

# Sécurité logiciel

# Taxonomie de failles de logiciel

## 1. Failles de contrôles d'accès & authentification

- Description : défauts permettant à un usager non autorisé d'accéder à des ressources (p. ex. escalade de priviléges, contournement d'authentification).
- Exemples : Broken Access Control, défauts d'authentification, sessions non protégées.

## 2. Défauts d'entrée / injections

- Description : absence ou insuffisance de validation/filtrage des entrées, entraînant exécution de code, requêtes ou commandes malveillantes.
- Exemples : SQL Injection, Command Injection, LDAP/OS/XML injection, Cross-Site Scripting (XSS).

## 3. Sécurité mémoire et corruption (memory safety)

- Description : erreurs de gestion mémoire qui mènent à débordements, use-after-free, etc. Exploitable pour exécution de code arbitraire.
- Exemples : buffer overflow, heap overflow, use-after-free.

## 4. Échecs cryptographiques / gestion des secrets

- Description : usages incorrects des primitives cryptographiques, clés ou certificats exposés, algorithmes obsolètes.
- Exemples : chiffrement maison, stockage en clair de secrets, validation TLS incorrecte.

## 5. Conception inadaptée & défauts architecturaux

- Description : problèmes causés par une conception inadaptée (manque de défense en profondeur, exigences de sécurité non prises en compte).
- Exemples : absence de segmentation, logique métier non protégée.

# Taxonomie de failles de logiciel

## 6. Gestion des dépendances et composants tiers

- Description : vulnérabilités introduites via bibliothèques externes, versions obsolètes, ou packages compromis.
- Exemples : vulnérabilités dans des frameworks, packages NPM/PyPI non mis à jour.

## 7. Désérialisation non sécurisée & exécution d'objet

- Description : désérialisation d'objets non fiables conduisant à exécution de code ou élévation de privilèges.
- Exemples : Java/insecure deserialization, PHP unserialize() d'entrée non fiable.

## 8. Mauvaise gestion des erreurs et fuite d'informations

- Description : messages d'erreur trop verbeux ou logs exposant des données sensibles (stack traces, chemins, clés).
- Mesure : masquer détails en production, journaliser de façon sécurisée.

## 9. Conditions de concurrence / race conditions

- Description : erreurs liées à l'accès simultané aux ressources (TOCTOU, race sur fichiers/transactions).
- Impact : corruption de données, contournement de vérifications.

# Taxonomie de failles de logiciel

## 10. Disponibilité / Denial-of-Service

- Description : failles permettant d'épuiser ressources (CPU, mémoire, connexions) ou de provoquer plantage.
- Exemples : requêtes coûteuses, amplification, loop infini sur input mal formé.

## 11. Erreurs de configuration & exposition (misconfiguration)

- Description : configurations par défaut, ports ouverts, informations de débogage exposées, permissions trop larges.
- Pratique : hardening, scans de configuration, principe du moindre privilège.

## 12. Failles de logique métier (business logic)

- Description : erreurs dans la façon dont l'application implémente les règles métier — pas forcément détectables par scanners automatisés.
- Exemples : contournement d'étapes de paiement, modifications de prix via paramètres.

# Taxonomie de failles de logiciel

| Catégorie                                   | Exemples   |
|---|--|
| Contrôles d'accès & authentification        | Escalade de priviléges, contournement d'authentification, sessions non protégées |
| Injections & défauts d'entrée               | SQL Injection, Command Injection, XSS, LDAP/OS injection                         |
| Sécurité mémoire                            | Buffer overflow, heap overflow, use-after-free                                   |
| Cryptographie & gestion des secrets         | Stockage en clair, clés exposées, algorithmes faibles, validation TLS incorrecte |
| Insecure design / défauts architecturaux    | Absence de défense en profondeur, logique métier non protégée                    |
| Dépendances et composants tiers             | Vulnérabilités dans frameworks, bibliothèques obsolètes ou compromises           |
| Déserialisation non sécurisée               | Unserialize() non filtré, exécution d'objets non fiables                         |
| Gestion des erreurs et fuite d'informations | Stack trace exposée, logs sensibles  |
| Conditions de concurrence                   | Race conditions, TOCTOU sur fichiers/transactions                                |
| Disponibilité (Denial-of-Service)           | Exploitation de ressources, requêtes coûteuses, amplification                    |
| Mauvaise configuration (misconfiguration)   | Ports ouverts, configs par défaut, débogage exposé                               |
| Failles de logique métier                   | Contournement d'étapes de paiement, modification de prix                         |

