Client side web security

Sicurezza (CT0539) 2025-26
Università Ca' Foscari Venezia

Riccardo Focardi
www.unive.it/data/persone/5590470
secgroup.dais.unive.it

Università
Ca'Foscari
Venezia

Web (in)security

Web applications are complex and offer an incredibly **wide** attack surface

- attacks directly targeting the **server-side code** or **databases** (see previous classes)
- attacks running in the **browser**
- attacks on the **network**

Web sessions

Web applications usually have a state

Example:

1. user **logs** into a web application
2. a session is started (**state** changes)
3. user gets access to data and resources (**authorization**)
4. web pages are customized based on the **user**

When the user browses to different web application pages, the **session** needs to be preserved

⇒ The user shouldn't log in again!

The session needs to be represented in the **browser**:

- a **session token** that works as a *"session password"*

Session token

The session token can be **stored** in various ways:

Browser cookie: it is automatically attached to any subsequent request to the server

URL parameter: in links to pages

Hidden form field: sent when forms are submitted

Note: if a session token is **guessed** or **leaked**, the session can be hijacked, and the user impersonated

⇒ token should be **unguessable** and **kept confidential**

Cookie theft is a typical web attack that can be used to hijack a session

Which token?

URL parameters are **exposed** in logs and referrers

⇒ bad for **security**!

Hidden form fields are only **visible** when forms are submitted

⇒ bad for **usability**: web session should be represented in any web page, not just forms

⇒ The standard approach is to use a **browser *session cookie***

It is automatically attached to <u>any</u> request and form submission

Note: combining different tokens may offer resistance to **session integrity attacks**, e.g. CSRF as we will see in next class

Cookies and cookie policy

A cookies is set using the HTTP header `Set-cookie` with the following fields:

```
NAME     = VALUE;
domain   = (es .unive.it)
path     = (es /teaching)
expires  = (when expires)
secure   = (boolean flag)
HttpOnly = (boolean flag)
```

The browser **automatically attaches** to a web request cookies such that:

- domain is a **suffix** of the URL domain
- path is a **prefix** of URL path
- protocol is **HTTPS** if cookie is flagged `secure`

The `Set-cookie` header can occur multiple times to set more cookies

Example

A cookie with

- domain `.unive.it`
- path `/teaching`

will be attached to a GET request to URL
`https://secgroup.dais.unive.it/teaching/security-course`

- `.unive.it` is a suffix of `secgroup.dais.unive.it`
- `/teaching` is prefix of `/teaching/security-course`

Example: cookie creation

Example: **creation** of two cookies with the same name and different paths from the browser javascript console (URL with path=`/search`, Try it in incognito!)

```
> document.cookie
""

> document.cookie = "username=test; path=/search"
"username=test; path=/search"

> document.cookie = "username=test1; path=/"
"username=test1; path=/"

> document.cookie
"username=test; username=test1"
```

`domain` and `path` are set, by default, to the host and path in the URL

Example: cookie deletion

Deletion by setting a **date in the past**

Each cookie is deleted separately by the **path**. When not specified the current one is applied (e.g. `/search`)

```
> document.cookie = "username=; expires=Thu, 01 Jan 1970 00:00:00 UTC"
"username=; expires=Thu, 01 Jan 1970 00:00:00 UTC"

> document.cookie
"username=test1"

> document.cookie = "username=; expires=Thu, 01 Jan 1970 00:00:00 UTC; path=/"
"username=; expires=Thu, 01 Jan 1970 00:00:00 UTC; path=/"

> document.cookie
""
```

Two cookies with the same name … really?

If paths are not disjoint they are **both sent** to the server

Which one will be used?

In a 2015 paper [ZJL15] authors show that equal cookies are treated differently depending on the language, framework and library

⇒ not good for security!

Java, JavaScript and Go read cookies as a **list**

PHP, Python, ASP, ASP.NET, Node.js, JQuery, … only provide a **dictionary** (only one of the two cookies, which one? Language-dependent!)

Note: only **name** and **value** are sent. The server cannot discriminate based on the path!

Cookie flags

```
NAME     = VALUE
domain   = (es .unive.it)
path     = (es /teaching)
expires  = (when expires)
secure   = (boolean flag)
HttpOnly = (boolean flag)
```

`Secure` cookies and mixed content

HTTPS requires more resources than HTTP because of cryptography

Web applications sometimes have **mixed HTTP/HTTPS content**

⇒ this can **expose** session cookies!

