## CWE, CVE, CVSS

**CWE (Common Weakness Enumeration)**
- Catalogue de faiblesses logicielles
- Taxonomie des vulnérabilités

**CVE (Common Vulnerabilities and Exposures)**
- Base de données de vulnérabilités connues
- Documentées et référencées

**CVSS (Common Vulnerability Scoring System)**
- Notation de gravité des vulnérabilités
- Score basé sur plusieurs dimensions

**Dimensions CVSS**
- **Vecteur d'Attaque (AV)** : Network, Adjacent, Local, Physical
- **Complexité (AC)** : Low, High
- **Privilèges Requis (PR)** : None, Low, High
- **Interaction Utilisateur (UI)** : None, Passive, Active
- **Impact CIA** : Confidentiality, Integrity, Availability (None/Low/High)

## Injections

**Principe général** : L'attaquant insère du code malveillant dans une entrée utilisateur que l'application traite comme du code légitime.

**OS Command Injection (CWE-78)**
- Commande système avec données utilisateur non validées
- Exemple : `dig heig-vd.ch; rm -rf /`
- Impact : Exécution arbitraire de commandes

**SQL Injection (CWE-89)**
- Insertion de code SQL malveillant dans requêtes
- Impact : Accès/modification/suppression de données

**Code Injection (CWE-94)**
- Injection générique dans n'importe quel langage
- Impact : Exécution de code arbitraire

**Défenses**
- Validation stricte des entrées
- APIs sécurisées (sans shell)
- Requêtes préparées (SQL)
- Échappement adapté au contexte

## Vulnérabilités générales

**Null Pointer Dereference (CWE-476)**
- Accès à pointeur nul/non initialisé
- Impact : Crash (DoS), parfois RCE en kernel
- Défense : Gestion explicite (Option, Maybe), smart pointers

**Unsafe Deserialization (CWE-502)**
- Désérialiser données non fiables sans validation
- Impact : RCE via constructeurs malveillants
- Défense : Ne pas désérialiser sources non fiables, signer cryptographiquement, formats sûrs (JSON)

**Integer Overflow (CWE-190)**
- Opération dépasse capacité du type
- Impact : Bypass vérifications, buffer overflow
- Défense : Valider entrées, arithmétique vérifiée (checked_add, addExact)

**Hardcoded Credentials (CWE-798)**
- Secrets stockés en dur dans le code
- Impact : Accès compromis si code accessible
- Défense : Variables d'environnement, vaults

**Incorrect Authorization (CWE-863)**
- Contrôle d'accès basé sur données client modifiables
- Impact : Escalade de privilèges
- Défense : Vérifier autorisation côté serveur avec session sécurisée

**Path Traversal (CWE-22)**
- Chemins fichier construits avec données utilisateur
- Exemple : `../../tmp/uploads/ws.php`
- Impact : Accès fichiers hors répertoire autorisé
- Défense : Valider chemin reste dans répertoire, whitelist, canonicaliser

## Attaques Web

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

**Principe** : Injection de JavaScript malveillant dans le contexte d'un site victime. Abuse de la confiance du client envers le service.

**Reflected XSS**
- Lien piégé, service reflète données sans échappement
- Exemple : `?q=<script src="https://evil.com/hook.js"></script>`

**Stored XSS**
- Code injecté en base de données
- Chaque utilisateur consultant les données exécute le code

**DOM-Based XSS**
- Attaque côté client uniquement
- JavaScript manipule DOM avec données non échappées

**Défenses XSS**
- Échappement adapté au contexte (HTML, JS, URL, CSS)
- Validation d'entrée
- Content Security Policy (CSP)
- Web Application Firewall (WAF)

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

**Principe** : Tromper utilisateur authentifié pour effectuer action en son nom. Abuse de la confiance du service envers le client.

```
<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>
```

**Défenses CSRF**
- Protection XSS (prérequis)
- Tokens CSRF (aléatoire, vérifié côté serveur)
- Vérification Origin/Referer
- Cookies : SameSite=Strict, Secure
- Sémantique HTTP : GET sans effets de bord

### Unrestricted File Upload (CWE-434)

**Problème** : Upload sans restrictions d'extension

**Impact** : WebShell, RCE via `malicious.php` exécuté

**Défenses**
- Valider extension et type MIME
- Séparer code et contenu (domaine différent)
- Renommer fichiers
- Répertoire sans exécution

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

**Principe** : Forcer serveur à requêter service non autorisé

```
$url = $_GET['filename'];
$image = fopen($url,
'rb');  // Attaquant: ?
filename=https://backend.
internal/secrets
```

**Variantes dangereuses**
- Protocoles : `gopher://`, `dict://` pour commandes brutes
- Cross-protocol : `gopher://redis:6379/_FLUSHALL`
- Métadonnées EC2 : `169.254.169.254`

