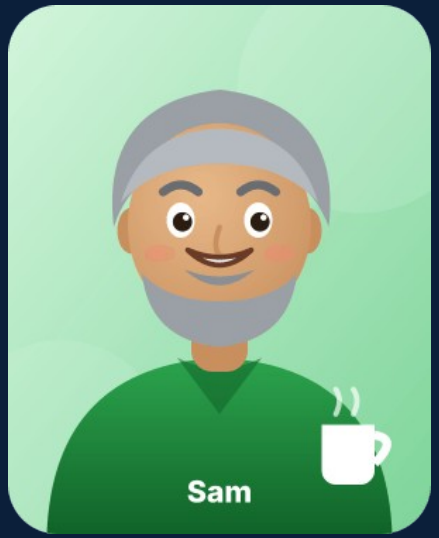
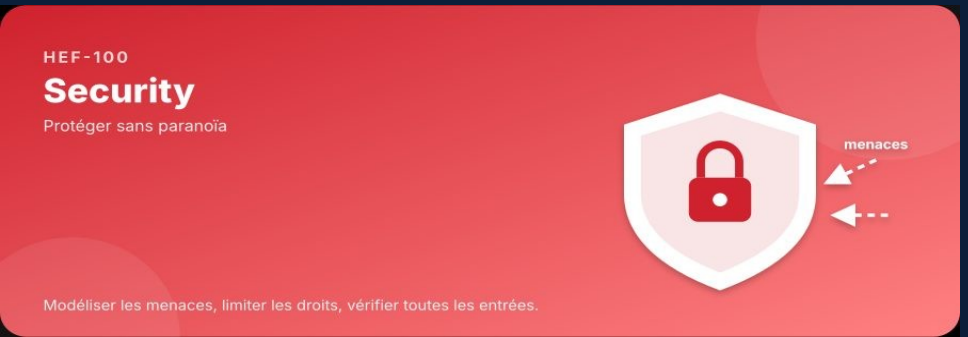
HEF-100 · MISSION 07 / 10

OWASP Top 10 et validation des entrées

Les failles les plus fréquentes sont toujours les mêmes. Les connaître, c'est déjà éviter la plupart.





Badge : Chasseur de failles (côté défense)

100 min · intermédiaire

L'histoire

L'histoire

HashBox reçoit son premier audit externe. Le rapport tient en une phrase par catégorie : « Injection SQL sur la recherche de cours. Traversée de répertoire sur le téléchargement de supports. Désérialisation non sûre dans l'import. Contrôle d'accès cassé sur les profils. »



> *« Quatre failles, quatre catégories du Top 10 de l'OWASP, dit Sam. Rien d'exotique. La bonne nouvelle : le Top 10 est aussi une liste de choses à vérifier avant de livrer. »*

Tu apprends à reconnaître les grandes familles de failles, à les reproduire sur du code jouet que tu écris, puis à les corriger avec des contrôles solides.

Ta mission

À la fin, je serai capable de…

1. énumérer les catégories de l'OWASP Top 10 et leurs défenses
2. reconnaître et corriger injection SQL, injection de commande, traversée de chemin, désérialisation dangereuse
3. appliquer la validation par liste blanche et l'encodage à la sortie
4. éviter l'assignation en masse et les erreurs de configuration
5. écrire des tests de sécurité automatisés

Défi d'échauffement

Défi d'échauffement

Un guichet accepte des formulaires. Un client remplit « Nom : Dupont ». Un autre remplit « Nom : Dupont' ; efface tous les dossiers ; ». Si l'employé **recopie telle quelle** la phrase dans le tampon qui déclenche les actions de l'ordinateur, que risque-t-il d'arriver ? Quelle différence y a-t-il entre **donnée** et **instruction**, et comment un guichet bien conçu s'assure-t-il que l'une ne devienne jamais l'autre ?

Le concept

L'OWASP Top 10 (édition 2021)

L'**OWASP** (*Open Worldwide Application Security Project*) publie une liste de référence des risques applicatifs les plus critiques.

