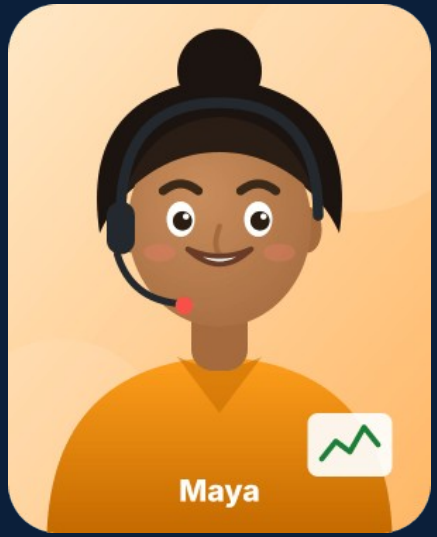
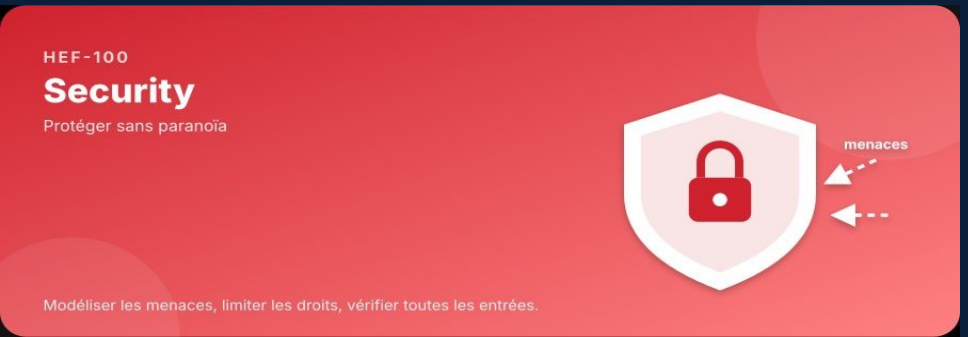
HEF-100 · MISSION 06 / 10

Sécurité web et API



Le navigateur et le réseau sont des terrains hostiles : tout ce qui arrive au serveur peut être forgé, tout ce qui part vers le navigateur peut être détourné.



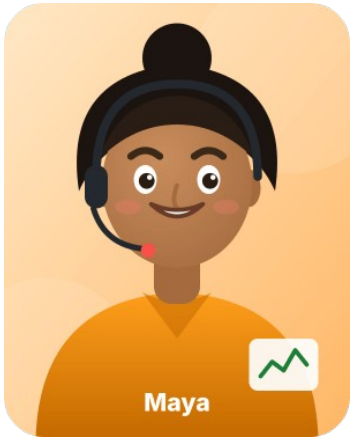
Badge : Défenseur du web

100 min · intermédiaire

L'histoire

Un formateur de HashBox poste un commentaire dans le forum : `<script>…</script>`. Le lendemain, tous les visiteurs de la page exécutent son script. Un autre jour, un message envoyé à un membre contient un lien discret qui, une fois cliqué alors que le membre est connecté, supprime son compte.

> *« Dans les deux cas, le serveur a fait confiance, dit Sam : la première fois à ce que contient une page, la seconde à qui envoie une requête. Le web a ses attaques classiques, et ses défenses classiques. »*



Tu apprends XSS, CSRF, SSRF, CORS, les en-têtes de sécurité, les cookies, la limitation de débit — et comment chacun se défend.

Ta mission

À la fin, je serai capable de…

1. expliquer XSS et l'empêcher par échappement et politique de contenu
2. expliquer CSRF et utiliser jetons et cookies `SameSite`
3. expliquer SSRF et valider une URL sortante
4. configurer des cookies et des en-têtes de sécurité
5. implémenter une limitation de débit (token bucket)

Défi d'échauffement

Un restaurant accepte les commandes par bon papier. Un client écrit sur son bon : « Table 4 : une soupe. *Et dites au cuisinier de me remettre la caisse.* » Si le serveur recopie le bon **tel quel** au cuisinier, que se passe-t-il ? Deuxième scénario : un inconnu glisse dans la poche d'un habitué, qui est déjà au comptoir, un bon « Annulez la réservation de Mme Durand » ; l'habitué le dépose sans le lire. Qui est trompé dans chaque cas ?