**Défenses SSRF**
- Whitelist d'URLs autorisées, restreindre schémas
- Filtrage réseau, proxy filtrant
- Architecture Zero Trust

## Concurrence

**Race Condition (CWE-362)**
- Exploiter fenêtre de temps entre opérations non atomiques
- Exemple : modifier fichier entre création et `chmod`
- Défense : Opérations atomiques, file locks

**Time-of-Check to Time-of-Use (TOCTTOU) (CWE-367)**
- Fichier/état change entre vérification et utilisation
- Exemple : `if (file_exists()) ... exec()` avec symlink créé entre les deux
- Défense : Opérations atomiques, éviter symlinks

## Vulnérabilités Mémoire

**Out-of-Bounds Write (CWE-787) - Buffer Overflow**
- Écrire au-delà des limites d'un buffer
- Impact : Écrasement return address → RCE
- Défenses : Stack non exécutable (NX), ASLR, fonctions sûres (strncpy)

**Out-of-Bounds Read (CWE-125) - Information Leak**
- Lire au-delà des limites
- Impact : Fuite de données sensibles (clés, tokens)
- Exemple : **Heartbleed** (CVE-2014-0160) - fuite clés privées OpenSSL
- Défense : Vérifier tailles, bounds checking

**Use-after-free (CWE-416)**
- Accéder à mémoire après free()
- Exploitation : Allouer même zone avec contenu contrôlé
- Défenses : Garbage collection, smart pointers, mettre à NULL après free

**Use of Format String (CWE-134)**
- Données utilisateur comme format string
- Exemple : `fprintf(log, command)` où `command = "%x %x %x"`
- Exploitation : Lecture stack avec `%123$x`, écriture avec `%n`
- Défense : **JAMAIS** user input comme format → `printf("%s", user_input)`

## Validation des Entrées

**Erreurs courantes liées aux entrées**
- Buffer overflow (taille)
- XSS (contenu + codage sortie)
- File upload of dangerous type
- Path traversal (chemins)
- SQL Injection (métacaractères)

### Fondamentaux

**Cohérence de la validation**
- Validation côté client ET serveur doit être alignée
- Factoriser logique pour éviter divergences
- Utiliser bibliothèques partagées

**Quand valider ?**
- **Dès que possible**
- Risques validation tardive : contenu dangereux en logs/stockage, side channels

**Nettoyer les entrées : DANGEREUX**
- Nettoyer correctement est difficile/impossible
- Chemins peu testés
- **Préférer rejeter** plutôt que nettoyer
- Tests cruciaux avec cas négatifs (entrées invalides) pour vérifier rejet approprié.

**Niveaux de validation**
1. **Syntaxique** : chaîne appartient à un langage
2. **Sémantique** : sens cohérent dans contexte
3. **Pragmatique** : véracité d'une proposition

**Exemple adresse mail**
- Syntaxique : `nom@domaine.tld` valide (utiliser libs)
- Sémantique : domaine existe, MX existe
- Pragmatique : utilisateur peut recevoir et communiquer secret

### Validation Pragmatique

**Principe** : Mécanismes externes (e-mail confirmation, SMS, services tiers)

**Secrets générés**
- Suffisamment longs (résistance force brute)
- Suffisamment aléatoires (éviter prédictions)
- Usage unique (éviter rejeu)
- Limités dans le temps (réduire fenêtre attaque)

### Validation Syntaxique

**Allow vs Deny Lists**
- **Allow lists** : spécifient entrées acceptables (recommandé)
- **Deny lists** : énumèrent entrées interdites (incomplet)

**Mesures générales**
- S'appuyer sur les types
- Canonicaliser valeurs avant filtrer
- Toujours vérifier tailles et intervalles
- Utiliser bibliothèques : Commons Validator (Java), Pydantic (Python), validator.js (JS)

**Expressions régulières**

Exemple Rust :

```
use regex::Regex;
let re =
Regex::new(r"^\d{2}-\d{2}-
\d{4}$")
  .expect("re compile
error");
assert!
(re.is_match("01-01-2024"));

// Précompilation
use std::sync::LazyLock;
static MY_REGEX:
LazyLock<Regex> =
LazyLock::new(|| {
  Regex::new(r"^\d{2}-
\d{2}-\d{4}$")
    .expect("re compile
error")
});
```

### Encodage des Sorties

**Principe** : Entrée validée doit être encodée selon contexte d'utilisation (HTML, SQL, etc.)

**Adressage indirect**
- Identifiants opaques pour communication externe
- Réassociation côté serveur
- Principe REST : URLs comme identifiants opaques

**Identifiants uniques**

Choisir 2 sur 3 :
- **Globalement uniques** + **Sécurisés** = **Clés publiques**
- **Globalement uniques** + **Conviviaux** = **Noms de domaine**
- **Sécurisés** + **Conviviaux** = **Keyring**