#	Catégorie	En une phrase	Défense principale
A01	**Contrôle d'accès défaillant**	un utilisateur agit hors de ses droits (IDOR, élévation)	autorisation côté serveur, refus par défaut
A02	**Défaillances cryptographiques**	données sensibles mal protégées	TLS, chiffrement au repos, algorithmes sûrs, gestion des clés
A03	**Injection**	des données interprétées comme des instructions (SQL, shell, LDAP, XSS)	requêtes paramétrées, encodage, validation

Le concept

L'OWASP Top 10 (édition 2021) (suite)

#	Catégorie	En une phrase	Défense principale
A04	**Conception non sécurisée**	l'architecture elle-même est fragile	modélisation des menaces, schémas sûrs
A05	**Mauvaise configuration de sécurité**	valeurs par défaut, services inutiles, erreurs verbeuses	durcissement, configuration reproductible
A06	**Composants vulnérables ou obsolètes**	bibliothèques non mises à jour	inventaire, mises à jour, analyse de dépendances
A07	**Défaillances d'identification et d'authentification**	mots de passe faibles, sessions mal gérées	MFA, hachage correct, limitation de tentatives
A08	**Défaillances d'intégrité des logiciels et des données**	mises à jour non vérifiées, désérialisation dangereuse, chaîne CI/CD fragile	signatures, formats sûrs, revue de la chaîne
A09	**Défaillances de journalisation et de surveillance**	attaques non détectées	journaux d'audit, alertes

Le concept

L'OWASP Top 10 (édition 2021) (suite)

#	Catégorie	En une phrase	Défense principale
A10	**SSRF**	le serveur appelle une cible choisie par l'attaquant	validation d'URL, segmentation

Le principe commun de l'injection

> *Une **injection** survient quand des **données** non fiables sont **mélangées à une instruction** qui est ensuite interprétée.*

La défense est toujours la même : **séparer les données du code**.

Contexte	Mauvaise pratique	Bonne pratique
SQL	concaténer dans la requête	**requêtes paramétrées**

Le concept

Le principe commun de l'injection (suite)

Contexte	Mauvaise pratique	Bonne pratique
Shell	`shell=True` avec du texte utilisateur	`subprocess.run([...])` avec liste d'arguments
HTML	insérer dans la page	encodage de sortie
Chemins de fichiers	concaténer `base + nom`	normaliser et vérifier que le résultat reste dans `base`
Évaluation de code	`eval`, `exec`	ne jamais évaluer une entrée externe
Désérialisation	`pickle.loads` sur des données externes	formats de données (JSON) avec schéma

Valider les entrées

La validation vérifie que l'entrée **correspond à ce qu'on attend** :

Le concept

Valider les entrées (suite)

- **liste blanche** (*allowlist*) plutôt que liste noire : on définit ce qui est permis, on refuse tout le reste ;
- vérifier **type, longueur, plage, format, ensemble de valeurs** ;
- valider **côté serveur** (la validation côté client n'est qu'un confort) ;
- **normaliser avant de valider** (encodage, casse, chemins) pour éviter les contournements ;
- les messages d'erreur ne révèlent pas les détails internes.

Valider **ne remplace pas** l'encodage à la sortie ni les requêtes paramétrées : ce sont des défenses **complémentaires**.

Le principe de défense en profondeur appliqué

Le concept

Le principe de défense en profondeur appliqué (suite)

Pour la recherche de cours : (1) valider la longueur et les caractères ; (2) requête **paramétrée** ; (3) compte de base de données **en lecture seule** ; (4) journaliser les motifs suspects ; (5) limiter le débit. Si une couche échoue, les autres limitent les dégâts.

Assignation en masse

Un objet est rempli directement avec tous les champs reçus (`utilisateur.update(**json)`) : l'attaquant ajoute `"role": "admin"`. **Défense** : liste blanche des champs acceptés.

Erreurs de configuration

Mode débogage activé en production, pages d'erreur avec traces, comptes par défaut, listes de répertoires, ports inutiles, services de test exposés, droits trop larges. La parade : une configuration **versionnée, revue et automatisée**, avec contrôle de conformité.