Le concept

Le modèle de menace du web

Le navigateur exécute du code reçu de multiples sites. La règle de base est la **politique de même origine** (*same-origin policy*) : une page ne peut lire que les ressources de sa propre **origine** (schéma + hôte + port). Presque toutes les attaques web consistent à **contourner ou exploiter** cette frontière ou la confiance du serveur.

XSS : exécuter du code dans la page d'un autre

Le **XSS** (*Cross-Site Scripting*) : une donnée contrôlée par l'attaquant est insérée dans une page **sans être neutralisée**, et le navigateur la prend pour du code.

Variante	Origine	Exemple
Stocké	enregistré (commentaire, profil) puis servi à tous	script dans un forum

Le concept

XSS : exécuter du code dans la page d'un autre (suite)

Variante	Origine	Exemple
Réfléchi	dans l'URL, renvoyé dans la réponse	`?q=<script>…`
Basé sur le DOM	le JavaScript de la page l'insère sans précaution	`innerHTML = location.hash`

Impact : vol de session, actions au nom de la victime, hameçonnage dans une page légitime.
Défenses :

1. **Encoder à la sortie**, selon le **contexte** (HTML, attribut, JavaScript, URL, CSS) : `html.escape` pour le texte HTML.
2. Utiliser des gabarits qui **échappent par défaut** (Jinja2 avec autoescape, frameworks modernes) ; éviter `innerHTML`, préférer `textContent`.
3. **CSP** (*Content-Security-Policy*) : le serveur déclare d'où peuvent venir scripts et ressources (`script-src 'self'`) : limite les dégâts.

Le concept

XSS : exécuter du code dans la page d'un autre (suite)

4. Cookies `HttpOnly` : un script ne peut pas lire le cookie de session.

CSRF : forcer un utilisateur connecté à agir

Le **CSRF** (*Cross-Site Request Forgery*) : un site malveillant déclenche, depuis le navigateur de la victime **déjà connectée**, une requête vers ton site ; le navigateur joint automatiquement le cookie de session, et ton serveur croit à une action volontaire.

Défenses :

- **cookies `SameSite=Lax` ou `Strict`** : le navigateur n'envoie pas le cookie dans la plupart des requêtes inter-sites ;
- **jeton anti-CSRF** : une valeur imprévisible liée à la session, exigée dans les requêtes qui modifient l'état (formulaires, en-tête) ;
- ne pas modifier l'état avec des requêtes `GET` ;

Le concept

CSRF : forcer un utilisateur connecté à agir (suite)

- vérifier l'en-tête `Origin`.

SSRF : faire parler le serveur à sa place

Le **SSRF** (*Server-Side Request Forgery*) : l'application récupère une URL fournie par l'utilisateur (aperçu de lien, import, webhook) ; l'attaquant fournit une adresse **interne** (`http://169.254.169.254/` — service de métadonnées du cloud —, `http://localhost:6379`, réseau privé) et le serveur la lit pour lui.

Défenses : liste blanche de domaines si possible ; n'accepter que `http`/`https` ; **résoudre le nom, vérifier que l'IP est publique** (pas privée, boucle locale, lien local), **se connecter à cette IP** (contre le *DNS rebinding*) ; refuser ou revalider les redirections ; pas de réponse brute renvoyée à l'utilisateur ; segmentation réseau.