# Checklist pratique de revue de code sécurité (pas pour le final)

| Catégorie                                       | Contrôle à vérifier  |
|---|--|
| <b>Contrôles d'accès &amp; authentification</b> | Toutes les ressources sont protégées par des contrôles d'accès côté serveur (pas seulement côté client). |
| <b>Contrôles d'accès &amp; authentification</b> | Les règles d'autorisation sont systématiquement appliquées par objet/ressource (ABAC/RBAC).              |
| <b>Contrôles d'accès &amp; authentification</b> | Les actions sensibles exigent une ré-authentification ou MFA.  |
| <b>Contrôles d'accès &amp; authentification</b> | Les sessions utilisent des cookies sécurisés (Secure, HttpOnly, SameSite) et expirent correctement.      |
| <b>Contrôles d'accès &amp; authentification</b> | Pas d'ID incrémentaux devinables (userId, orderId) exposés directement.                                  |
| <b>Injections &amp; entrées</b>                 | Toutes les requêtes SQL utilisent des requêtes paramétrées/ORM (pas de concaténation).                   |
| <b>Injections &amp; entrées</b>                 | Échappement/validation côté serveur pour commandes shell, LDAP, NoSQL, XPath.                            |
| <b>Injections &amp; entrées</b>                 | Sortie HTML/JS correctement encodée (XSS contextuel, CSP activée).                                       |
| <b>Injections &amp; entrées</b>                 | Les webhooks et inputs externes sont validés par schéma (JSON/XML).                                      |
| <b>Injections &amp; entrées</b>                 | Désactivation des interprétations dangereuses (e.g., eval, deserializers non sûrs).                      |
| <b>Sécurité mémoire</b>                         | Langages memory-safe privilégiés ou protections activées (ASLR, DEP, stack canaries).                    |
| <b>Sécurité mémoire</b>                         | Fuzzing et sanitizers (ASan/UBSan) sur composants natifs.  |
| <b>Cryptographie &amp; secrets</b>              | Pas de crypto maison ; bibliothèques standard et primitives à l'état de l'art.                           |
| <b>Cryptographie &amp; secrets</b>              | Stockage chiffré des secrets (vault), rotation et séparation des rôles.                                  |
| <b>Cryptographie &amp; secrets</b>              | TLS v1.2+ avec vérification stricte des certificats (pinning si pertinent).                              |
| <b>Cryptographie &amp; secrets</b>              | Aucune clé/secret en code/CI ; variables d'environnement/vault uniquement.                               |
| <b>Conception (Insecure Design)</b>             | Menaces modélisées (STRIDE, LINDDUN) et contre-mesures documentées.                                      |
| <b>Conception (Insecure Design)</b>             | Défense en profondeur (limitation d'impact si un contrôle échoue).                                       |
| <b>Dépendances &amp; composants</b>             | SBOM générée (CycloneDX/SPDX) et scannée automatiquement.  |
| <b>Dépendances &amp; composants</b>             | Mises à jour régulières ; politiques de versions et dépendances épinglees.                               |
| <b>Dépendances &amp; composants</b>             | Vérification de signature/hashe des artefacts (supply chain).  |
| <b>Désérialisation</b>                          | Formats sûrs (JSON) et listes d'objets autorisés si désérialisation obligatoire.                         |
| <b>Désérialisation</b>                          | Désérialisation jamais sur des données non fiables sans validation stricte.                              |

