Document Étudiant : Attaques Web Avancées et Méthodes de Protection

Module 5 : Les injections SQL

Les injections SQL constituent une faille de sécurité très dangereuse.

- **Qu'est-ce qu'une injection SQL ?** Le principe est de **détourner une requête SQL légitime** afin d'en faire exécuter une autre par le serveur, qui n'était initialement pas autorisée.
    - **Exemple simple :** Si une application utilise une requête comme `select Nom from etudiants where id=ID_SAISI;` où `ID_SAISI` est une valeur entrée par l'utilisateur, un attaquant peut saisir `' or 1=1#` comme identifiant. La requête devient alors `select Nom from etudiants where id='' or 1=1# ;`. Le `#` (ou `--` dans d'autres SGBD) commente la fin de la requête originale, et `1=1` étant toujours vrai, la clause `where` devient `id='' or TRUE`, permettant potentiellement de récupérer tous les noms de la table `etudiants`.
    - **Utilisation de `UNION` :** L'opérateur `UNION` peut être utilisé pour combiner le résultat de la requête originale avec celui d'une autre requête malicieuse, permettant d'accéder à d'autres champs ou tables. Par exemple, en saisissant `' or 1=1 union select Prenom from etudiants #`.
- **Types d'injections SQL :** Il existe deux principaux types :
    - **SQLi classique :** Se produit quand la page web **affiche des informations** basées sur la requête injectée. Il est rapide de voir l'impact de l'injection. La détection se fait par essais successifs et observation du comportement de l'application. Pour aller plus loin, on peut utiliser `GROUP BY` pour deviner le nombre de champs, interroger la base `information_schema` pour obtenir les noms des tables et champs, et utiliser `UNION` pour extraire des données. Des outils comme `sqlmap` simplifient grandement cette extraction.
    - **SQLi en aveugle (Blind SQL injection) :** L'application ne renvoie **aucune information directe** sur l'erreur ou les données. L'objectif est d'obtenir des informations (tables, champs, version de la base) en posant des questions binaires (vrai/faux) à la base de données.
        - Si le serveur renvoie une indication (même minime) d'erreur ou de succès, on peut l'utiliser.
        - Si le serveur ne renvoie rien, on doit se baser sur son **temps de réponse** (`time based`).
        - Des techniques impliquent l'utilisation de `GROUP BY` pour tester le nombre ou le nom des champs, et `LENGTH` ou `SUBSTRING` pour déterminer la longueur ou le contenu d'un champ. Pour le `time based`, des commandes comme `SLEEP(time)` ou `BENCHMARK(count, expr)` (pour MySQL) sont utilisées pour introduire des délais conditionnels.
        - L'extraction manuelle est très longue ; il est préférable d'utiliser des outils (comme `sqlmap` avec les bonnes options) ou de composer son propre script d'extraction.
- **Comment se protéger contre les injections SQL ?**
    - Donner les **droits minimums** à l'utilisateur de la base de données.

    - **Filtrer les requêtes** ou utiliser une API de préparation. Les fonctions comme `mysql_real_escape_string` (attention, cette fonction est spécifique à l'ancienne extension `mysql` et est souvent remplacée par des approches plus modernes comme PDO) ou l'utilisation de **`PDO::prepare`** sont essentielles pour préparer les requêtes. Les requêtes préparées séparent le code SQL des données utilisateur, empêchant l'injection.
    - Donner le **moins d'informations possible** à l'utilisateur en cas d'erreur.
- **Exercices (Approche Python) :** Les sources indiquent que **Python est un langage très adapté** pour développer ses propres outils d'audit, notamment grâce à ses nombreuses bibliothèques pour le Web comme `Requests` qui permet de forger des requêtes HTTP. Bien que les sources ne fournissent pas le code Python *des exercices*, elles décrivent les techniques d'attaque, ce qui permet de concevoir des scripts d'audit.
    - **Exercice 5.1 (SQLi classique) :** Créez un script Python utilisant la bibliothèque `requests` pour envoyer différentes charges (`payloads`) d'injection SQL classique (comme `' or 1=1#`, `UNION SELECT ...`) à une URL théoriquement vulnérable via une requête GET ou POST. Analysez la réponse HTTP pour détecter un changement de comportement ou l'affichage d'informations inattendues qui pourraient indiquer une vulnérabilité.
    - **Exercice 5.2 (Blind SQLi conceptuel / Python) :** Décrivez (ou commencez à coder) un script Python qui pourrait tester une Blind SQL Injection. Par exemple, pour le `time based`, le script enverrait des requêtes contenant `SLEEP(temps)` dans le paramètre vulnérable et mesurerait le temps de réponse du serveur pour déduire si la condition injectée était vraie. Le script pourrait itérer sur des longueurs de chaînes ou des caractères pour extraire progressivement des informations (basé sur).