Le concept

Tester la sécurité

Chaque faille corrigée doit devenir un **test de régression** : on rejoue l'attaque et on vérifie qu'elle échoue.

🧠 ***Astuce*** *: devant toute entrée externe, demande-toi : qu'est-ce qui interprète cette donnée ensuite ? La réponse (base, shell, navigateur, système de fichiers, sérialiseur) te dit quelle défense appliquer.*

Exemple pas à pas

Problème : reproduire quatre failles sur du code jouet, puis les corriger et prouver la correction par des tests.

```
import json, pickle, sqlite3, subprocess, tempfile
from pathlib import Path

# ======= 1. Injection SQL =======
bd = sqlite3.connect(":memory:")
bd.executescript('''
CREATE TABLE cours (code TEXT, titre TEXT, prive INTEGER);
INSERT INTO cours VALUES ('HEF-000', 'Mentalité', 0), ('HEF-010', 'Algorithmes', 0
   ), ('SECRET-1', 'Plan de licenciement', 1);
''')

def rechercher_vulnerable(terme):
    return bd.execute("SELECT code, titre FROM cours WHERE prive = 0 AND titre LIK
   E '%" + terme + "%'").fetchall()
```

Exemple pas à pas

Exemple pas à pas (suite)

```
def rechercher(terme):
    if len(terme) > 50:
        raise ValueError("terme trop long")
    return bd.execute("SELECT code, titre FROM cours WHERE prive = 0 AND titre LIK
   E ?", (f"%{terme}%",)).fetchall()

attaque = "x%' OR prive = 1 --"
assert ("SECRET-1", "Plan de licenciement") in rechercher_vulnerable(attaque)
      # faille : le privé fuit
assert rechercher(attaque) == []
       # corrigé : traité comme du texte

# ======= 2. Injection de commande =======
def saluer_vulnerable(nom):
    return subprocess.run(f"echo Bonjour {nom}", shell=True, capture_output=True,
   text=True).stdout.splitlines()

def saluer(nom):
    return subprocess.run(["echo", "Bonjour", nom], capture_output=True, text=True
```

Exemple pas à pas (suite)

```
    ).stdout.splitlines()

charge = "Ada; echo COMMANDE_INJECTEE"
assert saluer_vulnerable(charge) == ["Bonjour Ada", "COMMANDE_INJECTEE"]
        # faille : deuxième commande exécutée
assert saluer(charge) == ["Bonjour Ada; echo COMMANDE_INJECTEE"]
         # corrigé : simple texte

# ======= 3. Traversée de chemin =======
with tempfile.TemporaryDirectory() as dossier:
    base = Path(dossier, "supports").resolve()
    base.mkdir()
    (base / "cours.txt").write_text("contenu public")
    Path(dossier, "secret.txt").write_text("contenu secret")

    def lire_vulnerable(nom):
        return (base / nom).read_text()

    def lire(nom):
```

Exemple pas à pas

Exemple pas à pas (suite)

```
        chemin = (base / nom).resolve()
        if not chemin.is_relative_to(base):
            raise PermissionError("chemin hors du dossier autorisé")
        return chemin.read_text()

    assert lire_vulnerable("../secret.txt") == "contenu secret"
         # faille
    assert lire("cours.txt") == "contenu public"
    for mauvais in ("../secret.txt", "../../etc/passwd", "/etc/passwd", "supports/
   ../../secret.txt"):
        try:
            lire(mauvais)
            raise AssertionError(mauvais)
        except (PermissionError, FileNotFoundError):
            pass

# ======= 4. Désérialisation dangereuse =======
class Piege:
    def __reduce__(self):
```