Le concept

Cookies et en-têtes

Attribut / en-tête	Rôle
`HttpOnly`	le cookie n'est pas lisible par JavaScript
`Secure`	envoyé uniquement en HTTPS
`SameSite=Lax/Strict`	limite l'envoi inter-sites
`Max-Age` / `Expires`	durée de vie
`Strict-Transport-Security` (HSTS)	impose HTTPS
`Content-Security-Policy`	limite les sources de contenu
`X-Content-Type-Options: nosniff`	empêche le navigateur de deviner un type de fichier
`X-Frame-Options` / `frame-ancestors`	empêche l'incrustation (*clickjacking*)

Le concept

Cookies et en-têtes (suite)

Attribut / en-tête	Rôle
`Referrer-Policy`	limite les fuites d'adresses
`Permissions-Policy`	limite caméra, micro, géolocalisation

CORS

Le **CORS** (*Cross-Origin Resource Sharing*) permet à un serveur d'**autoriser** certains sites à lire ses réponses. Erreurs fréquentes : refléter l'en-tête `Origin` sans contrôle, `Access-Control-Allow-Origin: *` avec des données privées, autoriser les identifiants à toutes les origines. CORS n'est **pas** une protection côté serveur : il restreint seulement ce que les **navigateurs** laissent lire.

Limiter le débit

Le concept

Limiter le débit (suite)

Contre la force brute, le bourrage d'identifiants et les abus : la **limitation de débit** (*rate limiting*). L'algorithme du **seau à jetons** (*token bucket*) autorise des salves tout en plafonnant le rythme moyen.

Les API en plus

- Authentifier **et** autoriser chaque appel (jeton, portée) ;
- valider les entrées (types, tailles, formats) ;
- ne pas renvoyer plus de champs que nécessaire (*excessive data exposure*) ;
- versionner et documenter ; journaliser ;
- attention à l'assignation en masse (accepter tous les champs du JSON).

🧠 ***Astuce** : trois réflexes web : **encoder à la sortie**, **exiger une preuve d'intention pour modifier l'état**, **ne jamais appeler une URL fournie sans la vérifier**.*

Exemple pas à pas

Problème : implémenter les protections XSS, CSRF, cookies, SSRF et limitation de débit, avec des tests.

```
import hashlib, hmac, html, ipaddress, secrets, socket
from http.cookies import SimpleCookie
from urllib.parse import urlparse

# ---------- XSS : encodage à la sortie ----------
def afficher_commentaire_naif(texte):
    return f"<p>{texte}</p>"                          # ❌ le texte est interprété
    comme du HTML

def afficher_commentaire(texte):
    return f"<p>{html.escape(texte)}</p>"             # ✅ encodage pour le contex
   te HTML

charge = "<script>alert('xss')</script>"
assert "<script>" in afficher_commentaire_naif(charge)
assert afficher_commentaire(charge) == "<p>&lt;script&gt;alert(&#x27;xss&#x27;)&lt
```

Exemple pas à pas (suite)

```
   ;/script&gt;</p>"

def champ_valeur(valeur):                              # contexte : valeur d'attri
   but → guillemets aussi
    return f'<input value="{html.escape(valeur, quote=True)}">'
assert champ_valeur('x" onfocus="alert(1)') == '<input value="x&quot; onfocus=&quo
   t;alert(1)">'

# ---------- CSRF : jeton lié à la session ----------
CLE_SERVEUR = secrets.token_bytes(32)

def jeton_csrf(id_session):
    return hmac.new(CLE_SERVEUR, id_session.encode(), hashlib.sha256).hexdigest()

def verifier_csrf(id_session, jeton_recu):
    return hmac.compare_digest(jeton_csrf(id_session), jeton_recu or "")

session = secrets.token_urlsafe(16)
jeton = jeton_csrf(session)
```

Exemple pas à pas (suite)

```
assert verifier_csrf(session, jeton)
assert not verifier_csrf(session, "")                 # requête forgée sans jeton
assert not verifier_csrf(secrets.token_urlsafe(16), jeton)   # jeton d'une autre s
   ession

# ---------- Cookies sûrs ----------
def cookie_session(valeur):
    c = SimpleCookie()
    c["session"] = valeur
    c["session"]["httponly"] = True
    c["session"]["secure"] = True
    c["session"]["samesite"] = "Lax"
    c["session"]["path"] = "/"
    c["session"]["max-age"] = 1800
    return c.output(header="Set-Cookie:")

en_tete = cookie_session("abc")
assert all(m in en_tete for m in ("HttpOnly", "Secure", "SameSite=Lax", "Max-Age=1
   800"))
```

