

3A Sécurité Web

Cours 4

Maxime Puys

IUT de Clermont-Ferrand — Université Clermont Auvergne

2026-10-15

Ce document est distribué sous licence Creative Commons BY-NC-SA 4.0. La copie, la distribution et la modification sont autorisées à des fins pédagogiques.

Plan

1. Cadrage
2. Injection SQL
3. Cross-Site Scripting (XSS)
4. Cross-Site Request Forgery (CSRF)
5. Server-Side Request Forgery (SSRF)

6. Inclusion de fichiers (LFI / RFI)
7. Authentification
8. Reconnaissance et test d'intrusion
9. Sécurité Android
10. Investigation numérique (forensic)
11. Conclusion



**IUT CLERMONT
AUVERGNE**

Aurillac - Clermont-Ferrand - Le Puy-en-Velay
Montluçon - Moulins - Vichy

1. Cadrage

Une application web expose un client (le navigateur, que l'attaquant contrôle entièrement) et un serveur, reliés par des requêtes que l'attaquant peut forger librement.

- **Entrées non fiables** — tout ce qui vient du client (paramètres d'URL, corps de formulaire, en-têtes, cookies) est potentiellement hostile
- **Deux cibles distinctes** — le code qui s'exécute chez la victime (le navigateur) et le code qui s'exécute chez l'hébergeur (le serveur)
- **Une confusion à éviter dès maintenant** — savoir, pour chaque attaque, **où** s'exécute le code malveillant est la première question à se poser

⇒ Le rappel du cours 2 sur le Top 10 OWASP donne le panorama ; nous creusons ici les familles qui reviennent en TP.

1. Injecter du code dans une requête vers la base de données § *SQLi*
2. Injecter du code exécuté par le navigateur d'un autre utilisateur § *XSS*
3. Forger une requête au nom d'un utilisateur authentifié § *CSRF*
4. Faire exécuter au serveur un fichier qu'il n'aurait pas dû lire § *LFI/RFI*
5. Casser ou contourner l'authentification elle-même § *Auth*

⇒ Chaque famille a sa cible propre (serveur ou client) et sa contre-mesure propre — les confondre est l'erreur la plus fréquente.



**IUT CLERMONT
AUVERGNE**

Aurillac - Clermont-Ferrand - Le Puy-en-Velay
Montluçon - Moulins - Vichy

2. Injection SQL

Une application construit trop souvent ses requêtes SQL en concaténant directement une entrée utilisateur dans le texte de la requête.

Requête vulnérable (pseudo-code)

```
1 requete = "SELECT * FROM comptes WHERE login='" + login
2           + "'" AND mdp='" + mdp + "'"
```

Si `login` et `mdp` sont insérés sans validation, un attaquant qui contrôle leur contenu contrôle une partie de la **syntaxe** de la requête, pas seulement sa donnée.

⇒ Le moteur SQL ne peut pas distinguer « du texte fourni par l'utilisateur » de « du SQL que l'utilisateur a écrit ». Les deux sont mélangés dans la même chaîne de caractères.

Avec la requête précédente, un attaquant qui saisit comme mot de passe

```
1 ' OR '1'='1
```

obtient la requête suivante, exécutée telle quelle par le serveur :

```
1 SELECT * FROM comptes WHERE login='...' AND mdp='' OR '1'='1'
```

'1'='1' est toujours vraie : la clause `WHERE` sélectionne toutes les lignes de la table, quel que soit le mot de passe réel. Selon le code applicatif, la première ligne renvoyée suffit souvent à authentifier l'attaquant.

⇒ Il ne s'agit pas de deviner un mot de passe : il s'agit de changer la question posée à la base de données.

La concaténation de chaînes est la cause racine : la corriger en filtrant des caractères (échappement) reste fragile, car il faut penser à **tous** les contextes d'injection possibles.

À éviter

Échapper les guillemets à la main, ou s'appuyer sur une fonction d'échappement fournie par le pilote de base de données : facile à oublier à un seul endroit, et certaines de ces fonctions ont depuis disparu des langages qui les proposaient.

Requête préparée / paramétrée

```
1 SELECT * FROM comptes WHERE login=? AND mdp=?
```

Le texte de la requête est envoyé au moteur SQL **séparément** des valeurs. Les valeurs ne sont jamais interprétées comme du SQL, quel que soit leur contenu.