Exemple pas à pas

Exemple pas à pas (suite)

```
        return (print, ("⚠️ du code s'est exécuté pendant le chargement",))
    # ici inoffensif : un simple print

pickle.loads(pickle.dumps(Piege()))                    # pickle exécute du code au
    chargement

def charger_json(texte):
    donnees = json.loads(texte)                         # JSON ne produit que des
    données
    if not isinstance(donnees, dict) or set(donnees) - {"code", "titre"}:
        raise ValueError("format invalide")
    return donnees

assert charger_json('{"code": "HEF-020", "titre": "Structures"}') == {"code": "HEF
    -020", "titre": "Structures"}
try:
    charger_json('{"code": "X", "role": "admin"}')
    raise AssertionError
except ValueError:
```

Exemple pas à pas (suite)

```
    pass
print("quatre failles reproduites et corrigées")
```

Lecture :

1. **SQL** : la charge `x%' OR prive = 1 --` transforme la requête en « tout, y compris le privé ». Le paramètre `?` envoie la valeur **séparément** de la requête : elle ne peut plus changer sa structure.
2. **Commande** : avec `shell=True`, le `;` est interprété par le shell. Avec une **liste d'arguments**, il n'y a pas de shell : la chaîne reste un argument.
3. **Chemin** : `resolve()` normalise (`..`, liens symboliques), puis `is_relative_to` vérifie le confinement.
4. **Désérialisation** : `pickle` peut **exécuter du code** au chargement (ici un simple `print`, mais cela pourrait être n'importe quoi). Pour des données externes : JSON + validation de schéma.

Exemple pas à pas (suite)

Chaque correction repose sur la même idée : **séparer la donnée de l'instruction**, puis valider ce qui est attendu.

À toi, avec aide

Écris `valider_inscription(donnees)` qui valide un dictionnaire reçu d'un formulaire par **liste blanche** :

- champs autorisés : `nom` (1 à 60 caractères, lettres, espaces, apostrophes, tirets), `email` (format simple, ≤ 120 caractères), `age` (entier 13 à 120), `cours` (parmi `{"HEF-000", "HEF-010", "HEF-020"}`) ;
- tout autre champ (par exemple `role`) provoque une erreur ;
- renvoie `(donnees_nettoyees, erreurs)` : les erreurs sont une liste de messages **par champ**, sans révéler de détails internes.

Teste avec 10 entrées (valides, invalides, charges d'attaque).

À toi, avec aide · réponse

Indices progressifs

Indice 1

Convertis `age` avec `int` dans un `try`.

Indice 2

Utilise `re.fullmatch` avec des motifs stricts.

Indice 3

Compare `set(donnees)` aux champs autorisés pour détecter les champs en trop.

Réponse

```
import re

AUTORISES = {"nom", "email", "age", "cours"}
COURS = {"HEF-000", "HEF-010", "HEF-020"}
NOM = re.compile(r"[A-Za-zÀ-ÖØ-öø-ÿ][A-Za-zÀ-ÖØ-öø-ÿ '\-]{0,59}")
EMAIL = re.compile(r"[A-Za-z0-9._%+\-]+@[A-Za-z0-9.\-]+\.[A-Za-z]{2,}")

def valider_inscription(donnees):
    erreurs, propre = [], {}
    if not isinstance(donnees, dict):
        return {}, ["format invalide"]
    for champ in sorted(set(donnees) - AUTORISES):
        erreurs.append(f"champ non autorisé : {champ}")
    nom = donnees.get("nom")
    if not isinstance(nom, str) or not NOM.fullmatch(nom.strip()):
        erreurs.append("nom invalide")
    else:
        propre["nom"] = nom.strip()
    email = donnees.get("email")
```

À toi, avec aide · réponse

Réponse (suite)

```
    if not isinstance(email, str) or len(email) > 120 or not EMAIL.fullmatch(email
   ):
        erreurs.append("e-mail invalide")
    else:
        propre["email"] = email.lower()
    try:
        age = int(donnees.get("age"))
        if not 13 <= age <= 120:
            raise ValueError
        propre["age"] = age
    except (TypeError, ValueError):
        erreurs.append("âge invalide")
    if donnees.get("cours") in COURS:
        propre["cours"] = donnees["cours"]
    else:
        erreurs.append("cours inconnu")
    return propre, erreurs

bon = {"nom": "Ada O'Neil", "email": "Ada@Exemple.fr", "age": "30", "cours": "HEF-
```

