Client-side web security

Security 1 2018-19

Università Ca' Foscari Venezia

www.dais.unive.it/~focardi

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**
- attacks running in the **browser**
- attacks on the **network**

Sessions

Web applications usually have a state

Example:

- user logs into a web account
- a session is started (state changes)
- user gets access to her data and resources

The state needs to be represented in the browser

⇒ usually done by storing a freshly generated **session token** that works as a "*session password*"

Session token

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

- As a browser cookie, that is attached to any subsequent request to the server
- As a URL parameter in links
- As a hidden form field

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**

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**

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

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

Cookies

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)
```

Cookie policy

The browser **automatically attaches** to a web request all 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`

Example

A cookie with

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

will be sent on a GET 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`

Creating and deleting a cookie

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

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

A cookie can be deleted by setting expiration in the past

Example: cookie creation

The following example shows the creation of two cookies with the same name and different paths

```
> 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"
```

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 (`/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?

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

Which one will be used?

Language/framework/library-dependent [ZJL15]

- 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!

Demo

Inspect how cookies are set and sent in the browser

Cookie flags

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

Secure cookies

A typical situation that exposes session cookies is when a site has **mixed HTTP/HTTPS content**

- Even if the login is HTTPS, any access to HTTP pages might send the session cookie in the clear

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

What about 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 his choice** by mounting a MITM attack

⇒ user sends sensitive data into the attacker's account!

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

Session fixation

Is this enough?

1. Attacker sets a 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: 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

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

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

⇒ **Prevent cookie leaks** by XSS

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

Stateful vs. stateless

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

⇒ Can produce excessive **server-side overhead**

Stateless:

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

Same Origin Policy (SOP)

A standard browser policy that **restricts access** among documents or scripts loaded from different domains

Without SOP, browsing on a malicious site will allow it to access other open pages and hijack any open session!

SOP provides a simple, necessary form of **isolation** between web applications running in the same browser (see, e.g., mozilla page on SOP)

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

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**
Es. 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)

Cross-origin embedding is typically **allowed**
Examples are images, CCS and JavaScript;

Cross-origin reads are typically **not allowed**
Es. 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!

Script APIs

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

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

- **`window`**: a window containing a DOM document
  E.g. `window.document` refers to the document
- **`location`**: the URL of the object it is linked to
  E.g. `location.href` is the entire URL

⇒ can be **relaxed** by changing `document.domain`

Changing origin

The origin can be set to the **current** domain or to a **superdomain** (a suffix) of the current domain

⇒ useful when web pages belonging to different sub-domains need to communicate

```
> 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.
```

Storage and cookies

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

We defined **origin** as the triplet

`protocol`, **`host`**, **`port`**

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

[**`protocol`**], **`host`**, **`path`**

SOP for reading cookies

We have 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**

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

⇒ SOP **does not prevent** pages under different paths of the same domain to access each other DOM

SOP for writing cookies

`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

References

[ZJL15] Xiaofeng Zheng, Jian Jiang, Jinjin Liang, Hai-Xin Duan, Shuo Chen, Tao Wan, Nicholas Weaver:

Cookies Lack Integrity: Real-World Implications. USENIX Security Symposium 2015: 707-721