⇒ Requêtes préparées, moindre privilège du compte applicatif, pare-feu applicatif (WAF, cours 2) : trois couches indépendantes, la première étant la seule vraiment suffisante à elle seule.



**IUT CLERMONT
AUVERGNE**

Aurillac - Clermont-Ferrand - Le Puy-en-Velay
Montluçon - Moulins - Vichy

3. Cross-Site Scripting (XSS)

Une injection SQL vise le **serveur** : le code injecté est interprété par le moteur de base de données. Un XSS vise l'**inverse**.

Définition

Un XSS permet à un attaquant de faire exécuter du code JavaScript de son choix par le **navigateur** d'un autre utilisateur, sur un site que cet utilisateur considère comme fiable.

La cause est la même famille de bug que l'injection SQL — une entrée non fiable insérée telle quelle dans une sortie — mais la sortie ici est une page HTML, et l'interpréteur est le navigateur de la victime, pas le serveur.

⇒ C'est précisément parce que le code s'exécute **chez la victime, avec les droits de la victime** qu'un site de confiance devient l'arme de l'attaque.

Le code injecté fait un aller-retour immédiat : il vient d'un paramètre de la requête et repart dans la réponse, sans jamais être stocké.

Scénario

1. La victime clique un lien envoyé par l'attaquant, contenant un paramètre piégé :
`recherche.php?q=<script>...</script>`
2. Le serveur réinjecte ce paramètre tel quel dans la page de résultats (« Vous avez cherché : ... »)
3. Le navigateur de la victime exécute le script, croyant qu'il vient du site

⇒ L'attaquant doit convaincre la victime de cliquer un lien — l'attaque ne touche que celui ou celle qui clique.

Le code injecté est enregistré côté serveur — dans un commentaire, un profil, un avis — puis renvoyé à **tous** les visiteurs de la page qui l’affiche.

Scénario

1. L’attaquant poste un commentaire contenant `<script>...</script>` sur une page publique
2. Le serveur stocke ce commentaire tel quel en base de données
3. Chaque visiteur qui charge la page reçoit et exécute le script

⇒ Pas besoin de piéger qui que ce soit individuellement : l’attaque touche tout le monde qui consulte la page infectée.

- **Échappement contextuel en sortie** — encoder `<`, `>`, `"` selon le contexte (HTML, attribut, JavaScript) au moment de l’affichage, pas seulement à la saisie
- **Content-Security-Policy** — l’en-tête `Content-Security-Policy` restreint les sources de script autorisées ; un `<script>` injecté inline peut être bloqué avant même de s’exécuter



**IUT CLERMONT
AUVERGNE**

Aurillac - Clermont-Ferrand - Le Puy-en-Velay
Montluçon - Moulins - Vichy

4. Cross-Site Request Forgery (CSRF)

Un XSS fait exécuter du code par la victime. Un CSRF n'a pas besoin d'exécuter le moindre code : il fait simplement émettre, **à l'insu de la victime**, une requête que son navigateur enverra avec ses cookies de session.

Scénario : virement bancaire

La victime est connectée sur `banque.exemple`. Elle visite ensuite, dans un autre onglet, une page contrôlée par l'attaquant contenant :

```
1 
```

Le navigateur charge cette image — donc envoie cette requête — **avec les cookies de banque.exemple**, sans que la victime ne voie rien.

⇒ Le navigateur joint les cookies à toute requête vers leur domaine d'origine, quelle que soit la page qui a déclenché la requête.

La défense de référence : le serveur exige, en plus du cookie de session, une valeur imprévisible que seule une page **légitimement servie par lui** peut connaître.

Principe

1. Le serveur génère un jeton aléatoire et le stocke côté serveur, lié à la session de l'utilisateur
2. Il insère ce jeton dans un champ caché de chaque formulaire sensible
3. À la soumission, il vérifie que le jeton reçu correspond à celui attendu pour cette session

Une page attaquante peut forcer l'envoi du cookie, mais pas celui du jeton : elle ne le connaît pas, et ne peut donc pas le fournir dans sa requête forgée.

⇒ Le jeton ne protège rien par lui-même : c'est le fait qu'il soit **vérifié côté serveur** à chaque requête sensible qui fait la différence.