À toi, avec aide · réponse

Réponse (suite)

```
   010"}
assert valider_inscription(bon) == ({"nom": "Ada O'Neil", "email": "ada@exemple.fr
   ", "age": 30, "cours": "HEF-010"}, [])
mauvais = [
    dict(bon, role="admin"),
    dict(bon, nom="<script>alert(1)</script>"),
    dict(bon, nom="Ada'; DROP TABLE membres;--"),
    dict(bon, email="pas-un-email"),
    dict(bon, email="a@b.fr\nBcc: victime@x.fr"),
    dict(bon, age="12"),
    dict(bon, age="trente"),
    dict(bon, cours="../../etc/passwd"),
    dict(bon, nom=""),
]
for entree in mauvais:
    _, erreurs = valider_inscription(entree)
    assert erreurs, entree
print("validation par liste blanche ok")
```

Réponse (suite)

Notons que `"Ada'; DROP TABLE..."` est refusé par la validation **et** serait de toute façon inoffensif grâce aux requêtes paramétrées : deux couches indépendantes.

À toi, en solo

Un téléchargeur de fichiers sûr. Écris `servir_fichier(base, nom_demande)` qui sert des fichiers d'un dossier de supports, avec :

1. confinement strict dans `base` (y compris contre les liens symboliques qui sortent du dossier) ;
2. liste blanche d'extensions (`.pdf`, `.txt`, `.md`) ;
3. limite de taille (5 Mo) ;
4. refus des noms contenant des octets nuls, des séparateurs Windows (`\`), ou commençant par un point ;
5. un message d'erreur **unique et vague** (« introuvable ») pour tous les refus, afin de ne pas révéler l'existence de fichiers ;
6. 15 tests d'attaque : `../`, chemin absolu, lien symbolique sortant, `%2e%2e/` déjà décodé, octet nul, nom très long, extension interdite, fichier trop gros.

À toi, en solo · réponse

Pistes de correction

```
import os, tempfile
from pathlib import Path

EXTENSIONS = {".pdf", ".txt", ".md"}
TAILLE_MAX = 5 * 1024 * 1024

class Introuvable(Exception):
    pass

def servir_fichier(base, nom_demande):
    base = Path(base).resolve()
    try:
        if not isinstance(nom_demande, str) or not nom_demande or len(nom_demande)
    > 200:
            raise Introuvable
        if "\x00" in nom_demande or "\\" in nom_demande or Path(nom_demande).name.
   startswith("."):
            raise Introuvable
        chemin = (base / nom_demande).resolve()                # suit les liens
```

Pistes de correction (suite)

```
    symboliques
        if not chemin.is_relative_to(base) or chemin == base:
            raise Introuvable
        if chemin.suffix.lower() not in EXTENSIONS or not chemin.is_file():
            raise Introuvable
        if chemin.stat().st_size > TAILLE_MAX:
            raise Introuvable
        return chemin.read_bytes()
    except (Introuvable, OSError, ValueError):
        raise Introuvable("introuvable") from None              # message unique

with tempfile.TemporaryDirectory() as d:
    racine = Path(d)
    base = racine / "supports"
    base.mkdir()
    (base / "cours.txt").write_text("public")
    (base / ".cache").write_text("caché")
    (base / "script.py").write_text("print(1)")
    (base / "gros.pdf").write_bytes(b"x" * (TAILLE_MAX + 1))
```

À toi, en solo · réponse

Pistes de correction (suite)

```
    (racine / "secret.txt").write_text("secret")
    os.symlink(racine / "secret.txt", base / "lien.txt")                    # li
   en qui sort du dossier

    assert servir_fichier(base, "cours.txt") == b"public"
    attaques = ["../secret.txt", "/etc/passwd", str(racine / "secret.txt"), "lien.
   txt", "cours.txt\x00.png", "a" * 500 + ".txt",
                "script.py", "gros.pdf", ".cache", "..\\secret.txt", "", ".", "sup
   ports/../../secret.txt", "dossier_inexistant/x.txt", None]
    for a in attaques:
        try:
            servir_fichier(base, a)
            raise AssertionError(f"accepté : {a!r}")
        except Introuvable as e:
            assert str(e) == "introuvable"
print("15 attaques refusées avec un message unique")
```

