

Les sessions utilisateurs sont gérées dans le fichier `utils.php` par les fonctions `setUser()` (ligne 27) et `getUser()` (ligne 31) qui stockent et lisent les données de session (*i.e.* l'identité de l'utilisateur) directement dans un cookie. Ces fonctions sont ensuite utilisées pour le contrôle d'accès des fichiers `headers.php`, `login.php` et `show.php`.

Cette méthode n'est pas sûre car n'importe quel client peut forger ses cookies pour qu'il contienne des données arbitraires et ainsi, usurper n'importe quel utilisateur.

Il serait plus sûr de stocker les données de session côté serveur et non côté client via les mécanismes prévus par les langages web (*i.e.* `session_start()` en `PHP`).

2.2 SQL injection

De nombreuses requêtes `SQL` effectuées par l'applications sont vulnérables à une attaque par injection `SQL` :

- `add.php` en ligne 6,
- `delete.php` en ligne 7,
- `login.php` en ligne 7,
- `show.php` en ligne 10.

Un attaquant peut insérer dans les valeurs soumises via les formulaires, des fragments de requêtes `SQL` qui modifieront le comportement des requêtes originale et lui fourniront des accès en lecture et écriture sur l'ensemble de la base. Il serait plus prudent d'utiliser des requêtes préparées.

La requête effectuée par le fichier `list.php` en ligne 7 n'est par contre pas vulnérable puisque cette requête n'utilise aucun paramètre ni donnée client.

2.3 Cross Site Scripting

Les données fournies par les utilisateur lors de l'insertion (fichier `add.php`) des *posts* ne sont ni filtrées ni échappées et rendues telles quelles lorsque les visiteurs affichent ces derniers (fichiers `list.php` et `show.php`).

Un attaquant ajoutant un *post* pourrait donc ajouter du contenu arbitraire exploitant les visiteurs légitimes. Les possibilités sont multiples : phishing, vol de données, exécution de code sur les postes clients.

Il serait plus sûr de filtrer les données pour soit refuser toute balise HTML, soit purifier son contenu.

2.4 Cross Site Request Forgery

Aucun formulaire ne vérifie la légitimité des requêtes reçues avant d'exécuter son action (fichiers `add.php`, `delete.php`, `login.php`, `logout.php` et `upload.php`).

Un attaquant piégeant un utilisateur légitime via une attaque de type *Cross Site Scripting* (y compris sur une autre application que celle-ci) peut faire exécuter ces formulaires à l'insu de l'utilisateur.

Il serait plus sûr de protéger l'exécution de ces formulaires par des contrôles spécifiques (*i.e.* token, captcha, mot de passe, ...).

2.5 Contrôle d'accès

Bien qu'un contrôle soit effectué lors du rendu des pages web concernant les liens vers les scripts critiques (fichier `headers.php` en ligne 7), aucun contrôle n'est ensuite effectué dans les formulaires (fichiers `add.php`, `delete.php` et `upload.php`).

Un attaquant peut donc accéder directement à ces formulaire et lancer leur exécution. Bien que le contrôle lors de l'affichage soit pertinent, un contrôle systématique pour tous les formulaire est nécessaire.

2.6 Fuite d'informations

Deux fichiers critiques ne sont pas correctement protégés. Un attaquant pourrait y demander l'accès au serveur web, celui-ci lui affichant alors son contenu.

- `config.ini` contient les paramètres d'accès à la base de donnée (hôte, login et mot de passe),
- `utils.inc` contient la clé de chiffrement des mots de passes (ligne 13).

Ces fichiers devrait se trouver dans un dossier inaccessible via une requête Web, afin d'éviter tout risque d'accès frauduleux.

2.7 Faille *include*

Le fichier `index.php`, agissant comme un *front controler* se charge d'inclure les scripts (ligne 9) en fonction des requêtes utilisateurs sans effectuer de contrôle.

Un attaquant peut donc demander l'inclusion de n'importe quel fichier arbitraire, local au serveur mais également à distance, lui permettant ainsi d'exécuter un code arbitraire.

Il serait plus sûr d'éviter des inclusions de fichiers fournis par le client, ou d'utiliser une liste blanche des pages possibles.

2.8 Faille *upload*

Le script `upload.php` permet aux utilisateur de téléverser des fichiers à l'application. Malheureusement, aucun contrôle n'est effectué sur le type de fichier et sur l'emplacement où ils seront copiés.

Un attaquant peut donc écraser du contenu légitime et ajouter ses propres scripts à l'application, lui permettant d'exécuter du code arbitraire.

Il serait plus pertinent de supprimer ce type de fonctionnalité où, à minima, de contrôle le type de fichier et les emplacements où ils sont sauvegardés.

2.9 Injection d'objet

Le cookie utilisé par l'application (fichier `utils.php`) contient des données *sérialisées* (ligne 27). Un attaquant peut alors forger ses propres données qui seront *désérialisées* par l'application (ligne 31).

Cette vulnérabilité n'est ici pas exploitable puisque l'application n'utilise aucune classe mais il s'agit d'un code dangereux ; si une nouvelle version de l'application venait à inclure des objets utilisables par un attaquant, l'application serait alors vulnérable.

La désérialisation d'objets fournis par l'utilisateur doit être évitée à tout prix.

2.10 Injection de commandes

L'utilisation du script de *codage* est effectuée via la fonction `shell_exec()` (fichier `utils.php` en ligne 24). Aucun filtrage ni échappement des paramètres n'étant effectué, un attaquant peut injecter des fragments de commandes pour que celles-ci soit exécutées.

L'utilisation de paramètres fournis par l'utilisateurs devrait être échapée (*i.e.* via la fonction `escapeshellarg()`).

2.11 Cryptographie faible

Le stockage des mots de passes n'est pas sûr (il s'agit d'un *rot13*). Un attaquant ayant un accès en lecture à la base de donnée peut ainsi en déchiffrer son contenu facilement.

Le stockage des mots de passes devrait utiliser des algorithmes de hachage sûrs (*i.e. sha512*) et faire intervenir un sel.

2.12 Débordement de tableau sur la pile

Le fichier `codage.c` est vulnérable à une attaque par débordement de tableau dans la pile lors de la copie du mot de passe dans un tableau temporaire (ligne 25).

Un attaquant fournissant un mot de passe de plus de 1024 caractères débordera du tableau et pourrait écraser des données critiques. Ici l'adresse de retour de la fonction `main()` lui fournissant des possibilité d'exécution de code arbitraire.

Il serait plus pertinent de tronquer la copie (*i.e.* `strncpy()`) ou de vérifier la taille des tableaux avant la copie.

2.13 Chaîne de format

Le fichier `codage.c` est vulnérable à une attaque par *chaine de format* lors de l'écriture du résultat sur la sortie standard (ligne 28).

Un attaquant pourrait fournir un format particulièrement conçu en tant que mot de passe et obtenir ainsi une écriture arbitraire dans la mémoire du processus et donc une exécution de code arbitraire. Subtilité ici, le format devra être chiffré par un *rot13* avant d'être envoyé.

Pour éviter ces vulnérabilités, il faudrait éviter d'utiliser une donnée fournie ou dérivée depuis une donnée fournie par un utilisateur en tant que premier paramètre.

3 Conclusion

L'analyse du code source fait ressortir 13 vulnérabilités critiques permettant à un attaquant d'obtenir un accès en lecture, écriture et exécution de code arbitraire sur le serveur et les navigateurs des clients.

Il est donc très fortement déconseillé de mettre cette application en ligne avant la correction de ces vulnérabilités.