Module 6 : Les failles include et upload

- **La faille LFI (Local File Inclusion)** Cette faille permet d'**inclure (et potentiellement exécuter) le contenu d'un fichier** dans un autre script serveur, en utilisant des instructions comme `require()`, `require_once()`, `include()`, `include_once()`. Une erreur d'inclusion avec `include` ne stoppe pas le script, contrairement à `require`.
    - **Principe :** Si un script inclut un fichier dont le nom est spécifié par l'utilisateur dans une URL (ex: `http://site.com/index.php?page=mapage.php`), sans protection adéquate, un attaquant peut modifier l'URL pour inclure un fichier système (ex: `http://site.com/index.php?page=/etc/passwd`).
    - **Protections et Contournements :**
        - Protection basique (ajouter une extension). Facile à contourner en utilisant `../` pour remonter dans l'arborescence.
        - Réécriture des extensions de fichier. Peut être contournée en ajoutant un caractère `null byte` (`%00`) après le chemin du fichier souhaité (fonctionne sur d'anciennes versions de PHP < 5.3.4).
        - Lecture de fichiers source : Même si un fichier n'est pas exécuté mais interprété, on peut lire son contenu en utilisant les **wrappers PHP** comme `php://filter/resource=/chemin/fichier.php`.

        - Exécution de code arbitraire : Si la directive `allow_url_include` est activée sur le serveur PHP, on peut exécuter du code directement via un wrapper comme `data://text/plain;base64,CODE_BASE64`.
- **La faille upload** La fonctionnalité d'upload de fichiers est courante (avatar, CV, images) mais un manque de restriction conduit souvent à la faille **Unrestricted file upload**.
    - **Conséquences :** Très critiques si le fichier peut être exécuté (défacement, exécution de commandes, accès BDD, escalade de privilèges). Dangereuses même si non exécuté (hébergement de malware, contenu illégal, phishing, déni de service par gros fichiers, remplacement de fichiers importants).
    - **Protections et Contournements :**
        - Vérification du type MIME : Facile à contourner en modifiant le type MIME dans la requête HTTP avec des outils comme Burpsuite, ZAP, Tamper Data.
        - Utilisation d'une image : Même si le fichier est traité comme une image, il peut être possible d'y intégrer du code malicieux (par exemple, en utilisant un outil comme `jhead` pour insérer du code PHP dans les métadonnées d'une image) et de le faire exécuter si d'autres conditions sont remplies (comme un fichier `.htaccess` autorisant l'exécution de code dans le répertoire d'upload).
    - **Webshells :** Un fichier uploadé peut être un "webshell", une interface web permettant d'exécuter des commandes sur le serveur. Les basiques sont peu pratiques ; il existe des webshells avancés.
- **Conclusion du module :** L'utilisation des wrappers PHP et l'upload de fichiers peuvent être très dangereux. Il est essentiel de bien régler les options serveur et de filtrer correctement les données. La réécriture des extensions de fichier est une bonne pratique.
- **Exercices (Approche Python) :** Python et des bibliothèques comme `requests` ou même `selenium` peuvent être utilisées pour automatiser les tests de ces vulnérabilités.
    - **Exercice 6.1 (LFI Bypass) :** Écrivez un script Python qui tente d'exploiter une faille LFI théorique en testant différents chemins relatifs (`../../`) pour inclure des fichiers système connus (`/etc/passwd`, `/etc/shadow`). Utilisez `requests` pour envoyer les requêtes et analysez la réponse pour vérifier si le contenu du fichier est affiché.
    - **Exercice 6.2 (Upload Bypass - MIME) :** Créez un script Python utilisant `requests` pour simuler l'upload d'un fichier potentiellement malicieux (par exemple, un simple fichier texte renommé en `.php` ou `.asp`) en modifiant l'en-tête `Content-Type` pour qu'il ressemble à un type MIME autorisé (ex: `image/jpeg`). Testez différentes extensions de fichiers et types MIME.

