Client-Side Security

Riccardo BONAFEDE
Università di Padova

https://cybersecnatlab.it

License & Disclaimer

License Information

Disclaimer

- We disclaim any warranties or representations as to the accuracy or completeness of this material.
- Materials are provided “as is” without warranty of any kind, either express or implied, including without limitation, warranties of merchantability, fitness for a particular purpose, and non-infringement.
- Under no circumstances shall we be liable for any loss, damage, liability or expense incurred or suffered which is claimed to have resulted from use of this material.

Goal

- Understand fundamentals of Client-Side Security and of Same-Origin Policy
- Learn how to spot and exploit a Cross-Site Scripting vulnerability
- Learn how to mitigate Client-Side Vulnerabilities

Prerequisites

- Lecture:
  - *WS_1.1 - HTTP Protocol and Web-Security Overview*
- Basic knowledge about HTML and JavaScript

Outline

- Client-Side Security
  - Resources
  - SOP – Same Origin Policy
- XSS – Cross-Site Scripting
- CSRF – Cross-Site Request Forgery

Outline

- Client-Side Security
  - Resources
  - SOP – Same Origin Policy
- XSS – Cross-Site Scripting
- CSRF – Cross-Site Request Forgery

Overview

- In these slides we will talk about client-side security
- Client-side Security is a branch of web security whose goal is to secure web pages when they run inside a browser

Web Pages

- Web pages are written using a special language, called HTML
  - It works using tags, for example:

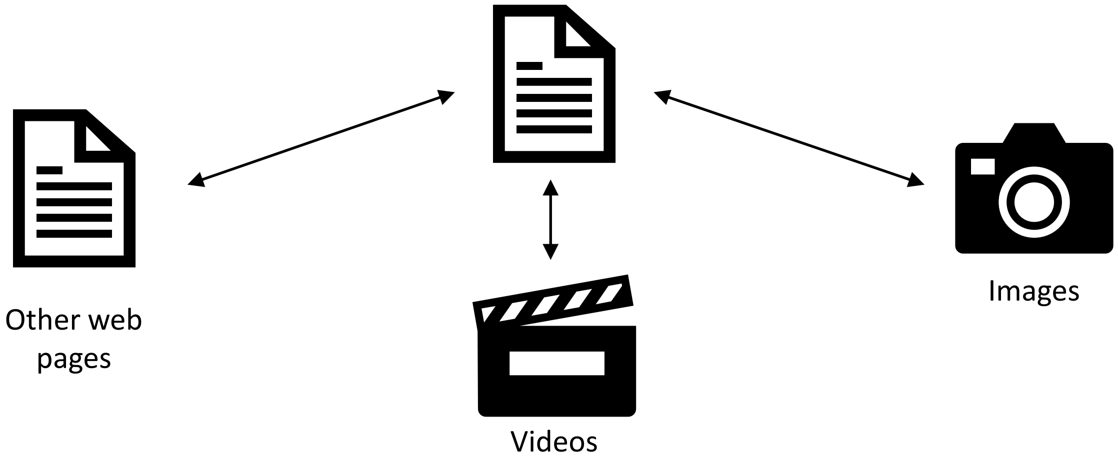
```
<b>Hello</b> World
```

⇩

Hello World

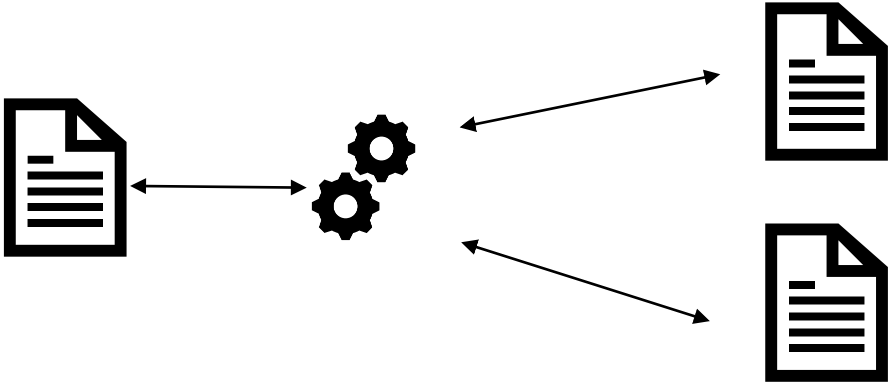
Web Pages

- Web pages are in turn applications:
  - They can contain a lot of different resources



Web Pages

- Web pages are in turn applications:
  - They contain a lot of different resources
  - They run JavaScript, a language that can manipulate both data of the page and both of from an external page



Web Pages

- Pages contain and manage a lot of information
  - Private user data
  - Security tokens
    - Tokens used to keep track of a session
  - Control panels
    - There are control panels that permit to manage servers/applications
    - What would happen if an attacker could execute actions on them?