Cookie SameSite

```
1 Set-Cookie: session=...;  
  SameSite=Strict
```

Le navigateur n'envoie plus ce cookie du tout lors d'une requête déclenchée depuis un autre site — la requête forgée part sans authentification.

Origin / Referer

Le serveur peut aussi vérifier que l'entête `Origin` (ou, à défaut, `Referer`) de la requête correspond bien à son propre domaine avant d'exécuter une action sensible.

⇒ Deux défenses complémentaires au jeton, pas des remplacements — un attaquant qui contrôle un sous-domaine ou un ancien navigateur peut contourner l'une ou l'autre isolément.



**IUT CLERMONT
AUVERGNE**

Aurillac - Clermont-Ferrand - Le Puy-en-Velay
Montluçon - Moulins - Vichy

5. Server-Side Request Forgery (SSRF)

Un CSRF force le **navigateur de la victime** à émettre une requête. Un SSRF force le **serveur** lui-même à en émettre une, vers une destination choisie par l'attaquant.

Scénario

Une application propose « aperçu d'une image depuis une URL » : elle télécharge côté serveur l'URL fournie par l'utilisateur pour en générer une miniature. Si aucune restriction n'est posée sur cette URL, le serveur devient un proxy que l'attaquant contrôle.

⇒ Le serveur envoie la requête depuis **son propre réseau** — avec les accès réseau qu'il a, et que l'attaquant n'a pas depuis l'extérieur.



**IUT CLERMONT
AUVERGNE**

Aurillac - Clermont-Ferrand - Le Puy-en-Velay
Montluçon - Moulins - Vichy

6. Inclusion de fichiers (LFI / RFI)

Une application qui construit un chemin de fichier à partir d'une entrée utilisateur, pour l'inclure et l'exécuter comme faisant partie de la page, ouvre une famille de vulnérabilités distincte de l'injection SQL ou du XSS.

Code vulnérable (pseudo-code PHP)

```
1 $page = $_GET['page'];  
2 include($page . '.php');
```

- **Local File Inclusion (LFI)** — inclure un fichier déjà présent sur le serveur, en dehors du répertoire prévu
- **Remote File Inclusion (RFI)** — faire inclure et exécuter un fichier hébergé sur un serveur distant, contrôlé par l'attaquant

```
1 page=../../../../etc/passwd%00
```

`../` remonte d'un niveau de répertoire à chaque occurrence, jusqu'à sortir du dossier auquel l'application était censée se limiter. Le suffixe `%00` (octet nul) servait historiquement à tronquer la chaîne avant que l'extension `.php` ne soit ajoutée par le code vulnérable.

Technique obsolète

Ce contournement par octet nul est corrigé côté PHP depuis la version 5.3.4 (2010) : les chaînes ne sont plus tronquées à l'octet nul. Il reste utile à connaître pour lire du code ancien, pas pour attaquer un système à jour.

Une LFI seule ne fait souvent que **lire** un fichier. Elle devient une exécution de code quand l'attaquant peut choisir le **contenu** du fichier inclus.

Empoisonnement de journal (log poisoning)

1. L'attaquant envoie une requête dont un en-tête (`User-Agent`, par exemple) contient du code PHP
2. Le serveur web l'écrit tel quel dans son fichier de journal (log)
3. L'attaquant utilise ensuite la LFI pour inclure ce fichier de journal — et le code PHP qu'il contient s'exécute

⇒ Une RFI saute directement cette étape : le fichier distant contrôlé par l'attaquant est du code exécutable dès l'inclusion.

Recommandations OWASP

- Ne jamais construire un chemin de fichier à partir d'une entrée utilisateur non validée
- Restreindre les valeurs possibles à une **liste blanche** de noms de fichiers autorisés, jamais à un filtrage de ../
- Désactiver `allow_url_include` et `allow_url_fopen` en PHP pour empêcher toute inclusion distante

⇒ Même défense de fond que pour l'injection SQL : ne jamais laisser une entrée utilisateur influencer la **structure** d'une opération sensible, seulement sa donnée.