Even if the login is HTTPS, any access to HTTP pages might send the <u>session cookie in the clear</u>

The `secure` flag **prevents** that the flagged cookie is sent over HTTP connections

IDEA: set **two session cookies**, a `secure` and a non-`secure` one for HTTPS and HTTP pages

⇒ The attacker can only hijack the HTTP, **non-sensitive** part

What about cookie integrity?

The secure flag was **not** designed for integrity

- In older browsers secure cookies could be set even over HTTP

A network attacker might set a **secure cookie of her choice** by mounting a *Man-In-The-Middle* (MITM) attack

Is this problematic for security?

⇒ User's data are leaked to the **attacker's account** when **submitted** to the web application!

In recent browsers secure cookies can only be set **over HTTPS**

⇒ Attacker cannot **overwrite** existing secure cookies from HTTP

Session fixation attack

Is this enough?

1. Attacker sets a (non secure) cookie value into a victim's browser (e.g. through a MITM over HTTP)
2. The user **authenticates**
3. Attacker's cookie is "**promoted**" to session cookie

⇒ the attacker **hijacks the session** (cookie is **known**!)

Realistic! It is often the case that cookies are set before authentication in a so-called **pre-session**

Solution: in case session is started before authentication, always **refresh** the token when user authenticates

Cookie flags

```
NAME     = VALUE
domain   = (es .unive.it)
path     = (es /teaching)
expires  = (when expires)
secure   = (boolean flag)
HttpOnly = (boolean flag)
```

`HttpOnly` cookies

Web pages execute **JavaScript** code in the browser

JavaScript can **get** and **set** cookies

A malicious JavaScript injected into a page might **leak cookies** (Cross Site Scripting, XSS, next class)

⇒ An attack in a single page would compromise the **whole session**

The `HttpOnly` flag prevents that JavaScript accesses the flagged cookie

⇒ **Prevent cookie leaks** by malicious JavaScript code

Session cookies should **always be flagged** as `HttpOnly`

`HttpOnly` cookies are sent to the server but are **invisible** to JavaScript

Stateful vs. stateless server

The session state can be either stored in the server or in the client (or a mix of the two)

Stateful server: have a `Secure` and `HttpOnly` session cookie in the browser and all the state information on the server

⇒ Can produce excessive **server side overhead**

Stateless server:

1. **encrypt** the session data together with a user ID and a timestamp using a server key
2. store the **encrypted blob** in a cookie in the browser
3. the server stores the time the user logged-in or out so to check the **validity** of the encrypted blob

The Same Origin Policy

Same Origin Policy (SOP)

Browsers access many different applications at the same time

Same Origin Policy (SOP) is a standard browser policy that **restricts access** among documents or scripts loaded from different domains

It provides a simple but necessary **isolation** between web applications running in the same browser