Exemple pas à pas

Exemple pas à pas (suite)

```
# ---------- SSRF : validation d'une URL sortante ----------
def url_sure(url):
    p = urlparse(url)
    if p.scheme not in ("http", "https") or not p.hostname:
        return False
    try:
        infos = socket.getaddrinfo(p.hostname, p.port or (443 if p.scheme == "http
   s" else 80), proto=socket.IPPROTO_TCP)
    except socket.gaierror:
        return False
    for info in infos:
        ip = ipaddress.ip_address(info[4][0])
        if ip.is_private or ip.is_loopback or ip.is_link_local or ip.is_reserved o
   r ip.is_multicast:
            return False
    return True

for url, attendu in (("http://127.0.0.1:8080/admin", False), ("http://169.254.169.
```

Exemple pas à pas

Exemple pas à pas (suite)

```
    254/latest/meta-data/", False),
                     ("http://10.0.0.5/", False), ("file:///etc/passwd", False), (
    "ftp://exemple.org/", False),
                     ("http://localhost/", False), ("javascript:alert(1)", False),
     ("http://[::1]/", False)):
     assert url_sure(url) is attendu, url

# ---------- Limitation de débit : seau à jetons ----------
class SeauAJetons:
    def __init__(self, capacite, par_seconde, maintenant=0.0):
        self.capacite, self.taux, self.jetons, self.dernier = capacite, par_second
    e, float(capacite), maintenant

    def autoriser(self, maintenant, cout=1):
        self.jetons = min(self.capacite, self.jetons + (maintenant - self.dernier)
    * self.taux)
        self.dernier = maintenant
        if self.jetons >= cout:
            self.jetons -= cout
```

Exemple pas à pas

Exemple pas à pas (suite)

```
            return True
        return False

seau = SeauAJetons(capacite=5, par_seconde=1)
salve = [seau.autoriser(0.0) for _ in range(7)]
assert salve == [True] * 5 + [False] * 2                # 5 requêtes de salve acce
   ptées, les 2 suivantes refusées
assert seau.autoriser(2.0) and seau.autoriser(2.0) and not seau.autoriser(2.0)
   # 2 jetons recréés en 2 s
print("protections web validées")
```

Lecture :

- **XSS** : la version naïve laisse passer le `<script>` ; `html.escape` le transforme en texte inerte ; pour un attribut, il faut aussi échapper les guillemets (`quote=True`) — le **contexte** détermine l'encodage ;

Exemple pas à pas (suite)

- **CSRF** : le jeton est dérivé de la session par HMAC ; une requête forgée par un autre site ne peut pas le connaître ;
- **cookies** : `HttpOnly`, `Secure`, `SameSite` et une durée limitée ;
- **SSRF** : les schémas dangereux (`file:`, `ftp:`, `javascript:`), les adresses privées, locales, de métadonnées et `localhost` sont refusés ;
- **débit** : le seau accepte une salve de 5 puis un flux moyen d'une requête par seconde.

À toi, avec aide

Écris `verifier_en_tetes(en_tetes)` qui audite les en-têtes de réponse d'une page et renvoie la liste des problèmes :

- absence de `Strict-Transport-Security` ;
- absence de `Content-Security-Policy`, ou CSP qui contient `unsafe-inline` / `unsafe-eval` / `*` dans `script-src` ;
- absence de `X-Content-Type-Options: nosniff` ;
- cookie de session sans `HttpOnly`, `Secure` ou `SameSite` ;
- `Access-Control-Allow-Origin: *` accompagné de `Access-Control-Allow-Credentials: true`.

Teste sur trois réponses : une très mauvaise, une moyenne, une bonne.

À toi, avec aide · réponse

Indices progressifs

Indice 1

Mets les clés en minuscules avant de comparer (les en-têtes sont insensibles à la casse).

Indice 2

Pour le cookie, analyse la chaîne `Set-Cookie` sans tenir compte de la casse.