# Checklist pratique de revue de code sécurité (pas pour le final)

|                                       |  |
|---------------------------------------|--|
| <b>Erreurs &amp; divulgations</b>     | Messages d'erreur neutres (pas de stack trace/details en prod).                    |
| <b>Erreurs &amp; divulgations</b>     | Logs centralisés, protégés, sans données sensibles en clair.                       |
| <b>Conditions de concurrence</b>      | Verrous/transactions atomiques pour opérations critiques (TOCTOU évité).           |
| <b>Conditions de concurrence</b>      | Files/idempotence pour endpoints non idempotents sensibles.                        |
| <b>Disponibilité (DoS)</b>            | Limitation de débit, quotas et timeouts côté serveur.                              |
| <b>Disponibilité (DoS)</b>            | Détection d'anomalies et mécanismes de backpressure/circuit breaker.               |
| <b>Configuration &amp; exposition</b> | Environnements séparés (dev/test/prod) avec secrets/accès distincts.               |
| <b>Configuration &amp; exposition</b> | Headers de sécurité (HSTS, X-Frame-Options, X-Content-Type-Options).               |
| <b>Configuration &amp; exposition</b> | Ports/services inutiles désactivés ; pas de pages d'admin exposées.                |
| <b>Logique métier</b>                 | Contrôles d'intégrité sur flux de paiement/ordre (anti-manipulation).              |
| <b>Logique métier</b>                 | Vérifications de préconditions (états, soldes, quotas) côté serveur.               |
| <b>Logique métier</b>                 | Protection contre l'automatisation abusive (anti-bot, preuve de travail si utile). |
| <b>Processus SDLC sécurisé</b>        | Revue de code systématique avec checklist sécurité intégrée au PR.                 |
| <b>Processus SDLC sécurisé</b>        | SAST/DAST/IAST/Fuzz intégrés à la CI, avec seuils de qualité.                      |
| <b>Processus SDLC sécurisé</b>        | Gestion des vulnérabilités (triage SLA, patch, retest, RCA).                       |

# Exemple d'escalade de privilèges

- Application web de gestion de comptes : chaque utilisateur a un rôle (user ou admin). L'UI montre des boutons d'administration uniquement aux admins, mais l'API REST côté serveur accepte une propriété rôle envoyée par le client lors de la mise à jour du profil.
- Conséquence : un utilisateur malveillant peut modifier sa requête pour demander role=admin. Si le serveur fait confiance à ce champ côté client, l'utilisateur obtiendra des droits d'admin (escalade de privilèges).
- Pourquoi la faille existe
  - Confiance côté client : l'application se repose sur le front-end pour cacher fonctions sensibles au lieu d'imposer des contrôles serveur.
  - Absence de vérification côté serveur : le backend n'effectue pas de contrôle d'autorisation basé sur l'identité réelle (ex : session/auth token) et applique directement le champ fourni.
  - Mauvaise séparation des responsabilités : logique d'autorisation pas centralisée / pas appliquée systématiquement.

# Exemple d'escalade de privilèges

```
// Mauvaise pratique — NE PAS reproduire en prod
app.post('/api/users/:id/update', authenticate, (req, res) => {
  const id = req.params.id;
  const updates = req.body; // contient possiblement { name, email, role }
  // Mise à jour directe — on applique ce que l'utilisateur envoie
  db.users.update(id, updates);
  res.send({ ok: true });
});
```

# Exemple SQL injection

L'injection SQL est une classe de vulnérabilité qui se produit quand une application transmet des données non-fiables (par exemple des valeurs saisies par l'utilisateur) au moteur SQL d'une façon qui permet à ces données de modifier la structure logique de la requête. Autrement dit : des données deviennent du « code » côté base de données.

Comment ça fonctionne, en principe

- L'application construit une requête SQL à exécuter sur la base de données.
- Si la construction de la requête est faite en concaténant directement des chaînes fournies par l'utilisateur, la structure finale de la requête dépend du texte entré par l'utilisateur.
- Le moteur SQL reçoit la chaîne résultante et l'analyse : s'il contient des éléments qui changent la logique (conditions, opérateurs, sous-requêtes), alors le comportement de la requête change — potentiellement en dehors de l'intention du développeur.
- Selon la façon dont la base est configurée et selon les informations retournées, un attaquant peut tirer parti de ce comportement pour, par exemple, obtenir des données non autorisées, manipuler des résultats, ou modifier des enregistrements.