**IUT CLERMONT
AUVERGNE**

Aurillac - Clermont-Ferrand - Le Puy-en-Velay
Montluçon - Moulins - Vichy

7. Authentification

Session serveur

Le cookie ne contient qu'un identifiant opaque. Le serveur garde, dans son propre stockage, qui est connecté et avec quels droits.

✓ Révocation immédiate : supprimer l'entrée côté serveur suffit.

✗ Chaque requête exige une consultation de ce stockage.

Jeton stateless (JWT)

Le jeton contient lui-même les informations d'identité, signées. Le serveur n'a rien à stocker pour le vérifier.

✓ Aucune consultation nécessaire : la signature suffit à vérifier.

✗ Invalider un jeton déjà émis est structurellement difficile.

Un JWT est trois blocs encodés en base64url, séparés par des points.

Anatomie

1 `eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJ1NDIiLCJyb2xlIjoiaXNlciJ9.<signature>`

- **Header** — `{"alg":"HS256"}` : l'algorithme de signature annoncé
- **Payload** — `{"sub":"u42","role":"user"}` : les revendications (claims)
- **Signature** — calculée sur header + payload avec une clé secrète (ou privée)

⇒ Base64url n'est pas du chiffrement : n'importe qui peut décoder header et payload sans connaître la clé. Seule la signature nécessite le secret.

Côté serveur, à chaque vérification

- **Liste blanche d'algorithmes** — n'accepter que l'algorithme attendu, jamais celui annoncé par le jeton lui-même
- Vérifier `exp` (expiration), `aud` (destinataire) et `iss` (émetteur)
- Ne jamais mettre de secret dans le payload : il est lisible en clair par quiconque intercepte le jeton

Le problème structurel reste la **révocation** : un jeton stateless reste valide jusqu'à expiration, même si le compte est désactivé entre-temps. Pallié en pratique par une expiration courte et un jeton de rafraîchissement, lui, stocké et révocable côté serveur.

Rappel — vu en détail au cours 3

PAM et TOTP (Google Authenticator et équivalents) ont déjà été présentés côté authentification locale. Le principe reste identique côté web : combiner **quelque chose que l'on sait** (mot de passe) et **quelque chose que l'on possède** (code à usage unique, clé physique).

Côté web, la seconde étape s'ajoute typiquement après une première authentification par mot de passe réussie, avant l'émission du cookie de session ou du jeton.

Déléguer l'authentification

Plutôt que chaque site gère ses propres mots de passe, l'utilisateur s'authentifie une fois auprès d'un fournisseur d'identité (Google, Microsoft, l'établissement...), qui délivre un jeton attestant de son identité aux applications qui lui font confiance.

- **OAuth2** — cadre d'autorisation : délègue un accès limité à une ressource (« cette application peut lire mon agenda »)
- **OpenID Connect (OIDC)** — ajoute une couche d'authentification au-dessus d'OAuth2 (« voici qui est cet utilisateur »)
- **SSO (Single Sign-On)** — un seul login pour plusieurs applications qui partagent le même fournisseur d'identité



**IUT CLERMONT
AUVERGNE**

Aurillac - Clermont-Ferrand - Le Puy-en-Velay
Montluçon - Moulins - Vichy

8. Reconnaissance et test d'intrusion

Article 323-1 du Code pénal

Accéder ou se maintenir frauduleusement dans un système de traitement automatisé de données (STAD) est punissable, **y compris sans dommage causé** et **y compris si le système était mal protégé**. L'absence de protection n'est jamais une autorisation.

- Une autorisation **écrite**, explicite, et **bornée à un périmètre précis** (adresses, dates, actions permises) est le seul fait qui rend un test d'intrusion légal
- En dehors de ce périmètre — y compris « juste pour vérifier » — on repasse sous le coup de l'article précédent

⇒ En TP, seules les machines de la salle et les cibles explicitement désignées sont dans le périmètre autorisé.

Rappel — vu au cours 2

Shodan indexe les bannières de services exposés sur Internet (serveur web, version SSH, caméra mal configurée...) sans jamais interroger lui-même la cible : c'est de la reconnaissance **passive**, invisible pour la cible tant qu'on ne fait qu'y consulter des résultats déjà indexés.

L'OSINT (open-source intelligence) élargit ce principe à toute source publique — réseaux sociaux, dépôts de code, moteurs de recherche — pour cartographier une cible avant la moindre interaction directe.