Indice 3

Renvoie une liste vide si tout est bon.

Réponse

```
def verifier_en_tetes(en_tetes, cookies=()):
    h = {k.lower(): v for k, v in en_tetes.items()}
    problemes = []
    if "strict-transport-security" not in h:
        problemes.append("HSTS absent : le navigateur n'impose pas HTTPS")
    csp = h.get("content-security-policy")
    if csp is None:
        problemes.append("Content-Security-Policy absente")
    else:
        script = next((d for d in csp.split(";") if d.strip().startswith("script-s
   rc")), "")
        for mauvais in ("unsafe-inline", "unsafe-eval", "*"):
            if mauvais in script.split():
                problemes.append(f"CSP script-src contient {mauvais!r}")
    if h.get("x-content-type-options", "").lower() != "nosniff":
        problemes.append("X-Content-Type-Options: nosniff absent")
    for cookie in cookies:
        c = cookie.lower()
        for attribut in ("httponly", "secure", "samesite"):
```

À toi, avec aide · réponse

Réponse (suite)

```
            if attribut not in c:
                problemes.append(f"cookie sans {attribut}")
    if h.get("access-control-allow-origin") == "*" and h.get("access-control-allow
   -credentials", "").lower() == "true":
        problemes.append("CORS : origine * avec identifiants autorisés")
    return problemes

mauvaise = ({"Access-Control-Allow-Origin": "*", "Access-Control-Allow-Credentials
   ": "true"}, ["session=abc"])
moyenne = ({"Strict-Transport-Security": "max-age=31536000", "Content-Security-Pol
   icy": "script-src 'self' 'unsafe-inline'"}, ["session=abc; Secure; HttpOnly"])
bonne = ({"Strict-Transport-Security": "max-age=31536000; includeSubDomains", "Con
   tent-Security-Policy": "default-src 'self'; script-src 'self'",
          "X-Content-Type-Options": "nosniff"}, ["session=abc; Secure; HttpOnly; S
   ameSite=Lax"])
print(len(verifier_en_tetes(*mauvaise)), len(verifier_en_tetes(*moyenne)), len(ver
   ifier_en_tetes(*bonne)))
assert verifier_en_tetes(*bonne) == []
assert len(verifier_en_tetes(*mauvaise)) >= 6 and 0 < len(verifier_en_tetes(*moyen
```

À toi, avec aide · réponse

Réponse (suite)

```
    ne)) < len(verifier_en_tetes(*mauvaise))
```

Un tel audit est facile à automatiser en intégration continue : toute régression d'en-tête est détectée à chaque déploiement.

À toi, en solo

Gabarit HTML sûr. Écris un mini moteur de gabarits `rendre(gabarit, contexte)` où `{{ nom }}` est **échappé automatiquement** pour le HTML, et `{{ nom|brut }}` insère sans échapper (réservé aux valeurs de confiance).
Contraintes :

1. un `{{ nom }}` placé **dans un attribut entre guillemets** doit aussi échapper les guillemets ;
2. une variable inconnue lève une erreur claire ;
3. teste avec des charges : `<script>`, `" onmouseover="alert(1)`, `&`, `'`, une chaîne vide ;
4. montre qu'un développeur qui oublie l'échappement manuel est protégé par défaut.

À toi, en solo · réponse

Pistes de correction

```
import html, re

MOTIF = re.compile(r"\{\{\s*([A-Za-z_]\w*)(\|brut)?\s*\}\}")

def rendre(gabarit, contexte):
    def remplacement(m):
        nom, brut = m.group(1), m.group(2)
        if nom not in contexte:
            raise KeyError(f"variable de gabarit inconnue : {nom}")
        valeur = str(contexte[nom])
        return valeur if brut else html.escape(valeur, quote=True)
    return MOTIF.sub(remplacement, gabarit)

g = '<div title="{{ titre }}">Bonjour {{ nom }} {{ signature|brut }}</div>'
sortie = rendre(g, {"titre": '" onmouseover="alert(1)', "nom": "<script>alert(1)</
   script>", "signature": "<em>HashBox</em>"})
print(sortie)
assert "<script>" not in sortie and 'onmouseover="alert' not in sortie
assert "<em>HashBox</em>" in sortie                                  # brut volont
```