# Exemple SQL injection

Il y a deux types de SQL Injection pour contourner le Pare-feu de l'application WEB

- **SQL Injection into a String/Char parameter**

Example: `SELECT * from table where example = 'Example'`

- **SQL Injection into a Numeric parameter**

Example: `SELECT * from table where id = 123`

[https://owasp.org/www-community/attacks/SQL\\_Injection\\_Bypassing\\_WAF](https://owasp.org/www-community/attacks/SQL_Injection_Bypassing_WAF)

## Example SQL Injection

| ID | NAME     | AGE |
|----|----------|-----|
| 1  | Bob      | 6   |
| 2  | L'eponge | 9   |
| 3  | Patrick  | 40  |
| 4  | Star     | 50  |

# SQL INSERT

```
INSERT INTO users (ID, NAME, AGE) VALUES
(1, 'Bob', 6),
(2, 'Leponge', 9),
(3, 'Patrick', 40),
(4, 'Star', 50);
```

## SQL Injection

| ID | NAME                                 | AGE |
|----|--------------------------------------|-----|
| 1  | Bob                                  | 6   |
| 2  | L'eponge                             | 9   |
| 3  | "Robert'); DROP TABLE<br>users; -- " | 40  |
| 4  | Star                                 | 50  |

# SQL Injection

```
INSERT INTO users (id, name, age) VALUES (3, 'Robert'); DROP TABLE users;--', 40);
```

# Injection SQL Basé sur $1 = 1$ est toujours vrai

L'objectif initial du code était de créer une instruction SQL pour sélectionner un utilisateur avec un nom d'utilisateur donné.

Si il n'y a rien pour empêcher a un utilisateur de faire entrer de fausses données celui-ci profitera pour mettre des données intelligentes comme:

## Server Result:

```
SELECT * FROM member WHERE username = " OR 1=1 OR '=1'  
AND  
password = 'grtgerhye5y5eytet'
```

User Authentication

Username:

Password:

# Injection SQL Basé sur --

## Server Result:

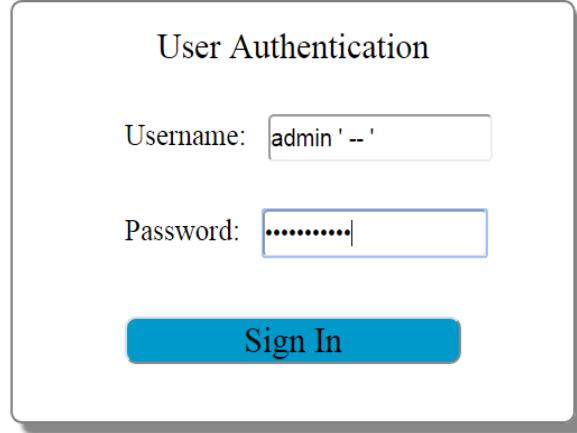
```
SELECT * FROM member WHERE username = " admin'--' AND  
password = 'grtgerhye5ytet'
```

User Authentication

Username:

Password:

**Sign In**



Si la table des utilisateurs contient les noms et mots de passe...

```
SELECT username, password FROM member WHERE username = 105 or 1=1
```

Un hacker intelligent pourrait obtenir l'accès à tous les noms d'utilisateur et mots de passe dans une base de données en insérant simplement 105 or 1 = 1 dans la zone de saisie.

# Injection SQL Basé sur "" = "" est toujours vrai

Voici une construction commune, utilisée pour vérifier la connexion de l'utilisateur vers un site Web:

## Server Code:

```
username = getRequestString("username");
password = getRequestString("password");

sql = "SELECT * FROM member WHERE username ='" + username + "' AND password ='" + password + "'"
```

Un hacker intelligent pourrait obtenir l'accès aux noms d'utilisateur et mots de passe dans une base de données en insérant simplement " or" "=" dans la boîte de texte Nom ou mot de passe de l'utilisateur.

Le code sur le serveur va créer une instruction SQL valide comme ceci:

## Result:

```
SELECT * FROM member WHERE username ="" or ""="" AND password ="" or ""=""
```

Comme le résultat SQL est valide.l'application nous retournera tous les noms d'utilisateurs

# Injection SQL fondé sur les batched SQL Statements

La plupart des bases de données supportent des instruction batched SQL statement, séparées par virgule.