À toi, en solo · réponse

Pistes de correction (suite)

Le point clé : on **normalise d'abord** (`resolve`) puis on **vérifie** le confinement. Vérifier le texte du nom avant normalisation est la source de la plupart des contournements.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Concaténer des données dans une requête ou une commande	Injection	Paramètres, listes d'arguments, encodage
Liste noire de caractères dangereux	Contournements (encodages, variantes)	Liste blanche de ce qui est permis
Valider côté client uniquement	Contournement immédiat	Toujours valider côté serveur
Désérialiser des données externes avec `pickle`	Exécution de code arbitraire	JSON + validation de schéma

Mini-boss

Mini-boss

Audit défensif d'une petite application « jouet ». Écris en 150 lignes une petite application (module de fonctions, sans framework) qui contient **volontairement** huit failles correspondant à huit catégories de l'OWASP : injection SQL, injection de commande, traversée de chemin, désérialisation dangereuse, contrôle d'accès cassé (IDOR), assignation en masse, secret en dur, message d'erreur trop bavard.

1. Pour chaque faille, écris le **test d'attaque** qui la démontre (il doit réussir sur la version vulnérable).
2. Corrige chaque faille ; le même test doit maintenant **échouer à attaquer** (l'attaque ne marche plus).
3. Ajoute un test de non-régression par faille.
4. Rédige un tableau : faille → catégorie OWASP → cause → correction → test.
5. Explique en 10 lignes quelle défense a le meilleur rapport efficacité/effort, et pourquoi la validation seule ne suffit pas.

Mini-boss · réponse

Pistes de correction

Un bon travail : une fonction `app_vulnerable` regroupant `rechercher`, `exporter`, `lire_support`, `charger_profil_pickle`, `lire_profil(id)` sans contrôle de propriétaire, `mettre_a_jour_profil(**champs)` sans liste blanche, `CLE_API = "sk_live_..."`, et `repondre_erreur(e)` qui renvoie la trace complète. La version corrigée utilise requêtes paramétrées, `subprocess.run([...])`, confinement par `resolve()`/`is_relative_to`, JSON + schéma, vérification `proprietaire == utilisateur`, liste blanche `{"nom", "email"}`, variable d'environnement, message générique + identifiant d'incident journalisé. Le tableau relie A03, A03, A01 (traversée), A08, A01, A04/A01, A02/A05, A05. Le meilleur rapport efficacité/effort est la **séparation donnée/instruction** (paramètres, listes d'arguments) : elle neutralise toute une famille de failles à faible coût, alors que la validation seule dépend d'une liste exhaustive de cas.

Quiz éclair

Quiz éclair

1. L'injection consiste à :

A. mélanger des données non fiables à une instruction interprétée
B. chiffrer deux fois
C. saturer un serveur

2. Pour éviter l'injection SQL, on utilise :

A. des guillemets doubles
B. des requêtes paramétrées
C. des majuscules

3. La validation d'entrées doit privilégier :

A. la liste noire des caractères dangereux

Quiz éclair

Quiz éclair (suite)

B. l'absence de contrôle
C. la liste blanche de ce qui est permis

4. 🔁 *Révision mission 06* : Le XSS est évité principalement par :

A. l'encodage des données à la sortie selon le contexte
B. un pare-feu réseau
C. un mot de passe plus long

5. 🔁 *Révision mission 06* : Un jeton anti-CSRF prouve :

A. que l'utilisateur est majeur
B. que la requête vient bien du site légitime avec l'accord de l'utilisateur
C. que le mot de passe est fort

Quiz éclair · réponse

Réponses

1. **A**.
2. **B**.
3. **C**.
4. **A**.
5. **B**.

