CWE, CVE, CVSS

CWE: Catalogue faiblesses logicielles, taxonomie vulnérabilités

CVE: Base données vulnérabilités connues

CVSS: Notation gravité vulnérabilités

Dimensions: AV (Network/Adjacent/Local/Physical), AC (Low/High), PR (None/Low/High), UI (None/Passive/Active), CIA (Confidentiality/Integrity/Availability)

Injections

Principe: Code malveillant dans entrée traité comme code légitime

OS Command (CWE-78): Données non validées dans commande système. Ex: `dig heig-vd.ch; rm -rf /` → exécution arbitraire

SQL (CWE-89): Code SQL malveillant → accès/modification données

Code (CWE-94): Injection tout langage → exécution arbitraire

Défenses: Validation stricte, APIs sécurisées, requêtes préparées, échappement

Vulnérabilités générales

Null Pointer (CWE-476): Accès pointeur nul → Crash/RCE. Défense: Gestion explicite, smart pointers

Unsafe Deserialization (CWE-502): Données non fiables → RCE. Défense: Signer, formats sûrs

Integer Overflow (CWE-190): Dépasse capacité → Bypass, overflow. Défense: checked_add

Hardcoded Credentials (CWE-798): Secrets en dur → accès compromis. Défense: Env vars, vaults

Incorrect Authorization (CWE-863): Données client modifiables → escalade. Défense: Vérif serveur

Path Traversal (CWE-22): `../../tmp/ws.php` → accès hors répertoire. Défense: Whitelist

Attaques Web

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

Principe: Injection JavaScript malveillant dans contexte site victime. Abuse confiance client envers service

Types
- **Reflected**: Lien piégé, service reflète données sans échappement
- **Stored**: Code injecté en BD, chaque utilisateur l'exécute
- **DOM-Based**: Attaque côté client, JS manipule DOM

Défenses: Échappement adapté contexte (HTML, JS, URL, CSS), validation entrée, Content Security Policy (CSP), WAF

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

Principe: Tromper utilisateur authentifié pour effectuer action en son nom. Abuse confiance service envers client

Défenses
- 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)

Impact: WebShell, RCE

Défenses: Valider extension/MIME, séparer code/contenu, renommer fichiers, répertoire sans exécution

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

Principe: Forcer serveur à requêter service non autorisé

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

Défenses: Whitelist URLs autorisées, restreindre schémas, filtrage réseau, Zero Trust

Concurrence

Race Condition (CWE-362): Exploiter fenêtre temps entre opérations non atomiques. Ex: modifier fichier entre création et `chmod`

TOCTTOU (CWE-367): Fichier/état change entre vérification et utilisation. Ex: `if (file_exists()) ... exec()` avec symlink créé entre

Défenses: Opérations atomiques, file locks, 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 tardifs: logs dangereux, side channels)

Nettoyer : DANGEREUX. Préférer rejeter. Tests avec cas négatifs cruciaux

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

Mécanismes externes: e-mail, SMS, services tiers

Secrets: Longs, aléatoires, usage unique, limités temps

Validation Syntaxique

Allow vs Deny: Allow lists recommandé (deny incomplet)

Mesures: Types, canonicaliser, vérifier tailles/intervalles, libs (Pydantic, validator.js), regex précompilées

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

Authentification

Mot de passe

Stockage sécurisé
- JAMAIS en clair, JAMAIS hashé simple (MD5/SHA-1)
- **Fonctions dédiées**: bcrypt, scrypt, Argon2 (recommandé OWASP)
- Salt unique par utilisateur (évite rainbow tables)
- Itérations élevées (ralentit brute-force)
- **Pepper**: Secret global supplémentaire (stocké séparément)

Politiques: 12+ car, diversité, HaveIBeenPwned API, éviter expirations fréquentes. **Passkeys** (WebAuthn) résiste phishing

Authentification multi-facteurs (MFA)

Principe: Combiner plusieurs facteurs indépendants

Facteurs d'authentification
1. **Connaissance**: Mot de passe, PIN, réponse secrète
2. **Possession**: Smartphone, token hardware, carte à puce
3. **Inhérence**: Empreinte digitale, reconnaissance faciale, voix

Avantages: Même si mot de passe compromis, accès bloqué sans 2e facteur

Implémentations: TOTP (Time-based OTP), SMS (moins sûr), authentificateurs hardware (YubiKey)

Biométrie

Avantages: Difficile dupliquer.
Inconvénients: Non révocable, faux positifs/négatifs, vie privée

Single Sign-On (SSO)

