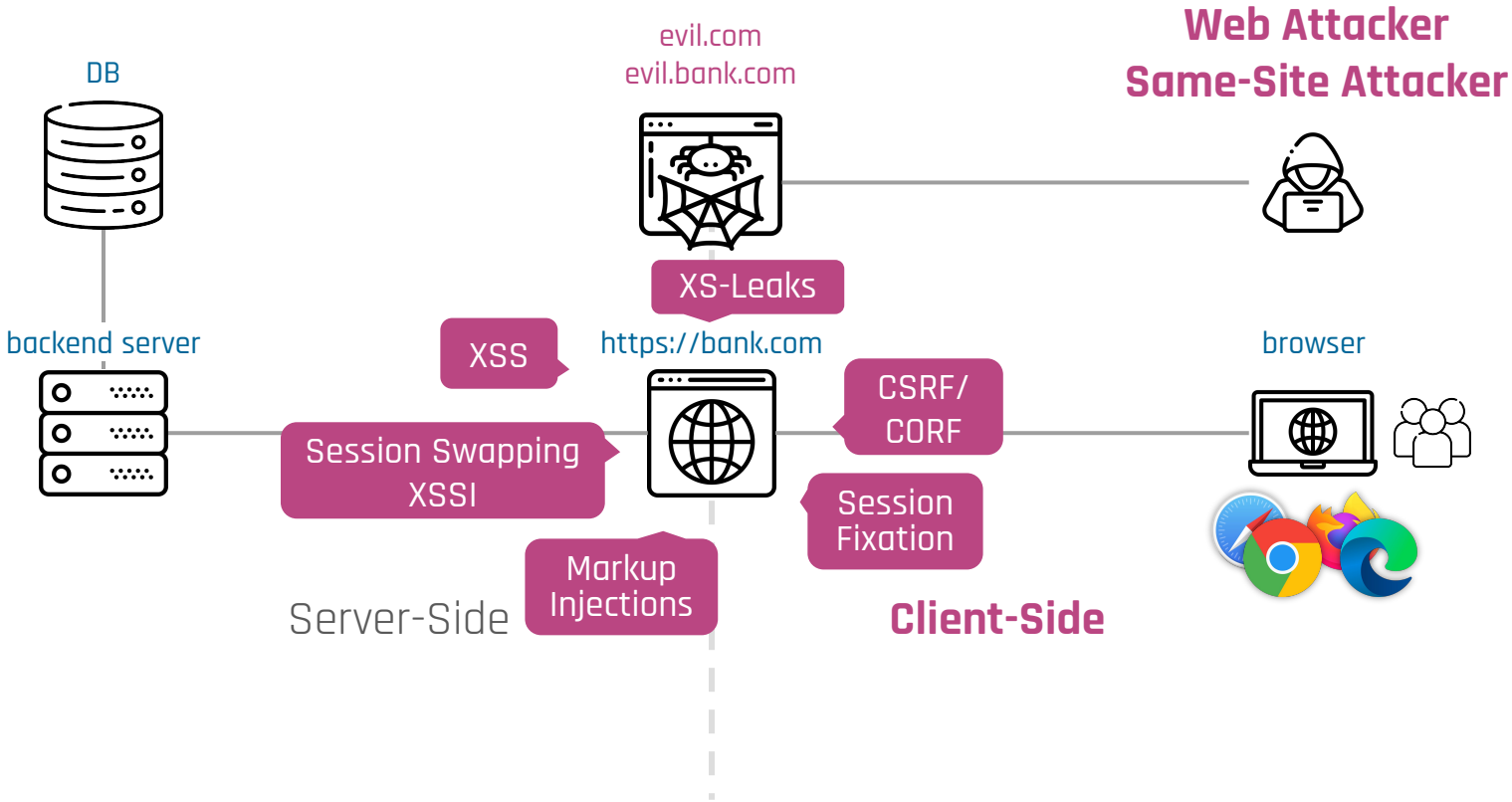
Client-Side Web (in)Security

Introduction to Security (192.019)