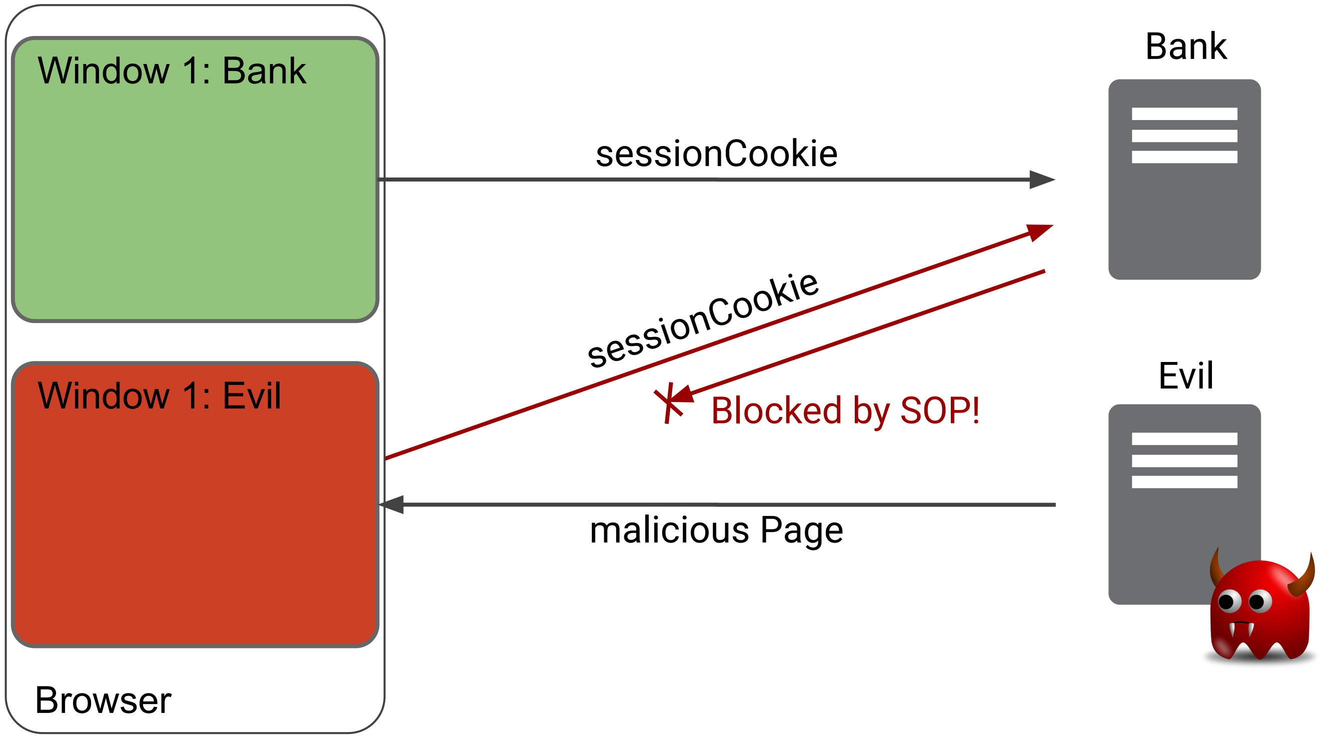
Example: Alice is browsing her home banking web app B and opens a web site E that sends requests towards B

⇒ The cookie is attached and E exfiltrates **sensitive data** from B!

Without SOP, a malicious site would hijack any other open session!

(see, e.g., mozilla page on SOP)

SOP prevents cross-site leakage



Origin

Two pages have the same origin if the **protocol**, **port**, and **host** are the same for both pages

Example: `http://store.company.com/dir/page.html`

```
http://store.company.com/dir2/other.html   OK
http://store.company.com/dir/in/pag.html   OK
https://store.company.com/secure.html      NO different protocol
http://store.company.com:81/dir/etc.html   NO different port
http://news.company.com/dir/other.html     NO different host
```

Scope of SOP

SOP affects:

- Network access
- Script APIs
- Data storage
- Cookies

If **cross-origin**, access is **restricted** or **forbidden**

SOP network access

Cross-origin writes are typically **allowed**

Example: following a link, redirection and submitting a form

The reached page is **different** from the originating one (no risk of leaking information to the originating page)

⇒ SOP protect **confidentiality** and not integrity!