Contrairement à l'OSINT, nmap interroge directement la cible : on quitte la reconnaissance passive.

Quelques types de scans

- `nmap -sn <réseau>` — balayage ping : qui répond, sans toucher aux ports
- `nmap -sS <cible>` — scan SYN : ports TCP ouverts, sans achever la connexion (« furtif »)
- `nmap -sU <cible>` — scan UDP, plus lent, souvent moins fiable
- `nmap -O / -sV` — détection du système d'exploitation et des versions de service, à partir des réponses caractéristiques obtenues

⇒ Chaque type de scan est déjà, légalement, une tentative d'accès : le cadre légal de la slide précédente s'applique dès la première commande.

`ssh-audit` se connecte à un serveur SSH et évalue sa configuration, sans tenter la moindre authentification.

Ce qu'il signale

- Algorithmes d'échange de clé, de chiffrement et de MAC obsolètes (encore acceptés par le serveur pour rester compatible avec de vieux clients)
- Version du serveur SSH, souvent corrélable à des CVE connues
- Recommandations de durcissement concrètes (algorithmes à désactiver)

⇒ Un serveur SSH accessible n'est pas la vulnérabilité — un serveur SSH mal configuré, oui.

L'exécution de code à distance (RCE) n'est pas une famille de vulnérabilité de plus à côté de SQLi, XSS ou LFI : c'est souvent **ce que ces familles permettent d'atteindre** en bout de chaîne.

Exemples de chaînes

- LFI + fichier de log empoisonné → RCE (§ LFI/RFI)
- Débordement de tampon sur un service réseau → RCE (cours 3)
- Désérialisation non sécurisée d'un objet fourni par le client → RCE

⇒ Se demander « quelle vulnérabilité initiale a rendu cette exécution de code possible » est plus utile que de classer la RCE comme un type d'attaque isolé.

Metasploit

Un framework d'exploitation qui rassemble des exploits connus (souvent liés à des CVE précises), des modules post-exploitation et un environnement pour les enchaîner sur une cible autorisée.

`msfvenom` en génère la partie « charge utile » (payload) : un programme, ou un greffon inséré dans un exécutable existant, qui ouvre une connexion vers l'attaquant une fois exécuté sur la machine cible.

⇒ Cette même technique — greffer une charge utile dans un APK légitime — est le point de départ classique d'une application Android trojanisée, section suivante.

CVE

Un identifiant public et unique (ex. CVE-2024-12345) attribué à une vulnérabilité précise, dans un produit précis, pour permettre d'en parler sans ambiguïté d'une base de données à l'autre.

CVSS

Un score de 0 à 10 estimant la gravité d'une CVE (vecteur d'attaque, complexité, privilèges requis, impact). Un score élevé priorise le correctif, il ne dit rien sur la probabilité réelle d'exploitation.

⇒ Consultables sur des bases publiques comme le NVD (National Vulnerability Database).



**IUT CLERMONT
AUVERGNE**

Aurillac - Clermont-Ferrand - Le Puy-en-Velay
Montluçon - Moulins - Vichy

9. Sécurité Android

Android construit son isolation applicative directement sur les mécanismes Linux déjà vus au cours 3, plutôt que de réinventer un modèle propre.

Un UID Linux par application

Chaque application installée reçoit son propre identifiant utilisateur Linux. Deux applications ne peuvent pas lire les fichiers l'une de l'autre, pour la même raison que deux comptes Unix classiques ne le peuvent pas.

Depuis Android 5, **SELinux** fonctionne en mode **enforcing** système entier (cours 3, §MAC) : même un processus qui parviendrait à obtenir les droits root reste contraint par la politique SELinux, qui limite ce qu'un tel accès permet réellement de faire.

Chaîne de build

- 1 code source (Java / Kotlin)
- 2 -> compilation en bytecode Dalvik
- 3 -> classes.dex, empaqueté dans l'APK (une archive zip)
- 4 -> exécuté sur l'appareil par l'ART (Android RunTime)

Un APK est donc littéralement un fichier zip : `AndroidManifest.xml` (permissions, composants déclarés), `classes.dex` (le code compilé), `resources.arsc` et le dossier `res/` (ressources), `META-INF/` (signature).