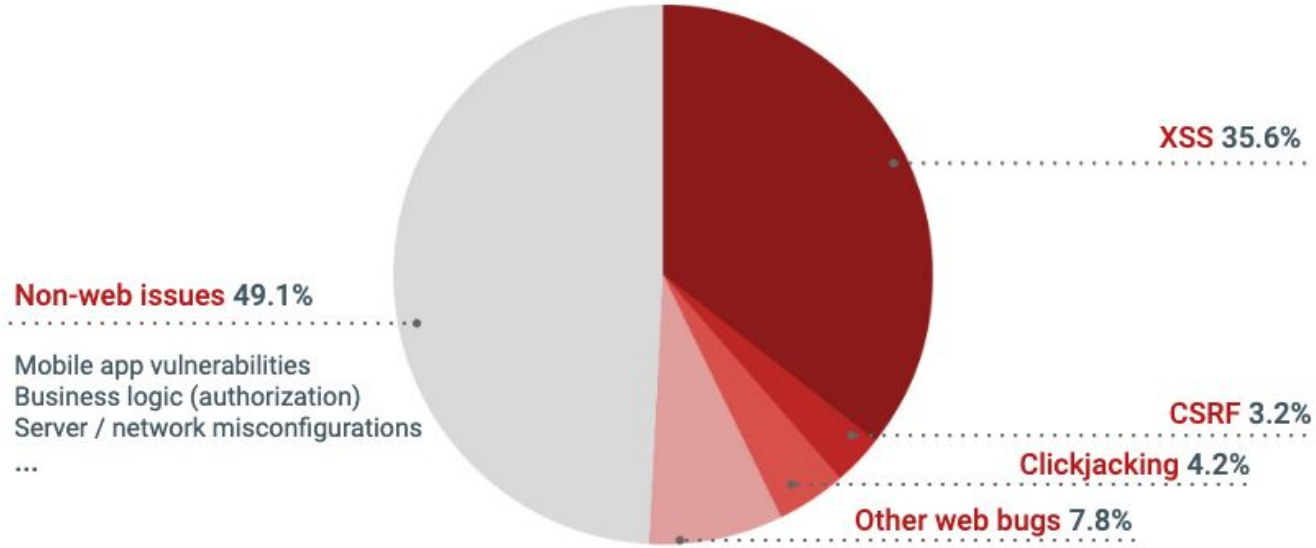
Marco **Squarcina**, Sebastian **Roth**

Security & **Privacy Research Unit** (192-06)
https://**secpriv.wien**

On Today's Menu: Client-Side Vulnerabilities



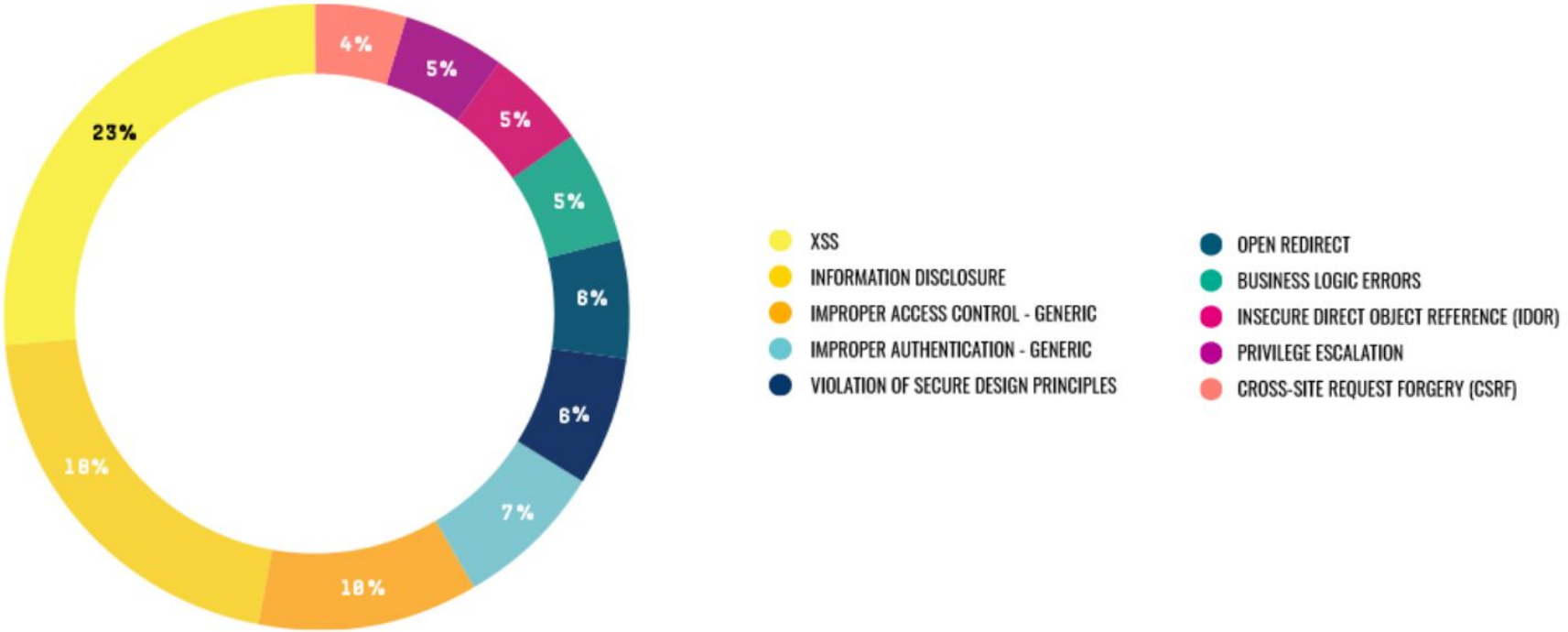
Google VRP, 2018



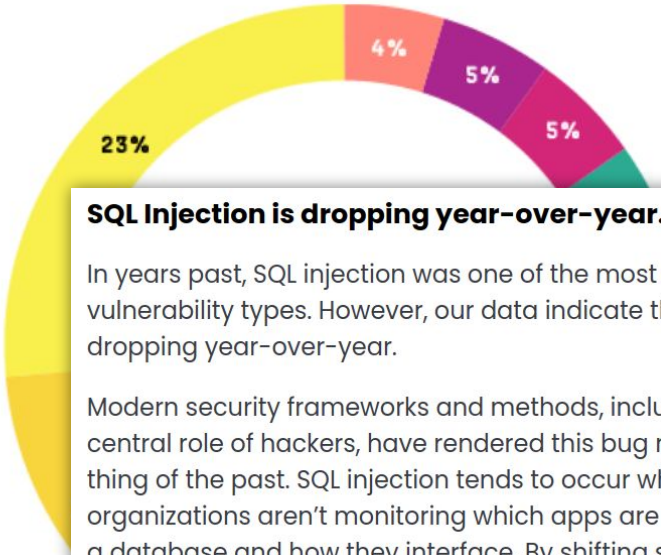
Total Google Vulnerability Reward Program payouts, covering regular user-facing products (including web applications)

3.4 million $ of total rewards in 2018

HackerOne Top 10, 2020



HackerOne Top 10, 2020



SQL Injection is dropping year-over-year.

In years past, SQL injection was one of the most common vulnerability types. However, our data indicate that it's been dropping year-over-year.

Modern security frameworks and methods, including the central role of hackers, have rendered this bug nearly a thing of the past. SQL injection tends to occur when organizations aren't monitoring which apps are mapped to a database and how they interface. By shifting security left, organizations are leveraging hackers and other methods to proactively monitor attack surfaces and prevent bugs from entering code.

Organizations are using creative tools to cut down on XSS.

Cross-site Scripting (XSS) continues to be the most awarded vulnerability type with US$4.2 million in total bounty awards, up 26% from the previous year.