Cross-origin embedding is typically **allowed**

Examples: images, CSS and JavaScript

Cross-origin reads are typically **not allowed**

Example: responses to cross-origin AJAX requests

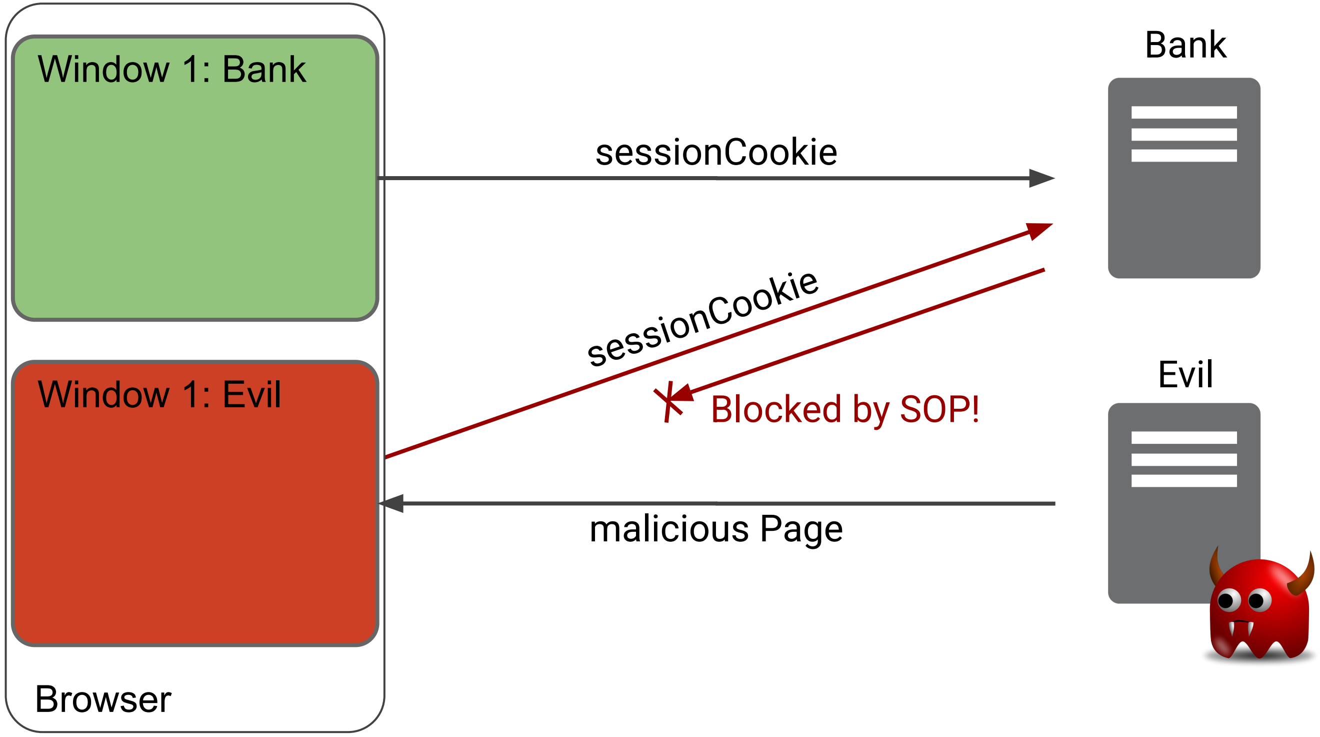
Example: AJAX

```
var xmlHttp = new XMLHttpRequest();
xmlHttp.open( "GET", "https://www.google.it");
xmlHttp.send( null );
```

```
Access to XMLHttpRequest at 'https://www.google.it/' from origin
'https://www.unive.it' has been blocked by CORS policy: No
'Access-Control-Allow-Origin' header is present on the requested
resource.
```

Note: request is sent, response is rejected!

SOP prevents cross-site leakage



Script APIs

Some JavaScript APIs allow documents to **reference each other**

When two documents do not have the same origin, only a **limited access** is provided