Principe: Une authentification pour plusieurs services

Protocoles: OAuth 2.0, OpenID Connect, SAML

Flux OAuth 2.0
- Authorization Code (serveur, recommandé)
- Implicit (obsolète, insécure)
- Client Credentials (machine-to-machine)

Tokens
- **Access token**: Accès ressources (courte durée)
- **Refresh token**: Renouveler access token (longue durée)
- **ID token** (OIDC): Informations utilisateur (JWT)

Avantages: Expérience utilisateur améliorée, gestion centralisée

Risques: Point de défaillance unique, compromission IdP → accès tous services

Gestion de sessions

Session IDs
- Aléatoires cryptographiquement (256+ bits)
- Transmis via cookies sécurisés (HttpOnly, Secure, SameSite)
- Régénérer après authentification (évite session fixation)
- Timeout inactivité + durée max absolue

Défenses
- **Session fixation**: Régénérer ID après login
- **Session hijacking**: HTTPS obligatoire, IP binding (avec précaution)
- Logout: Invalider session côté serveur

Rate Limiting

Principe: Limiter tentatives pour ralentir brute-force

Stratégies: Limiter par IP/compte, délais progressifs (exponential backoff), CAPTCHA après N échecs, account lockout temporaire

Autorisation

Principe: Déterminer si utilisateur authentifié peut effectuer action sur ressource

CWE-285: Improper Authorization - Actions critiques sans vérification permissions appropriées

Modèles d'accès

Capability-Based: Tokens non falsifiables (file descriptors, API keys). Moindre privilège naturel

ACL: Ressource → liste users/perms (Unix rwx). Gestion centralisée, difficile voir droits user

Role-Based Access Control (RBAC)

Principe: Permissions assignées à rôles, utilisateurs assignés à rôles

Structure
- Utilisateur → Rôles → Permissions → Ressources
- Exemple: Admin, Moderator, User
- Rôles hiérarchiques possibles (Admin hérite User)

Avantages
- Gestion simplifiée (ajouter/retirer rôles)
- Séparation des responsabilités
- Principe moindre privilège

Limitations: Rigide, pas contexte dynamique (heure, localisation)

Attribute-Based Access Control (ABAC)

Principe: Décisions basées sur attributs (utilisateur: département/clearance, ressource: classification, environnement: heure/IP)

Avantages: Flexible, contexte dynamique, granulaire **Inconvénients**: Complexité configuration, debug difficile

MAC

Principe: Système impose politique centralisée. Labels sécurité, no read up/write down (Bell-LaPadula). SELinux

DAC

Principe: Propriétaire définit accès (chmod, Google Drive). Flexible mais propagation non contrôlée

Bonnes pratiques

Principe moindre privilège: Accès minimal nécessaire tâche **Séparation responsabilités**: Aucun utilisateur pouvoir complet seul **Défense en profondeur**: Multiples couches vérification **Audit logs**: Logger décisions autorisation (qui, quoi, quand, résultat) **Fail securely**: En cas erreur, refuser accès par défaut **Vérification serveur**: JAMAIS fier données client (CWE-863)

Frameworks

Casbin: Lib autorisation (RBAC, ABAC, ACL), DSL PERM. **OAuth 2.0 Scopes**: Limiter accès API. **XACML**: Standard XML ABAC

CIA

Confidentialité

Protection accès non autorisé: Chiffrement symétrique (AES, ChaCha20), asymétrique (RSA, ECC), AEAD (AES-GCM, ChaCha20-Poly1305). Contrôle accès RBAC/ABAC, auth forte MFA/biométrie

Menaces: MITM, divulgation, side channels

Intégrité

Données non modifiées: Hachage (SHA-256, SHA-3), signatures (RSA, ECDSA, EdDSA - non-répudiation), MACs (HMAC, Poly1305 - plus rapide), CRC (erreurs accidentelles)

Menaces: Altération, replay attacks, corruption

Disponibilité

Accès ressources: Redondance (clusters, failover), scalabilité (load balancing, CDN), backups (RPO/RTO), protection (anti-DDoS, rate limiting, WAF)

Menaces: DoS/DDoS, pannes, désastres

Crypto vs Sécurité

Crypto ≠ Sécurité: Crypto = primitives. Sécurité = crypto + architecture + processus + humain. AEAD (AES-GCM) = Encrypt-then-MAC intégré. CSPRNG requis pour IVs/nonces/salts

TLS (Sécurité en Transit)

TLS: CIA pour TCP (DTLS pour UDP). SSL 2.0/3.0, TLS 1.0/1.1 cassés. TLS 1.2 compatible (config précise), TLS 1.3 recommandé (post-quantum)