- **Example**

```
SELECT * FROM member; DROP TABLE member;
```

Le SQL ci-dessus renverra toutes les lignes dans la table des utilisateurs, puis supprimera la table appelée member.

# Exemple Buffer overflow

Un buffer overflow se produit quand un programme écrit plus de données qu'un espace mémoire alloué (tampon), ce qui écrase les octets voisins en mémoire (variables locales, pointeurs, métadonnées du heap, etc.).

Conséquences possibles : plantage (crash), divulgation d'informations, corruption de données, et — dans certains cas mal protégés — exécution de code arbitraire.

- Variantes courantes :
  - Stack overflow : tampon alloué sur la pile (stack). Écrase variables locales et la zone de contrôle d'exécution (ex. pointeurs de retour).
  - Heap overflow : dépassement d'un tampon sur le heap — peut corrompre structures d'allocation.
  - Off-by-one : écriture d'un octet de trop (courante) ; provoque souvent des bugs subtils.
  - Integer overflow menant à allocation trop petite (taille calculée incorrectement).

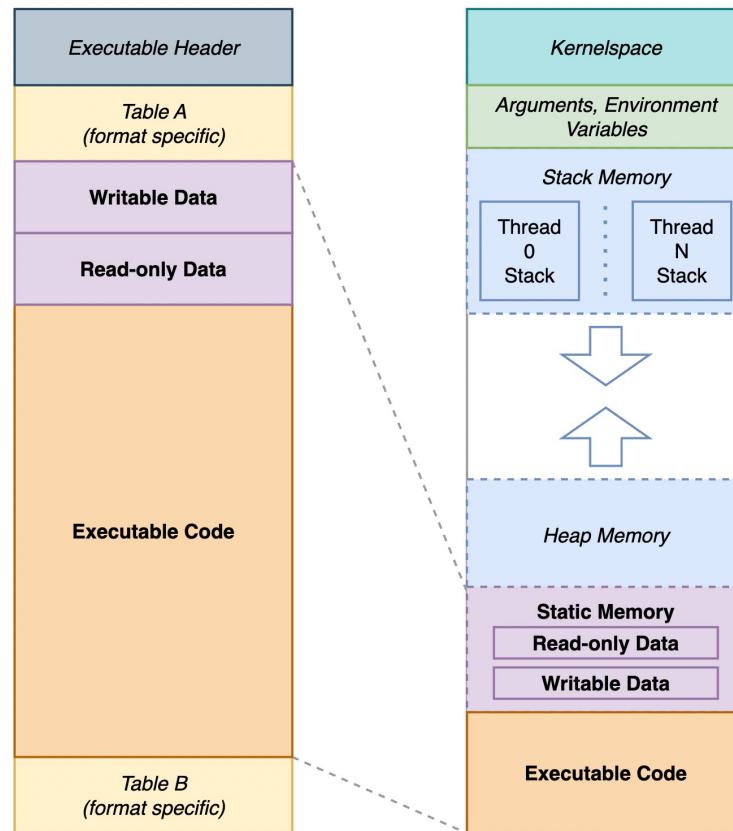
# Exemple Buffer overflow

Comment fonctionne la mémoire dans un processus ?

- La mémoire d'un processus peut être divisée en cinq segments principaux :
- Le code : Ce segment contient les instructions compilées du programme, prêtes à être exécutées en langage machine.
- Le segment de données : Il regroupe les variables globales initialisées. Il comporte deux zones : une zone en lecture seule, pour les constantes; et une zone en lecture-écriture, pour les variables modifiables.
- Segment BSS (Block Starting Symbol) : Il contient les variables globales et statiques initialisées à zéro ou sans valeur initiale définie dans le code.
- La pile (stack) : Elle est utilisée pour gérer les appels de fonction. Et elle contient, les paramètres des fonctions, leurs variables locales, ainsi que l'adresse de retour après l'exécution. Par ailleurs, la pile suit une organisation « Dernier Entré, Premier Sorti » (Last In, First Out, ou LIFO), ce qui signifie que les dernières données ajoutées sont les premières à être retirées.
- Le tas (heap) : Il s'agit de la zone dédiée à l'allocation dynamique de mémoire, réalisée par le programmeur. Par exemple, avec `malloc()` en C ou `new()` en C++. La mémoire allouée persiste jusqu'à sa libération explicite.