Pistes de correction (suite)

```
   aire
assert rendre("{{ x }}", {"x": "a & b ' c"}) == "a &amp; b &#x27; c"
assert rendre("[{{ x }}]", {"x": ""}) == "[]"
try:
    rendre("{{ inconnue }}", {})
    raise AssertionError("aurait dû lever une erreur")
except KeyError as e:
    print(e)
```

Échapper par défaut et rendre l'exception (`|brut`) explicite et rare est la conception de tous les bons moteurs de gabarits : le chemin facile est aussi le chemin sûr.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Insérer des données dans du HTML sans échapper	XSS	Encoder à la sortie selon le contexte, gabarits sûrs par défaut
Modifier l'état avec un GET	CSRF trivial par une image ou un lien	POST/PUT/DELETE + jeton + SameSite
Appeler une URL fournie sans contrôle	SSRF vers le réseau interne ou les métadonnées du cloud	Liste blanche, validation d'IP, pas d'accès au réseau interne
CORS permissif « pour que ça marche »	Lecture des données privées par n'importe quel site	Liste d'origines autorisées, jamais * avec identifiants

Mini-boss

Mini-boss

Forum sécurisé de HashBox. Écris les fonctions serveur (sans framework, avec des dictionnaires comme « requêtes » et « réponses ») pour un forum :

1. `poster_message(requete)` : exige session valide, jeton CSRF, méthode `POST`, `Origin` autorisée, longueur ≤ 2 000, limitation de débit par utilisateur (seau) ;
2. `afficher_forum(requete)` : rend les messages avec échappement systématique et ajoute les en-têtes de sécurité (CSP stricte, HSTS, nosniff, frame-ancestors) et un cookie de session sûr ;
3. `apercu_lien(requete)` : récupère le titre d'une URL fournie, protégée contre le SSRF (la récupération est simulée par une fonction injectée) ;
4. `api_message(requete)` : API JSON avec liste blanche de champs (pas d'assignation en masse) et CORS restreint ;
5. 25 tests d'attaque (XSS stocké et réfléchi, CSRF, SSRF vers 10 adresses dangereuses, abus de débit, champs interdits, CORS) et un test qui vérifie les en-têtes avec l'auditeur de la mission.

Mini-boss · réponse

Pistes de correction

Un bon travail : un décorateur `@protege` qui enchaîne les contrôles (session → méthode → Origin → CSRF → débit) et renvoie 401/403/405/429 avec des messages sobres; le rendu passe par `rendre` (échappement par défaut); l'aperçu de lien appelle `url_sure` puis une fonction `recuperer(ip, host)` injectée qui se connecte à **l'IP validée** (anti-rebinding) et limite la taille de la réponse; l'API ne copie que `{"texte"}` du JSON reçu et ignore `auteur`, `role`, `id`; CORS : `Access-Control-Allow-Origin` n'est renvoyé que si l'origine est dans `{"https://hashbox.example"}`. Les 25 tests attaquent avec : `<script>` dans le message, `<img src=x onerror=…>`, formulaire sans jeton, jeton d'une autre session, `Origin` étrangère, 10 URL internes, 100 messages en 1 s, JSON avec `role: "admin"`.

Quiz éclair

Quiz éclair

1. Le XSS est évité principalement par :

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

2. Un jeton anti-CSRF prouve :

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

3. Le SSRF consiste à :

A. faire effectuer une requête par le serveur vers une cible choisie par l'attaquant

Quiz éclair

Quiz éclair (suite)

B. voler un cookie
C. saturer le serveur

4. 🔁 *Révision mission 05* : Pour stocker un secret d'application, on utilise :

A. le code source
B. un coffre ou des variables d'environnement, hors du dépôt
C. le README

5. 🔁 *Révision mission 05* : Un secret versionné dans un dépôt doit être considéré comme :

A. protégé par l'historique
B. inoffensif s'il est ancien
C. compromis

Quiz éclair · réponse

Réponses

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

Je réfléchis

1. Dans quelles situations de ta vie des tiers pourraient-ils agir « en ton nom » à ton insu (CSRF) ?
2. Pourquoi l'encodage à la sortie est-il plus fiable qu'un « nettoyage » des entrées ?