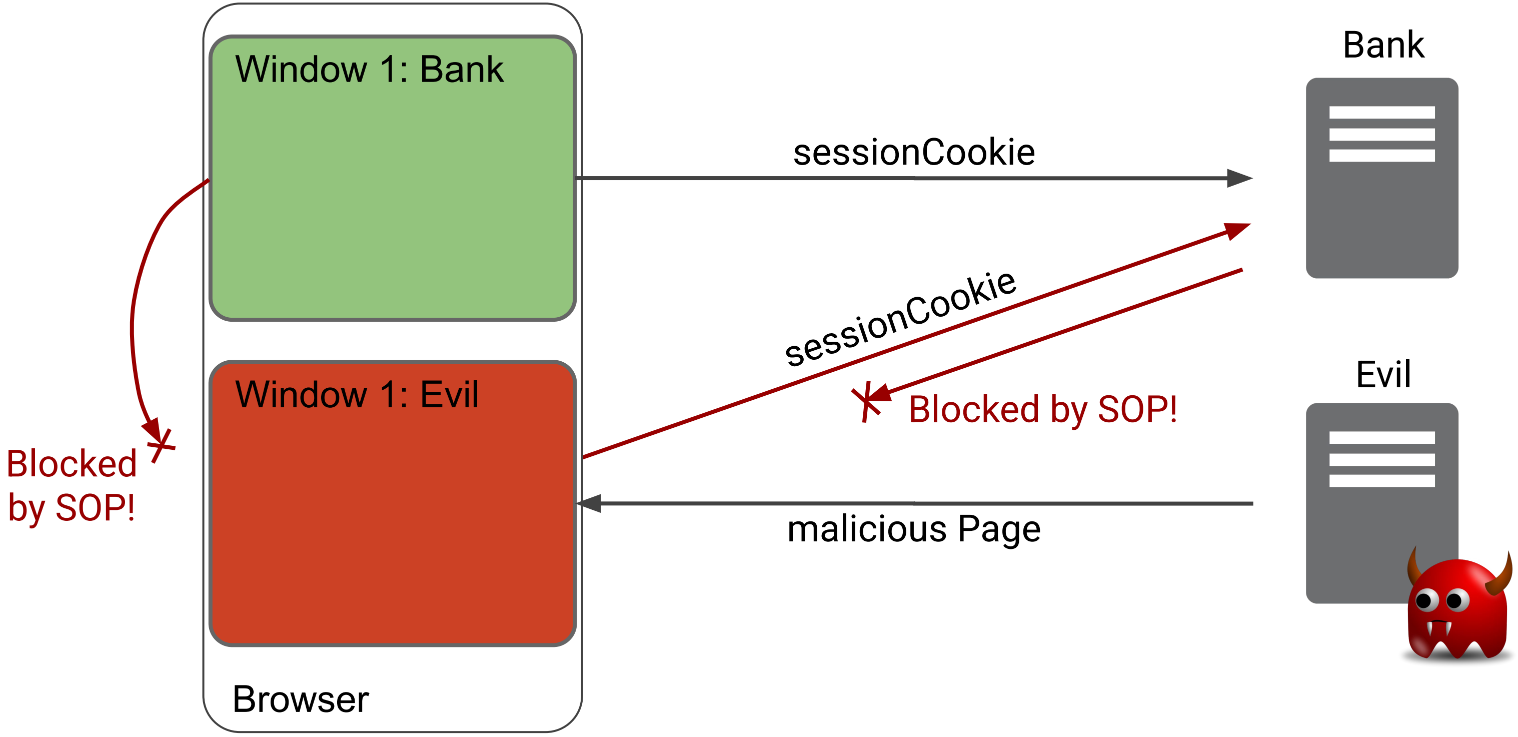
Example 1: `window.document` gives access to the whole document of a window. Cross-origin access is forbidden

Example 2: `location.href` is the entire URL which might contain sensitive data. Cross-origin access is forbidden

This restriction can be **relaxed** by changing `document.domain`

⇒ useful when web pages belonging to different subdomains **need to communicate**

SOP prevents cross-site leakage



Changing origin

The origin can be set to the **current** domain or to a **superdomain** (a suffix) of the current domain (not a top-level domain)

⇒ useful when SOP blocks API access in the same web application

```
> document.domain
"www.unive.it"

> document.domain = "unive.it"
"unive.it"

> document.domain = "www.unive.it"
"www.unive.it"
```

Changing origin (ctd.)

```
> document.domain = "idp.unive.it"
VM777:1 Uncaught DOMException: Failed to set the 'domain' property
on 'Document': 'idp.unive.it' is not a suffix of 'unive.it'.

> document.domain = "it"
VM792:1 Uncaught DOMException: Failed to set the 'domain' property
on 'Document': 'it' is a top-level domain.
```

NOTE: deprecated in chrome as it relaxes SOP too much.

Storage and cookies

Storage is separated by origin: each origin has its own storage

We defined **origin** as the triplet

`protocol,host,port`

Examples: Web Storage and IndexedDB

For **cookies**, protocol is optional and the path is considered instead of the port. The **origin** for a cookie is

`[protocol],host,path`

NOTE: the restriction on path is for performance and not for security

Using it for security can be risky as SOP **does not prevent** pages under different paths to access **each other**

SOP for reading/writing cookies

We have already seen that browser sends cookies such that:

- cookie domain is a **suffix** of the URL domain
- cookie path is a **prefix** of URL path
- protocol is **HTTPS** if cookie is flagged **secure**

`domain` can be set to any suffix of URL-hostname except top-level domains

For example, `.unive.it` will specify a cookie that applies to any subdomain of `unive.it`

`path` can be set to any prefix of the current path