Web Pages

- JavaScript, in particular, is dangerous from a security point of view
  - Can load external resources
  - Can send data to an external server
- Why use secure a site with a form of authentication if any page on the internet can just fetch the content it wants?

Web Pages

- The browser sends its cookies to every page it visits
- So, if evil.com has a script to send a request to good.com and retrieve the response, then every form of authentication that good.com implements is basically useless

Same-Origin Policy

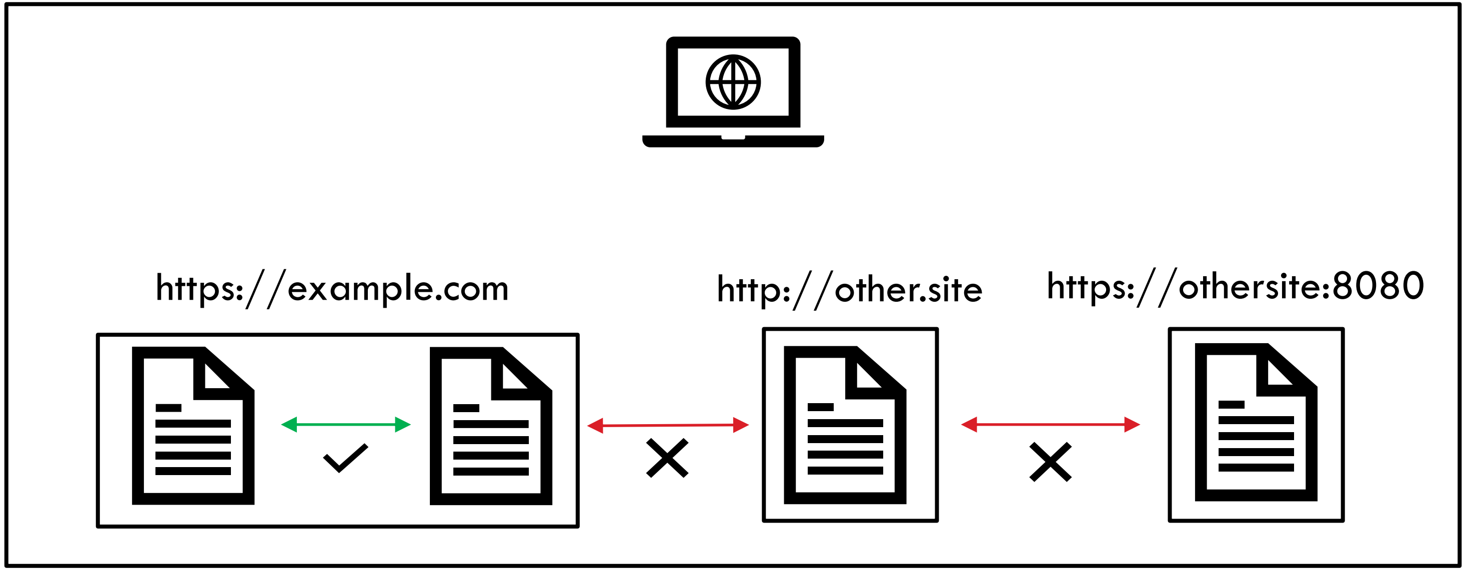
- To cope with these problems, every browser implements the *Same-Origin Policy* (SOP)
- On the paper the concept is simple:
  - Every Script loaded in an origin can access the data of an external resource if and only if the origin is the same
  - The origin becomes a perimeter, secrets and private data inside a page cannot be accessed by pages on another origin

Same-Origin Policy

- An important concept is the Origin
- The Origin of a web page is the combination of the protocol, hostname, and port

```
https://example.com:8080/foo/bar
                ⇩
https://example.com:8080
```

Same-Origin Policy



Outline

- Client-Side Security
  - Resources
  - SOP – Same Origin Policy
- XSS – Cross-Site Scripting
- CSRF – Cross-Site Request Forgery

Cross-Site Scripting

- From a security point of view, it is interesting to see how a "bad guy" can break the Same-Origin Policy
- Because of how complex web pages are, obviously, there are multiple ways to do so
- The most dangerous and the most common types of vulnerabilities that break the SOP are the Cross-Site Scripting Vulnerabilities (XSSs)

Cross-Site Scripting

- An XSS is an injection of JavaScript code inside a page
- If an attacker manages to execute JavaScript code inside a page of a victim, he/she can:
  - Steal session cookies, and then log-in as the victim
  - Steal information inside pages
  - Do actions on behalf of the victim

