

Attaques par injection de commandes

Injection de commandes

- Qu'est-ce c'est?
- Comment localiser les points vulnérables?
- Comment l'exploiter?
- Comment s'en prévenir?

Injection de commandes vs. injection de code

- **Injection de code:**
 - piratage d'une application pour la faire exécuter du code malveillant
- **Injection de commandes:**
 - utilisation de code existant ou d'outils pour exécuter des **commandes shell**
 - conséquence: limité par le shell disponible et les programmes installés

L'injection de commandes fait partie de la famille des attaques par injection.
Les attaques par injection sont en 3ème position du top 10 OWASP 2021.

Example

```
import java.io.IOException;
import javax.servlet.http.HttpServletRequest;

public void runCmd(HttpServletRequest request) throws IOException {
    String command = request.getParameter("cmd");
    String arguments = request.getParameter("args");
    Runtime.getRuntime().exec(command + " " + arguments);
}
```

Les composantes de l'injection de commandes

- Sur une application web, 3 composantes:
 - la **requête web** contenant une entrée utilisateur qui sera ensuite utilisée dans une commande shell côté serveur
 - la **commande injectée** elle-même
 - la **réponse** du serveur

Impacts potentiels

- Accès non autorisé à l'application et à l'OS hôte
 - **Confidentialité**: accès non autorisé à des données
 - **Intégrité**: modification de données
 - **Disponibilité**: suppression de données, rendant indisponibles certains services de l'application ou l'application elle-même

Catégories d'injection de commandes

Deux grandes catégories

- **Injection basée sur le résultat** (*In-band Command Injection*)
- **Injection aveugle** (*Blind Command Injection*)

Injection basée sur le résultat

- *In-band Command Injection*
- L'attaquant peut **visualiser** la sortie de la commande via la réponse de l'application

Injection aveugle

- *Blind Command Injection*
- L'attaquant ne peut pas visualiser le résultat de la commande (pas de composante «réponse»)
- **Le code a cependant bien été exécuté** sur le serveur
- Mais plus difficile à exploiter, car on n'a pas de retour direct

Localiser les vulnérabilités

Approches de Pentesting

- Trois grandes approches, qui différencient les informations préalables dont disposent le pentester :
 - **Boîte noire (*Black Box*)**: aucune information sur le système
 - **Boîte blanche (*White Box*)**: informations concernant le système, **y compris le code source** de l'application
 - **Boîte grise (*Gray Box*)**: une partie des informations

Pentesting en boîte noire

- **Cartographier l'application**
 - **parcourir** tout le site web, page par page
 - pendant ce travail, un **proxy web intercepte et enregistre toutes les requêtes**
 - **identifier** toutes les instances où l'application semble interagir avec l'OS sous-jacent à partir d'entrées utilisateur

Boîte noire et injection *in-band*

- **Analyser la réponse**
- Parfois évident car l'application renvoie directement la sortie d'une commande
- Souvent plus subtil :
 - retour vide
 - erreur
 - résultat de la commande peut être :
 - partiel
 - formaté
 - transformé
 - « fondu » dans la vue

Boîte noire et injection *blind*

- Comme pas de retour évident, il faut être plus créatif
 - forcer un délai pour savoir si ça fonctionne
 - forcer la sortie de la commande sur le répertoire racine du serveur et récupérer le fichier directement avec le navigateur
 - ouvrir et utiliser un canal *out-of-band* vers un serveur que l'on contrôle

Pentesting en boîte blanche

- **Cartographier comme en boîte noire** pour déterminer tous les potentiels vecteurs d'entrée
- Revue du code source :
 - déterminer si ces entrées sont utilisées en **paramètres de fonctions** qui exécutent des **commandes systèmes**
 - déterminer si les **éventuelles protections** contre les entrées malveillantes sont efficaces *et* **systematiquement** mises en place
- **Tester *en live*** chaque vulnérabilité potentielle découverte

Exploitation d'une vulnérabilité en injection de commandes

Fonctionnement

- De nombreuses injections de commandes fonctionnent simplement **en ajoutant à la commande initiale une commande « personnalisée »**
 - cette commande malveillante va alors être exécutée **en plus** de la commande prévue
- Pour cela, il faut utiliser des **méta-caractères**
 - ceux-ci permettent de venir augmenter la commande initiale en incluant la commande supplémentaire