Module 7 : Plate-forme d'entrainement

Pour vous entraîner à trouver des failles de sécurité, il existe des plateformes dédiées.

- **En ligne :** http://www.root-me.org/
- **Installables en local :** OWASP WebGoat (https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project) ou bwapp, dvwa.

Il est crucial de pratiquer sur ces plateformes sécurisées pour appliquer ce que vous avez appris.

Module 8 : Comment se protéger

Les failles de sécurité Web proviennent principalement de **mauvais réglages du serveur**, d'**erreurs de développement** et d'un **manque de filtrage des données** envoyées par l'utilisateur.

- **Configuration Serveur / Langage :** Limitez toujours les actions possibles des serveurs (Apache, PHP, MySQL) au strict minimum nécessaire au bon fonctionnement de l'application. Cela doit être défini dans le cahier des charges.
- **Filtrage des données :** C'est une consigne de sécurité qui revient régulièrement et est primordiale. Il peut être effectué :
    - **Dans le code :** En utilisant des **expressions régulières** (`regex`) ou des fonctions spécifiques au langage (comme les fonctions PHP `preg_*` pour les regex, `htmlspecialchars`, `htmlentities` contre les XSS, ou `PDO::prepare` contre les injections SQL). Les regex permettent de limiter ce que l'utilisateur peut saisir au strict nécessaire (ex: chiffres pour un numéro de téléphone).
    - **Avant l'application web :** En utilisant un matériel ou une application de filtrage appelée **WAF (Web Application Firewall)**.
- **WAF (Web Application Firewall) :** Un WAF filtre les données *avant* qu'elles n'atteignent l'application web.
    - **Avantages :** Répartit la charge de filtrage, protège les applications mal développées, permet de modifier les règles de filtrage sans toucher à l'application.
    - **Catégories :** WAF matériels (DenyAll, FortiWeb) et WAF logiciels (versions VM ou solutions gratuites comme le module `mod_security` d'Apache).
    - **ModSecurity :** Un WAF logiciel populaire pour Apache. Il utilise des règles, notamment les **OWASP CRS (Core Rule Set)**. Les règles sont définies par des directives comme `SecRuleEngine`, `SecAction`, `SecRule`.

Conclusion Générale

Les attaques par injection SQL et les failles liées aux fichiers (inclusion, upload) sont des menaces courantes et dangereuses. La protection repose sur une combinaison de bonnes pratiques : une **configuration serveur sécurisée** (droits minimums, options restreintes), un **développement sécurisé** (utilisation de requêtes préparées, fonctions de filtrage adaptées), et un **filtrage rigoureux de toutes les données utilisateurs** (par regex, fonctions dédiées, ou WAF). Aucune application automatique ne remplace le raisonnement humain pour imaginer des scénarios d'attaque, mais les outils (y compris les scripts Python personnalisés) et les plateformes d'entraînement sont essentiels pour auditer et comprendre ces vulnérabilités.