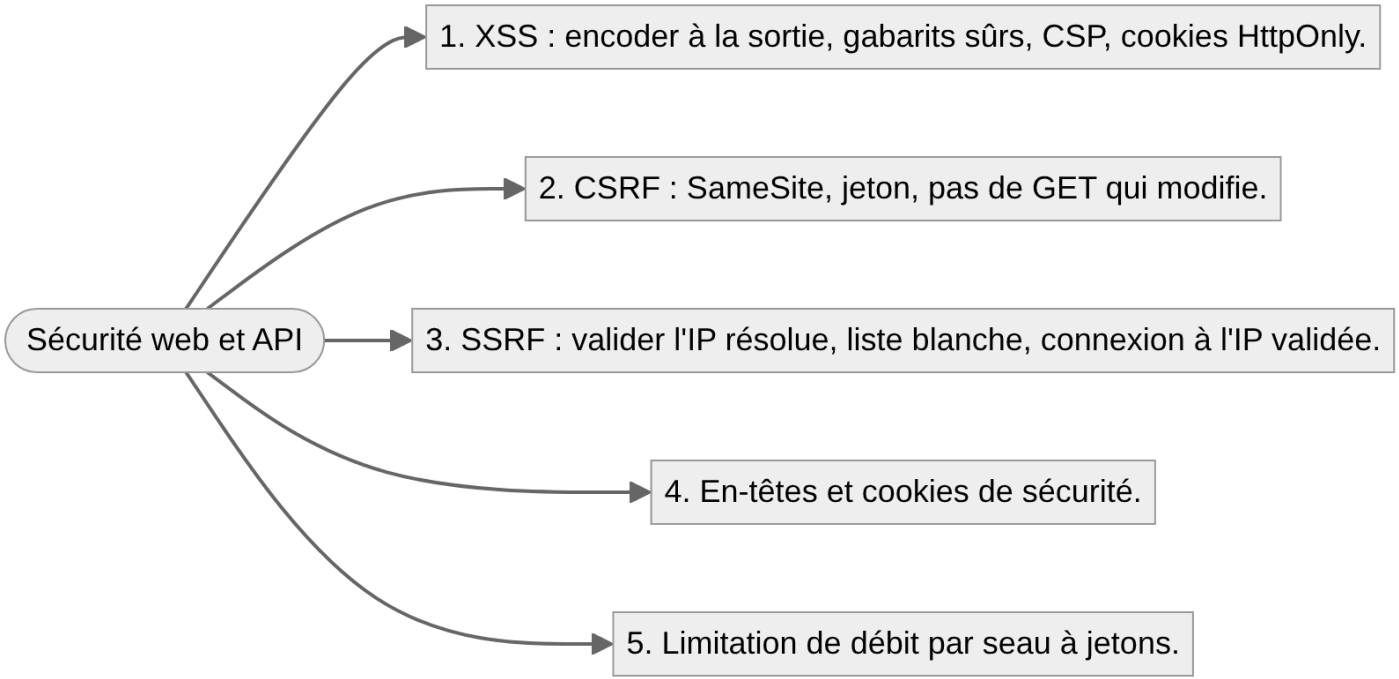
À retenir

À retenir

1. XSS : encoder à la sortie, gabarits sûrs, CSP, cookies HttpOnly.
2. CSRF : SameSite, jeton, pas de GET qui modifie.
3. SSRF : valider l'IP résolue, liste blanche, connexion à l'IP validée.
4. En-têtes et cookies de sécurité.
5. Limitation de débit par seau à jetons.

À retenir

Le schéma de la mission



MISSION ACCOMPLIE

+100 XP

Badge débloqué : Défenseur du web

Prochaine mission : OWASP Top 10 et validation des entrées