Les méta-caractères

- Les injections de commandes dépendent de caractères spéciaux ou de séquences de caractères spéciaux, appelés **méta-caractères**.
- Ex: &, &&, ;, |, ||, >, *, \, \n, \$()

Séquence de commandes

- On peut écrire en une seule ligne plusieurs commandes shell qui seront exécutées les unes à la suite des autres
- Il suffit de séparer les commandes par:
 - un *ampersand* & (Windows)
 - un point-virgule ; (Linux)

Pipe

- Le *pipe* | («tuyau») est utilisé pour envoyer la sortie d'une commande vers l'entrée d'une autre
- Exemples:
 - `ls | grep "test"`
 - `cat /etc/passwd | grep root`
 - `cat /etc/passwd | cut -d: -f1`

Double *pipe*

- Le double *pipe* (| |) est utilisé pour exécuter une commande si la précédente a échoué (OU logique)
- Exemple:
 - `rm un_fichier_absent || echo "Le fichier n'a pas pu être supprimé"`

Wildcard

- L'astérisque * est utilisé pour représenter n'importe quelle suite de caractères
- Autres noms: «étoile», *star*, *wildcard*, *glob*
- Exemples:
 - `ls *.txt`
 - `rm -f *`

Redirection

- La redirection (>) permet de rediriger la sortie d'une commande vers un fichier
- Exemples:
 - `ls > liste.txt`
 - `head liste_films > top_films`

Échappement de caractère

- Pour échapper un caractère spécial, on utilise le caractère \
- Exemple:
 - `echo -e "Ceci affiche deux backslashes \\ \\ et saute 3 lignes\n\n\n"`

Attaques web en injection de commandes

Exemple avec une requête GET

```
https://lesite.fr/dns.php?domain=undomaine.com
```

```
Server: 192.168.25.1  
Address: 192.168.25.1#53
```

```
Non-authoritative answer:  
Name: undomaine.com  
Address: 81.92.80.56
```

Attaque avec ce vecteur

```
https://lesite.fr/dns.php?domain=undomaine.com;id
```

```
Server: 192.168.25.1  
Address: 192.168.25.1#53
```

```
Non-authoritative answer:  
Name: undomaine.com  
Address: 81.92.80.56
```

```
uid=99 (apache) gid=99 (apache) groups=99 (apache)
```

- L'attaquant connaît alors le système d'exploitation répondant aux requêtes
 - ; pour séparer les commandes \Rightarrow Linux
- L'attaquant peut ensuite injecter *d'autres commandes shell*
 - si `id` fonctionne, toutes les commandes auxquelles l'utilisateur *apache* peut accéder pourront être exécutées, avec les droits de cet utilisateur

Qu'en est-il si le système avait répondu ceci?

```
'id' is not recognized as an internal or external command, operable program or batch file.
```

Même exemple avec POST

```
POST /dns.php HTTP/1.1
Host: lesite.fr
[...]
Cookie: PHPSESSID=1234567890abcdef
Authorization: Basic YWRtaW46cGFzc3dvcmQ=
Content-Type: application/x-www-form-urlencoded
Content-Length: 20

Domain=undomaine.com
```


Localiser une vulnérabilité dans le code d'une application web (Python)

Localiser une vulnérabilité dans le code d'un CMS (PHP)

```
if (isset($_POST['Submit'])) {  
    $target = $_REQUEST['ip'];  
    $substitutions = array(  
        '&&' => '',  
        ';' => '',  
        '|' => ''  
    );  
    $target = str_replace(array_keys($substitutions), $substitutions, $target);  
    $res = shell_exec('ping -c 3 ' . $target);  
    echo "<pre>{$res}</pre>";  
}
```

Identifier les points d'entrée potentiels

Point d'entrée

- On regarde cette fois du point de vue du code client
- On cherche les points d'entrée potentiels qui pourrait permettre une injection de commande:
 - champs de formulaire
 - en-têtes de requête HTTP
 - paramètres d'URL
 - *body* de HTTP POST
 - etc.