Cross-Site Scripting

- XSSs are a type of code injections
- They happen in two ways:
  - When the backend reflects unsafe user input on the output page, without any type of sanitization
  - When a generated page unsafely use external input (DOM XSS)

Reflected XSS

- Reflected XSSs are the most common XSS types
- They are called reflected because they happen when the content of some HTTP variable is "reflected" (= echoed) on the response page
- If this reflection happens without any sanitization, then it is possible to inject a <script> tag, and consequentially some JavaScript code

Reflected XSS

- For example, take the following PHP code of the site "foo.bar":

```
<?php
echo 'hello ' . $_GET['name'];
```

Reflected XSS

- If the script is in the page "hello.php", then at the link http://foo.bar/hello.php?name=baz a user would get the response:

```
hello baz
```

- The parameter *name* is reflected in the page, without any kind of sanitization

Reflected XSS

- An attacker could then inject a <script> tag, with some php
- For example, once a user clicks on the link

  http://foo.bar/hello.php?name=<script>alert(1)</script>

- The page would execute the JavaScript code "alert(1)", and the user would only see a pop-up with a 1

Stored XSS

- Stored XSSs work in a similar way of the reflected XSSs
- They are an injection flaw too, but they don't require user interaction at all
- A stored XSS takes place when some data that is "stored" by the site is reflected somewhere without any sanitization
- Stored means that it is saved in some way by the application, for example inside a database

Stored XSS

- A typical example is the comment section of a blog:
  - If the content of a comment is not sanitized before being outputted in the page, a rogue user can inject JavaScript code
- Since the comment is saved in the database, every user that visits the page with that comment is "attacked" by the malicious JavaScript code

Stored XSS

- This kind of injection doesn't require any user interaction at all, because the attacker doesn't need to send a link to the victim
- The user just needs to wait until the victim visit the compromised page by itself

Preventing XSS

- Reflected and Stored XSS are difficult to prevent:
  - Even in a small web application, there are a lot of ways some user input can be output
  - Different contexts require different sanitization methods
- As always, the general rule is to sanitize any unsafe data

Preventing XSS

- Generally, the sanitization is done by replacing every dangerous HTML character with its HTML encoded version
- Dangerous characters are different by the point in which they are echoed on the page, normally they are:
  - < > "
- Once "HTML-encoded", they become:
  - < > "

Preventing XSS

- Every language has its functions that do this kind of sanitization
- PHP for example uses htmlspecialchar
  - https://www.php.net/manual/en/function.htmlspecialchars.php
- Using this kind of function can be dangerous:
  - It is very likely to forget to call them when echoing some user-supplied data

Preventing XSS

- A less error-prone way to prevent XSS is to use templates
- Templates are documents that look-like the final page, with placeholders in which unsafe data can be echoed in the page in a safe manner
- Since every output in which some user-input can be echoed is a placeholder, the web application can sanitize everything by default

Preventing XSS

- For example, a template of the "hello" page would look like the following

```
<html>
<head>….</head><body>
Hello {{user_name}}
</body>
</html>
```

- The "template engines", that is the program that renders the template, would then substitute the `user_name` var with the sanitized user input, thus preventing the XSS

Outline

- Client-Side Security
  - Resources
  - SOP – Same Origin Policy
- XSS – Cross-Site Scripting
- CSRF – Cross-Site Request Forgery

CSRF

- Cross-Site Request Forgery (CSRF) is a type of attack that occurs when a malicious Web site, email, blog, instant message, or program causes a user's Web browser to perform an unwanted action on a trusted site for which the user is currently authenticated

CSRF

- Imagine a bank that lets users send money to other users
- One they start the transaction, a link the following is generated:

```
http://bank.site/transact?to=user2&money=1000
```

- An attacker could replace the "user2" with its account name, then send the link to the victim
- If the victim clicks on the link, then the transaction

CSRF

- An attacker could replace `user2` with its account name, then send the link to the victim

```
http://bank.site/transact?to=attacker&money=1000
```

- If the victim clicks on the link, then the transaction would start, sending 1000€ to `attacker`

CSRF - Mitigations

- This happens because HTTP is stateless, and the server is not aware of the site from which the link is accessed
- Prevention is simple:
  - Make the request stateful

CSRF - Mitigations

- To make the request stateful there are a lot of ways
- The best way is to create a random token and save it in the session
- When the link that triggers an action is generated, the token is inserted
- The backend then checks if the provided token is the same as the one in the session, and, if not the backend rejects the action