Je réfléchis

1. Dans quelles situations de ton travail passes-tu une donnée externe à un interpréteur (shell, base, navigateur, fichier) sans t'en rendre compte ?
2. Pourquoi est-ce plus durable de corriger la cause (séparer donnée et code) que de bloquer une charge précise ?

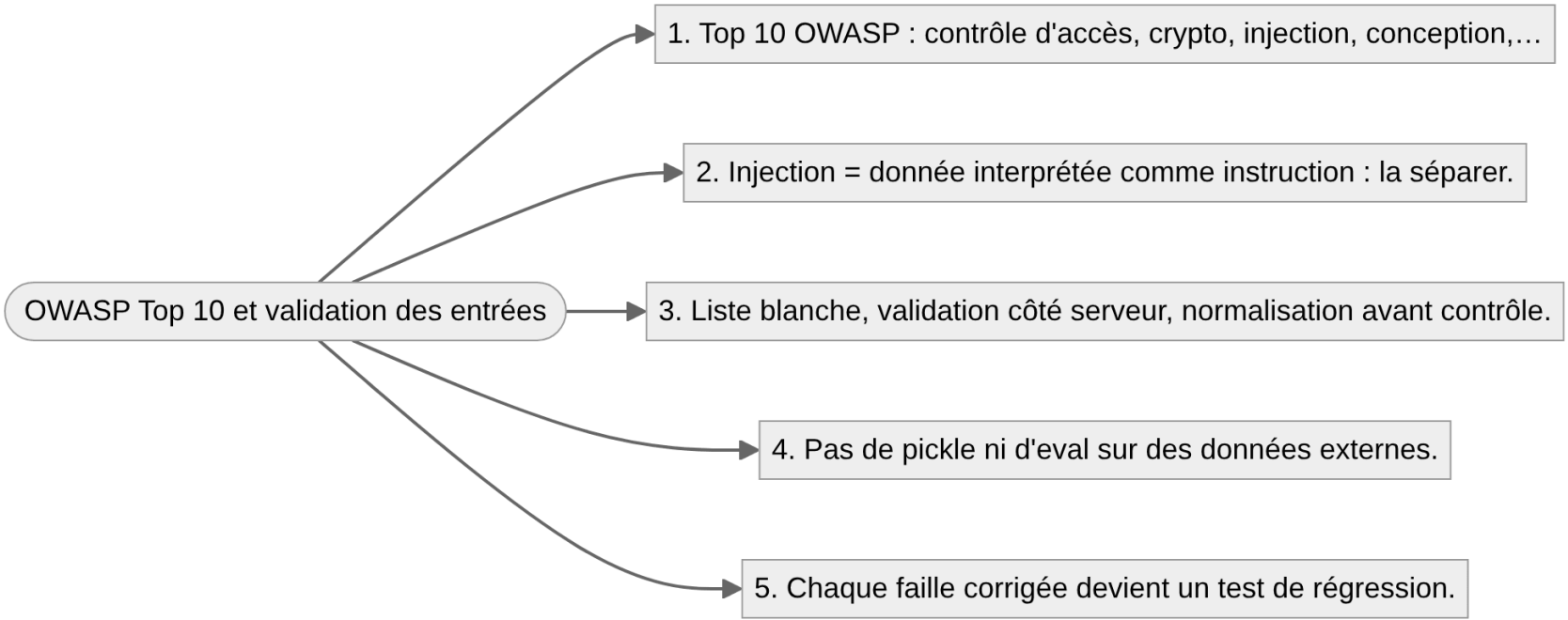
À retenir

À retenir

1. Top 10 OWASP : contrôle d'accès, crypto, injection, conception, configuration, composants, authentification, intégrité, journalisation, SSRF.
2. Injection = donnée interprétée comme instruction : la séparer.
3. Liste blanche, validation côté serveur, normalisation avant contrôle.
4. Pas de `pickle` ni d'`eval` sur des données externes.
5. Chaque faille corrigée devient un test de régression.

À retenir

Le schéma de la mission



MISSION ACCOMPLIE

+100 XP

Badge débloqué : Chasseur de failles (côté défense)

Prochaine mission : Sécurité réseau et système