Contre-mesures (1)

- Contre-mesure principale: *ne pas* utiliser de commande shell du tout (en tout cas pas avec des entrées utilisateur)
- Préférer:
 - les fonctions natives du langage ou du *framework* utilisé
 - ou bien une bibliothèque externe

Contre-mesure (2)

- L'une des causes principales qui rendent l'injection de commandes possible est le manque d'attention portée aux entrées utilisateur (qui peuvent alors «se confondre» avec des commandes shell)
- Donc, *si l'utilisation de commandes shell est inévitable*, il faut **valider** et **assainir** les entrées

Validation et assainissement des entrées

Assainissement

- ***Sanitization*** en anglais
- **L'assainissement** s'occupe de la suppression des caractères *unsafe* (non autorisés)
 - **liste blanche**: on autorise *uniquement* les caractères autorisés
 - **liste noire**: on interdit les caractères interdits
 - **échappement**: on échappe les caractères interdits, ce qui neutralise un potentiel effet néfaste dans un contexte d'exécution

Validation

- La **validation** s'occupe de vérifier que les données sont conformes à un format attendu
 - **syntaxe**: l'entrée doit être conforme au niveau syntaxique et structurel
 - ex: no. passeport français: 2 chiffres + 2 lettres + 5 chiffres
 - **sémantique**: l'entrée doit être conforme au niveau du sens
 - ex: année de naissance d'un utilisateur < 1900 ou > année courante
 - ex: adresse IP = 4 nombres entre 0 et 255 séparés par des points, MAIS certaines plages ne sont pas utilisées

Validation côté client

- La validation côté client est une **bonne pratique** :
 - UX (expérience utilisateur): l'utilisateur est informé immédiatement d'une erreur, sans *roundtrip* vers le serveur
 - charge réduite du serveur: moins de requêtes inutiles qui arrivent grâce à la pré-validation
- Mais **elle ne doit pas être considérée comme une protection**
 - souvent incomplète (on n'utilise pas forcément toute la panoplie de validation disponible, notamment pour éviter d'avoir à charger de lourdes bibliothèques JS de validation)
 - **facilement contournable** (désactivation du JS, interception des requêtes, forge de requête...)

Validation et assainissement
Lequel effectuer en premier?

Validation d'abord

- Il faut toujours valider les entrées avant de les assainir
 - inutile de lancer un traitement d'assainissement si l'entrée n'est même pas valide
 - pire: assainir l'entrée court le risque de modifier et avoir des conséquences néfastes

Exemple: champ "nom"

- Vous entrez votre nom avec un point-virgule (erreur de frappe)
- Le système devrait vous signaler que l'entrée n'est pas valide
 - et vous permettre de corriger
- Si assainissement d'abord, le système va «corriger» de manière silencieuse
 - et, dans certains cas, accepter des entrées différentes de ce que l'utilisateur souhaitait

Principe de moindre privilège (PMP)

- Concept de sécurité par lequel une entité est limitée à des accès restreints et bien spécifiés, qui correspondent aux ressources strictement nécessaires à l'accomplissement de sa tâche
 - l'entité peut être de types très variés: compte utilisateur, équipement réseau, application web...
- Ce principe est l'une des meilleures armes, en général, contre de nombreuses attaques

PMP et injection de commandes

- Imaginez un serveur web vulnérable aux injections de commandes
- Depuis cet accès, l'attaquant va naturellement:
 - faire de la reconnaissance
 - tenter le mouvement latéral
- Si le PMP est correctement implémenté, l'attaquant va voir ses possibilités largement restreintes
- Connaissez-vous les accès dont disposent vos serveurs Apache ou Nginx?