CSRF - Mitigations

- This works because of the SOP
- The only way an attacker could get a valid token, would be:
  - To get the content of the page (and with SOP the attacker can't)
  - To get the cookie/session (and the attacker can't)

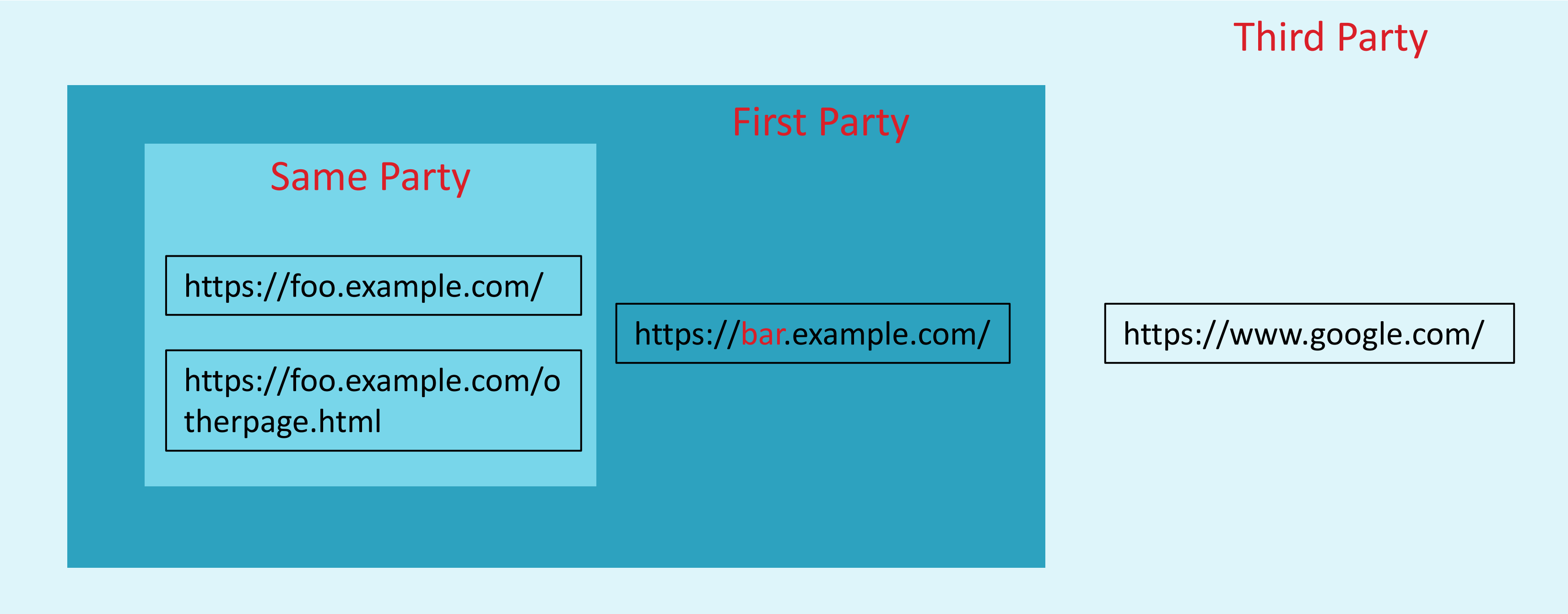
CSRF - Mitigations

- Since 2018, most browsers implement the SameSite attribute for cookies
- The SameSite attribute governs the way a cookie is shared or not to a certain site when visiting or loading assets from another location

CSRF - Mitigations

- The decision on sharing a cookie or not is based on the origin of the cookie, with the concept of "parties"
  - Same Party: The origin is the same
  - First Party site: The origin differs only by the subdomain
  - Third Party site: The origin is completely different

CSRF - Mitigations



CSRF - Mitigations

- To activate this mechanism, cookies use the flag "SameSite"
- This flag can have 3 different values:
  - Strict: Share the cookie only with Same Site locations
  - Lax: Share the cookie only with First Party locations (and same site)
  - None: Share the cookie with every site
- Note that if the SameSite attribute is not set, by default, the browser treats the cookie as lax

CSRF - Mitigations

- For example, let's say the site *www.google.com* has the cookie "foo" with the SameSite attribute set to lax
- If *www.example.com* loads an image from *www.google.com*, the browser will not send the cookie "foo" with the request, because *www.example.com* is a third party site respectively to *www.google.com*

CSRF - Mitigations

- This mechanism effectively prevents CSRF, because an attacker is not able anymore to send authenticated requests to other origins

Client-Side Security

Riccardo BONAFEDE
Università di Padova

CYBER CHALLENGE.IT

CYBERSECURITY NATIONAL LABORATORY

https://cybersecnatlab.it