Choix: TLS 1.3 si contrôle, TLS 1.2 si compatibilité

Auth X.509: Serveur seul/mutual/aucun. Interne: CA interne. Externe: LetsEncrypt

Test: testssl.sh, ssllabs.com/ssltest

TLS 1.3: ECDHE (forward secrecy), AES-GCM/ChaCha20-Poly1305, HSTS

Équilibre CIA

Trade-offs: Confidentialité vs perf (ChaCha20 rapide, AES-NI), intégrité vs overhead (AEAD efficace), disponibilité vs sécurité (auth stricte ralentit)

Conclusion: Libs pour crypto, focus gestion clés. Pas solution universelle

Logging et Monitoring

Principe: Enregistrer événements système pour détection incidents et analyse forensique

Niveaux de logs

Standards: DEBUG, INFO, WARNING, ERROR, CRITICAL **Production**: WARNING et supérieur (éviter pollution)

Événements de sécurité critiques

À logger systématiquement: Authentifications (succès/échecs répétés), changements autorisation, modifications config sécurité, accès ressources sensibles, erreurs validation entrées, anomalies réseau

Bonnes pratiques

Contenu logs: Timestamp (UTC), user/session ID, action/résultat, source (IP). **JAMAIS** secrets

Protection logs
- Contrôle accès strict (append-only)
- Intégrité: signatures, stockage externe
- Rétention appropriée, centralisation (SIEM)
- Rotation: Archivage, compression
- Format structuré: JSON préféré
- Correlation IDs: Tracer requête à travers services

Vulnérabilités logging

Log Injection: Entrées avec n → fausses entrées. **Log Forging**: Injection commandes. **Data Exposure**: PII/secrets. **Défense**: Échapper, JSON, pas de secrets

SIEM

Principe: Agrégation, corrélation logs. Détection patterns, alertes temps réel, corrélation événements, conformité

Outils: Splunk, ELK Stack, Wazuh

Outils de Sécurité

Vérification Dynamique

Principe: Analyse comportement programme lors exécution (détection bugs runtime)

Sanitizers

AddressSanitizer (ASan)
- Détecte: Buffer overflow, use-after-free, double-free
- Impact: 2x ralentissement, mémoire +3x
- Usage: `-fsanitize=address`

LeakSanitizer (LSan): Fuites mémoire (intégré ASan ou standalone) - `fsanitize=leak`

UndefinedBehaviorSanitizer (UBSan): Comportements indéfinis (integer overflow, division par 0) - `fsanitize=undefined`

MemorySanitizer (MSan): Lectures mémoire non initialisée - `fsanitize=memory`

ThreadSanitizer (TSan): Data races (incompatible autres sanitizers) - `fsanitize=thread`

Combinaison: ASan+LSan+UBSan ensemble, TSan seul. **Compilation**: Flags `-g -O1`

Valgrind

Memcheck: Erreurs mémoire (fuites, accès invalides). **Helgrind/DRD**: Data races. 10-50x plus lent. `valgrind --leak-check=full`

Code Coverage

Métriques: Line (lignes), Branch (conditionnelles), Path (chemins)

Outils: gcov/lcov (C/C++), coverage.py (Python), JaCoCo (Java), Istanbul (JS)

Analyse Statique

Principe: Examiner code source sans l'exécuter (détection vulnérabilités potentielles)

SAST (Static Application Security Testing)
- Analyse patterns dangereux (buffer overflow, SQL injection, XSS)
- Outils: SonarQube, Semgrep, Bandit (Python), ESLint (JS)
- Avantages: Tôt dans dev cycle
- Limites: Faux positifs, bugs runtime non détectés

Linters de sécurité: Clippy (Rust), RuboCop (Ruby), pylint (Python)

Dependency scanning: Détecter CVE dans dépendances (npm audit, OWASP Dependency-Check)

Fuzzing

- **Principe**: Tests automatisés entrées aléatoires/mutées
- **AFL++**: Coverage-guided, mutations intelligentes, détecte crashes/hangs
- **Bonnes pratiques**: Combiner ASan+AFL++, seeds qualité, longue durée, dictionnaires (JSON/XML)
- **Autres**: LibFuzzer (LLVM), Honggfuzz, OSS-Fuzz

DAST (Dynamic Application Security Testing)

- **Principe**: Tests application en exécution (boîte noire)
- **Outils**: OWASP ZAP, Burp Suite (proxies, scanners). Détectent XSS, SQLi, CSRF
- **Complémentarité**: SAST (dev) + DAST (pré-prod) + tests manuels