Ce modèle de segmentation est essentiel pour comprendre les failles liées aux dépassemens de tampon. Le schéma ci-dessous représente l'organisation de la mémoire pour un programme une fois compilé.

# Exemple Buffer overflow



Mapping of on-disk, executable contents to in-memory process space.

Source: [https://highassurance.rs/chp4/sw\\_stack\\_1.html](https://highassurance.rs/chp4/sw_stack_1.html)

# Exemple Buffer overflow

## Types d'attaques par dépassement de tampon

Il existe plusieurs types d'attaques par dépassement de tampon que les attaquants utilisent pour exploiter les systèmes des organisations. Les plus courants sont les suivants :

- **dépassements de tampon** : il s'agit de la forme la plus courante d'attaque de dépassement de tampon. L'approche basée sur la pile se produit lorsqu'un attaquant envoie des données contenant un code malveillant à une application, qui stocke les données dans un tampon de pile. Cela écrase les données de la pile, y compris son pointeur de retour, qui permet de contrôler les transferts vers l'attaquant.
- **dépassements de tampon basés sur le tas** : une attaque basée sur le tas est plus difficile à mener que l'approche basée sur la pile. Elle implique que l'attaque sature l'espace mémoire d'un programme au-delà de la mémoire utilisée pour les opérations d'exécution actuelles.
- **attaque de chaîne de format** : une exploitation de chaîne de format se produit lorsqu'une application traite les données d'entrée comme une commande ou ne valide pas efficacement les données d'entrée. Cela permet à l'attaquant d'exécuter du code, de lire des données dans la pile ou de provoquer des défauts de segmentation dans l'application. Cela pourrait déclencher de nouvelles actions qui menacent la sécurité et la stabilité du système.

# Exemple Buffer overflow

## Quels langages de programmation sont les plus vulnérables ?

Presque toutes les applications, tous les serveurs Web et environnements d'applications Web sont vulnérables aux dépassesments de tampon. Les environnements écrits dans des langages interprétés, tels que Java et Python, sont immunisés contre les attaques, à l'exception des débordements dans leur langage interprété.

**Le ping de la mort** est une forme d'attaque par déni de service (DoS) qui se produit lorsqu'un attaquant plante, déstabilise ou gèle des ordinateurs ou des services en les ciblant avec des paquets de données volumineux. Cette forme d'attaque DoS cible et exploite généralement les faiblesses héritées que les organisations peuvent avoir corrigées. Les systèmes non corrigés sont également exposés aux inondations de ping, qui ciblent les systèmes en les surchargeant de messages ping (ICMP).

- Changer le ping en commande ping de mort

Les attaquants utilisent des commandes ping pour développer une commande ping de mort. Ils peuvent écrire une boucle simple qui leur permet d'exécuter la commande ping avec des tailles de paquets qui dépassent le niveau maximal de 65 535 octets lorsque la machine cible tente de remettre les fragments en place.

- Exploiter la vulnérabilité

L'envoi de paquets supérieurs à 65 535 octets enfreint les règles de l'IP. Pour éviter cela, les attaquants envoient des paquets en fragments que leur système cible tente ensuite de fragmenter. Dans ce cas, le paquet surdimensionné entraîne un dépassement de mémoire.

# Stack overflow – Ping de la mort

Le Ping de la mort (Ping of Death) est une attaque réseau historique qui exploitait une faille dans la gestion des paquets ICMP (Internet Control Message Protocol) dans **les anciens systèmes d'exploitation**.

Le protocole ICMP, utilisé notamment par la commande ping, envoie des paquets pour tester la connectivité entre deux machines.

Normalement, un paquet ICMP ne doit pas dépasser 65 535 octets, taille maximale définie par la norme IP.

- L'attaque du Ping de la mort consistait à :
  - Envoyer un paquet ICMP fragmenté, mais dont la taille totale, une fois réassemblée, dépassait la limite autorisée.
  - Lors du réassemblage du paquet par la machine cible, un dépassement de mémoire (buffer overflow) se produisait.
  - Cela entraînait souvent un plantage, un redémarrage, voire un gel du système.

# Ping de la mort