⇒ Cette structure explique pourquoi un APK s'inspecte déjà en partie avec de simples outils d'archive, avant tout outil spécialisé.

Avant Android 6

Toutes les permissions déclarées dans le manifeste sont accordées en bloc à l'installation. L'utilisateur accepte tout, ou n'installe pas.

Depuis Android 6 / API 23

Les permissions **dangereuses** (caméra, contacts, SMS, localisation...) sont demandées individuellement, au moment où l'application les utilise réellement pour la première fois.

⇒ L'utilisateur peut accorder l'accès aux contacts sans jamais accorder celui à la localisation — un contrôle bien plus fin qu'un tout-ou-rien à l'installation.

- **Signature d'application** — chaque APK est signé avec le certificat de son développeur ; une mise à jour n'est acceptée que si elle est signée avec le **même** certificat que la version déjà installée
- **Verified Boot** — vérifie, à chaque démarrage, que le système lui-même n'a pas été altéré depuis sa version signée par le fabricant
- **Play Protect** — analyse les applications installées (y compris hors du Play Store) à la recherche de comportements malveillants connus

⇒ Trois vérifications indépendantes, à trois moments différents : qui a écrit le code, qui a écrit le système, et ce que le code fait une fois installé.

Une application Android communique avec les autres par **intents**. Un composant **exporté** accepte ces messages venant de n'importe quelle autre application.

Manifeste vulnérable (extrait)

```
1 <activity android:name=".AfficherProfil"  
2         android:exported="true">  
3     <!-- pas de android:permission -->  
4 </activity>
```

N'importe quelle application installée, sans aucune permission particulière, peut lancer `AfficherProfil` directement.

⇒ C'est la classe de vulnérabilité Android la plus fréquente : un composant exporté sans contrôle d'accès expose directement une fonctionnalité interne à toute application tierce installée.

Boîte à outils

- `aapt` — lit les métadonnées d'un APK (nom de package, permissions) sans le décompresser entièrement
- `apktool` — désassemble les ressources et le bytecode en un format lisible et ré-empaquetable
- `jadx` / `dex2jar` — décompilent le bytecode Dalvik vers du code source **Java** lisible

Le code source original peut avoir été écrit en Kotlin : `jadx` régénère quand même du Java, car Kotlin et Java compilent tous deux vers le **même** bytecode Dalvik — l'information « langage d'origine » ne survit pas à la compilation.

Principe

`msfvenom` (section précédente) peut greffer une charge utile dans un APK légitime déjà signé par un vrai développeur, sans en casser le fonctionnement apparent : l'application continue de sembler fonctionner normalement.

Le vecteur d'installation n'est alors plus une vulnérabilité technique mais de l'ingénierie sociale : convaincre la victime d'installer cet APK modifié en dehors du Play Store, en le faisant passer pour légitime.

⇒ La revue de permissions demandées (§ précédente) est justement la première ligne de défense contre ce scénario.

Chaque application stocke ses données dans un répertoire isolé par le même mécanisme d'UID que celui vu en début de section.

Où chercher

- `/data/data/<package>/` — répertoire privé de l'application (bases de données, préférences, fichiers internes)
- Bases SQLite fréquentes pour les données structurées (messages, contacts, historique) — inspectables avec `sqlite3` une fois extraites
- `/data/system/packages.xml` — registre système : liste des applications installées, leurs permissions accordées, leur date d'installation

⇒ C'est exactement cette isolation par répertoire qui transforme une analyse forensique de téléphone en une fouille systématique application par application — section suivante.



**IUT CLERMONT
AUVERGNE**

Aurillac - Clermont-Ferrand - Le Puy-en-Velay
Montluçon - Moulins - Vichy

10. Investigation numérique (forensic)

Ordre de volatilité (RFC 3227)

- 1 registres / cache CPU -- disparaît en microsecondes
- 2 mémoire vive (RAM) -- perdue à l'extinction
- 3 état des connexions réseau
- 4 disque -- survit à un redémarrage
- 5 sauvegardes / archives -- la plus stable

Toujours collecter dans cet ordre : capturer le disque en premier alors que la RAM contenait encore des preuves perdues, c'est perdre ces preuves.

La **chaîne de possession** documente qui a manipulé chaque preuve, quand, et comment — indispensable si l'analyse doit un jour servir en justice.