Accès du serveur web

- Le serveur web ne devrait pas avoir d'accès :
 - administrateur
 - à des postes de travail
 - à d'autres serveurs non nécessaires à son fonctionnement (parefeu, mail...)
- En général, le serveur web n'a besoin que d'accès :
 - en lecture à ses fichiers de configuration
 - en lecture/écriture à ses fichiers de logs, session, cache, temporaires, certificats...
 - à la base de données (qui elle-même devrait avoir des accès restreints en général)

Contrôles techniques

- Les contrôles techniques sont des mesures de sécurité qui sont mises en place pour protéger les systèmes et les données
 - parefeu, antivirus, chiffrement...
 - Journal d'événements (*audit logs*)
 - activités utilisateurs, systèmes, réseaux, applications...
 - ACLs (*Access Control List*)
 - de système de fichiers (instructions à l'OS)
 - de réseau (instructions au commutateurs/routeurs)
 - Parefeu d'application web (WAF)

WAF

- *Web Application Firewall*
 - surveille et filtre le **trafic HTTP/S entrant et sortant** d'une application web protégée
 - tout trafic va être soumis à la **politique de sécurité** définie sur le WAF
- Peut être déployé en mode:
 - **réseau/matériel**: en tant qu'équipement dédié
 - **hôte/logiciel**: en tant qu'application sur le serveur web lui-même ou sur un serveur dédié, éventuellement virtualisé (*appliance*)
 - **cloud**: en tant que service cloud (*Cloudflare WAF, AWS WAF...*)

Exemple concret de contre-mesure

Validation PHP

Code vulnérable

```
if (isset($_POST['Submit'])) {  
    $target = $_REQUEST['ip'];  
    $substitutions = array(  
        '&' => '',  
        ';' => '',  
        '|' => '',  
        '-' => '',  
        '$' => '',  
        '(' => '',  
        ')' => '',  
        '`' => '',  
        '||' => ''  
    );  
    $target = str_replace(array_keys($substitutions), $substitutions, $target);  
    $res = shell_exec('ping -c 3 ' . $target);  
    echo "<pre>{$res}</pre>";  
}
```

Revue de code

- Le code présenté implémente une liste plus complète de substitutions (liste noire) que l'exemple montré précédemment
- Néanmoins, ce code introduit un bug potentiellement difficile à détecter:
 - la substitution | est suivie d'un espace
 - or, dans une commande shell, | est un méta-caractère qui n'a pas besoin d'être suivi d'un espace pour être interprété
 - de plus, les éventuels tests cybers implémentés pourraient utiliser des espaces pour séparer les méta-caractères, et ainsi ne pas détecter le problème

- De plus, la liste noire est moins efficace que la liste blanche
 - il est difficile de penser à tous les méta-caractères possibles
 - cette liste pourrait évoluer avec le temps (mise à jour du système introduisant des modifications du shell utilisé)
 - il est difficile de s'assurer que toutes les substitutions sont toujours pertinentes
- Il faut toujours implémenter validation et assainissement **en fonction du contexte**
 - par exemple, ici, il est relativement aisé de valider que l'entrée est une adresse IP

Validation - Version 1

```
if (isset($_POST['Submit'])) {
    $target = $_REQUEST['ip'];
    // Assainissement : élimination des caractères d'échappement
    $target = stripslashes($target);
    // Découpage de l'entrée suivant les '.'
    $octets = explode('.', $target);
    // Validation : 4 parties attendues
    if (count($octets) != 4) {
        echo "<pre>ERREUR : Adresse IP non valide</pre>";
        return;
    }
    // Validation : chaque octet doit être un nombre entre 0 et 255
    foreach ($octets as $octet) {
        if (!is_numeric($octet) || $octet < 0 || $octet > 255) {
            echo "<pre>ERREUR : Adresse IP non valide</pre>";
            return;
        }
    }
}
```

Version 2 utilisant des REGEX

```
if (isset($_POST['Submit'])) {  
    $target = $_REQUEST['ip'];  
    // Utilisation de REGEX pour valider l'entrée  
    if (preg_match('/^(\d{1,3}\.){3}\d{1,3}$/', $target)) {  
        $res = shell_exec('ping -c 3 ' . $target);  
        echo "<pre>{$res}</pre>";  
    } else {  
        echo "<pre>ERREUR : Adresse IP non valide</pre>";  
    }  
}
```

Version 3 utilisant un filtre pré-implémenté

```
if (isset($_POST['Submit'])) {  
    $target = $_REQUEST['ip'];  
    if (filter_var($target, FILTER_VALIDATE_IP)) {  
        $res = shell_exec('ping -c 3 ' . $target);  
        echo "<pre>{$res}</pre>";  
    } else {  
        echo "<pre>ERREUR : Adresse IP non valide</pre>";  
    }  
}
```


