Web and Mobile Application Security

Chapter 2 — Web Security Model

Mr Laïdi FOUGHALI

l.foughali@univ-skikda.dz

(Website ⇒ www.al-moualime.com)

University of Skikda — Computer Science Department

2nd Year Master in Information Security (IS)

Academic Year 2026-2027

Updated on 2026-09-22 at 23:41:02

Outline

Chapter objectives

By the end of this chapter, you will be able to

- Explain why the browser behaves like an operating system
- Describe permission-based access control (consent, secure context)
- Decide whether two URLs share the same origin and state what the SOP allows or forbids
- Identify the limits of the SOP (cookies, special origins, authentication)
- Use postMessage and CORS correctly to communicate across origins

2.1 — The browser, an operating system

A role similar to an operating system

The browser constantly runs code coming from potentially hostile sites (HTML, CSS, JavaScript), on a simple click. It must:

- Manage "processes" (tabs, frames)
- Allocate resources and control access to the network, storage and devices
- Isolate applications that do not trust one another

A key difference

Unlike a classic OS (unified user/group model), **the Web has no global, coherent security model**: it relies on an accumulation of partial mechanisms, added as it evolved.

Example: Chrome (*Site Isolation*) and Firefox (*Fission*) place each **site** in a separate, restricted process (*sandbox*), just as an OS isolates programs.

2.2 — Permission-based access control

Sensitive capabilities subject to consent

Geolocation, camera and microphone, notifications, clipboard, persistent storage. . .

The browser as a reference monitor

- It intercepts each request, shows a prompt and grants access only after **explicit consent**, revocable at any time
- These APIs are only available in a **secure context** (HTTPS or `localhost`)
- Active content runs in a **sandbox**: no direct access to the disk or to system processes

On the server side, the `Permissions-Policy` header restricts the capabilities of a page and its iframes, e.g. `Permissions-Policy: camera=(), geolocation=(self)`.

2.3 — The Same-Origin Policy (SOP)

Cornerstone of Web isolation (Netscape 2, 1995)

An **origin** = **protocol** + **host** + **port**. Two origins are identical only if all three match **exactly**.

Same origin as `https://exemple.dz/`?

`https://exemple.dz/page2`	**Yes**	only the path differs
`http://exemple.dz/`	No	different protocol
`https://exemple.dz:8443/`	No	different port (implicit 443)
`https://www.exemple.dz/`	No	different host (subdomain)

2.3 — What the SOP allows and what it forbids

Across different origins, a page can. . .

- **Write** (send): submit a form, follow a link, redirect ⇒ **allowed**
- **Embed**: `<img>`, `<script>`, style sheets, `<iframe>` ⇒ **allowed**
- **Read**: the DOM of another frame, the response of a `fetch`/`XMLHttpRequest` ⇒ **forbidden**

Consequence

The SOP protects the **reading** of responses, not the **sending** of requests: a malicious site can still make the browser issue an authenticated request (with the victim's cookies) to another site. This is the basis of **CSRF** (chapter 3).

2.3 — Limits of the SOP

A partial, unsynchronized policy

- **Not tied to authentication or TLS**: two windows remain of the same origin if the user switches accounts or if the certificate becomes invalid
- **Cookies**: their scope is based on the domain and path, not the origin — they **ignore the port** and, without the `Secure` attribute, the protocol
- **Special origins**: `about:blank` inherits the origin of its creator; `data:` and `sandbox` iframes have an **opaque** origin (`null`)

To avoid

Never trust the `null` origin (e.g. `Access-Control-Allow-Origin: null`): any site can produce it with a `sandbox` iframe.

2.3 — Controlled communication: postMessage

Exchanging messages between windows and frames of different origins

- **Sender**: `target.postMessage(data, "https://partner.dz")` — always specify the target origin, **never** `"*"` if the message is sensitive
- **Receiver**: in the `message` event handler, **check** `event.origin` before processing `event.data`

A common mistake

A receiver that does not check `event.origin` accepts messages from **any site**: data injection, token theft, even XSS if the content is inserted into the DOM.

2.3 — Controlled communication: CORS

Cross-Origin Resource Sharing: the server explicitly allows reading

- The server answers `Access-Control-Allow-Origin: https://partner.dz`; the browser hands the response to the script only if the origin is allowed
- Non-"simple" requests (`PUT`/`DELETE` methods, JSON, specific headers): a **preflight** `OPTIONS` request is sent before the actual request
- With cookies: `Access-Control-Allow-Credentials: true` + a **specific** origin (the `*` wildcard is then rejected by the browser)

A classic misconfiguration

Echoing the received `Origin` header back in `Access-Control-Allow-Origin`, without an allow-list, together with `Allow-Credentials: true`: **any site** can then read the logged-in user's data.

Key points to remember

The essentials of the chapter

- The browser runs untrusted code: it works like an OS (isolated processes, permissions, sandbox)
- Sensitive capabilities require consent and a secure context (HTTPS)
- Origin = protocol + host + port; the SOP forbids cross-origin **reading**, not the **sending** of requests
- The SOP is partial: cookies, opaque origins, no link with authentication
- postMessage (target origin + checking `event.origin`) and CORS (server-side allow-list) are the controlled channels

Questions and exercises

Practice

1. Are two documents `http://site.dz:80` and `https://site.dz:443` of the same origin? Justify.
2. Explain how the browser can be compared to an operating system.
3. Does the SOP prevent a malicious site from sending an authenticated request to your bank? What does it actually prevent?
4. Why is a cookie set by `https://site.dz:8443` sent to `https://site.dz`?
5. An API returns the copied `Origin` header and `Access-Control-Allow-Credentials: true`. What is the risk and how can it be fixed?
6. Which check must a `postMessage` receiver perform before processing a message?

Chapter sources

- Official module syllabus (UEF322, chapter 2)
- Conceptual framework (browser as an OS, SOP and its limits) informed by M. Zalewski, *The Tangled Web* (2011)
- Standard technical documentation (MDN Web Docs) for CORS, postMessage and Permissions-Policy