Avant toute analyse, on fige un état du système sous une forme qui ne changera plus.

Principe

- Copie bit à bit du support (image disque), jamais une analyse sur le système original en fonctionnement
- Empreinte cryptographique (hash) de l'image, calculée immédiatement et reconsignée : prouve que l'image n'a pas été altérée depuis
- Toute analyse se fait ensuite sur une **copie** de l'image, l'original restant intact comme référence

Dead

Système éteint. Le support est extrait et analysé hors ligne — la méthode la plus sûre, rien ne peut plus changer sous l'analyste.

Live

Système encore allumé, analysé en fonctionnement — on capture de la RAM, des connexions actives, qui disparaîtraient à l'extinction.

Realtime

Surveillance continue pendant l'incident lui-même, pas après coup — détecter et documenter l'attaque à mesure qu'elle se déroule.

Wireshark permet de filtrer une capture pour isoler le trafic pertinent parmi des milliers de paquets.

Filtres de base

```
1 ip.addr == 10.0.0.5 && http      # tout le HTTP vers/depuis une machine
2 http.request.method == "POST"    # uniquement les soumissions de formulaire
```

Un grand nombre de requêtes `POST` consécutives vers la même page d'authentification, avec des identifiants différents à chaque fois, est la signature réseau caractéristique d'un bruteforce de mot de passe.

De nombreux formats de stockage applicatif (Android, navigateurs, messagerie locale) sont en réalité des bases SQLite – un simple fichier, interrogeable sans serveur de base de données.

En ligne de commande

```
1  sqlite3 fichier.db
2  sqlite> .tables
3  sqlite> PRAGMA table_info(nom_de_table);
4  sqlite> SELECT * FROM nom_de_table WHERE ...;
```

⇒ `.tables` et `PRAGMA table_info` répondent à la même question qu'un `\d` en PostgreSQL ou un `DESCRIBE` en MySQL : quelle est la structure de ce que je regarde, avant de savoir quoi y chercher.

Les bases de données système d'Android (SMS, calendrier, journaux d'appels) stockent très généralement leurs dates en **millisecondes** depuis l'époque Unix, pas en secondes.

Erreur fréquente

Un convertisseur de timestamp qui attend des secondes, appliqué directement à une valeur en millisecondes, place l'événement des décennies dans le futur ou dans le passé. Toujours vérifier l'unité avant de convertir — ou diviser par 1000 si le convertisseur utilisé attend des secondes.

1. **Acquisition** — figer une copie intègre, horodatée, avant toute analyse
2. **Investigation** — reconstruire les faits à partir des preuves collectées
3. **Remédiation** — refermer la vulnérabilité qui a permis l'incident
4. **Rapport** — documenter la chronologie et les preuves pour les parties prenantes

⇒ Une investigation qui s'arrête à l'étape 2 explique ce qui s'est passé, sans empêcher que cela se reproduise.



**IUT CLERMONT
AUVERGNE**

Aurillac - Clermont-Ferrand - Le Puy-en-Velay
Montluçon - Moulins - Vichy

11. Conclusion

1. Injecter du code dans une requête vers la base de données § *SQLi*
2. Injecter du code exécuté par le navigateur d'un autre utilisateur § *XSS*
3. Forger une requête au nom d'un utilisateur authentifié § *CSRF*
4. Faire exécuter au serveur un fichier qu'il n'aurait pas dû lire § *LFI/RFI*
5. Casser ou contourner l'authentification elle-même § *Auth*

Reconnaissance, RCE, Android et forensic prolongent cette chaîne côté attaquant (comment on trouve et on exploite) et côté défenseur (comment on reconstitue, après coup, ce qui s'est passé).

⇒ Comme au cours 3 : identifier **où** s'exécute le code et **ce que permet** chaque faille compte plus que mémoriser des noms de catégories.

TP4 — Mobile et forensic

Analyse post-mortem d'une trace réseau et d'un système de fichiers Android, rétro-ingénierie d'un APK : § Android et § Forensic de ce cours, appliqués sur un cas complet.

TP5 — Attaques web

XSS, LFI/RFI, injection SQL avec `sqlmap`, CSRF : § Injection SQL à § CSRF de ce cours, à exploiter puis à corriger.

Questions ?