XSS vulnerabilities are extremely common and hard to eliminate, even for organizations with the most mature application security. XSS vulnerabilities are often embedded in code that can impact your production pipeline.

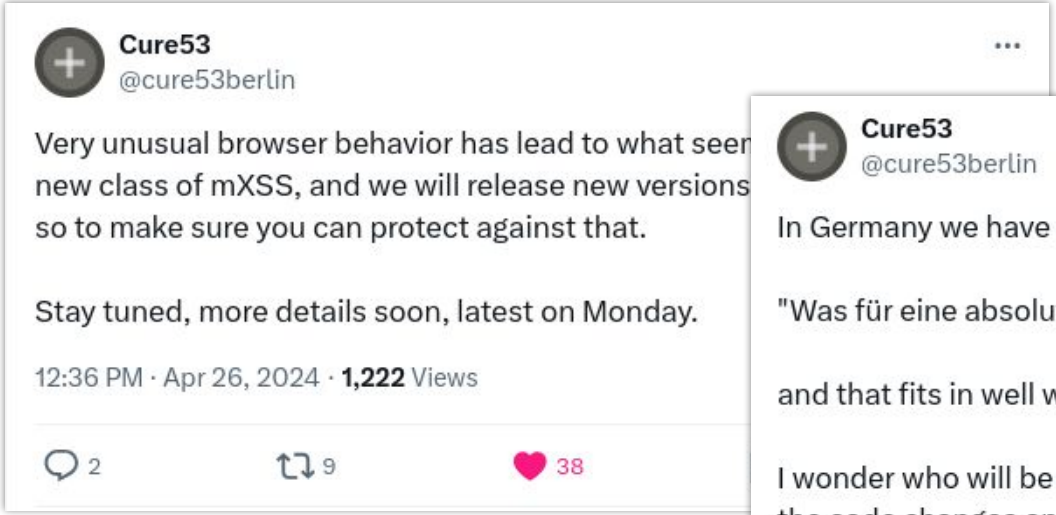
- XSS
- INFORMATION DISCLOSURE
- IMPROPER ACCESS CONTROL - GENERIC
- IMPROPER AUTHENTICATION - GENERIC
- VIOLATION OF SECURE DESIGN PRINCIPLES
- OPEN REDIRECT
- BUSINESS LOGIC ERRORS
- INSECURE DIRECT OBJECT REFERENCE (IDOR)
- PRIVILEGE ESCALATION
- CROSS-SITE REQUEST FORGERY (CSRF)

Separating code and data on the client-side is hard

Client-Side Web Security is HARD

cure53/**DOMPurify**

DOMPurify - a DOM-only, super-fast, uber-tolerant XSS sanitizer for HTML, MathML and SVG. DOMPurify works with a secure default, but...



Overview

Overview

- **Web Boundaries**
  - Origins, Same Origin Policy and Sites
  - Cross-Origin Communication
- **Cookies and Sessions**
  - Background on Cookies
  - Sessions: Server-Side vs Client-Side
  - Attacks on Cookies: Cookie Tossing, CSRF/CORF, Session Fixation
  - Clickjacking
- **XSS**
  - Attacks, Protections, and Limitations
- **XS-Leaks**

Web Boundaries

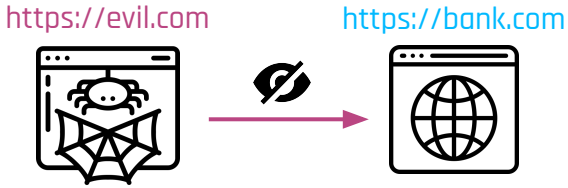
Same Origin Policy (SOP)

- SOP is the **baseline security policy** implemented by browsers (introduced by Netscape 2 in 1995)
- Access control policy that depends on the concept of **origin**, defined as the triplet

  `<protocol, domain, port>`

  Example `<https:, shop.example.com, 443>`
- **Scripts** running on a certain origin can only access **resources** from the **same origin**:
  - access (read/write) to DOM
  - access (read/write) to the cookie jar (relaxed concept of origin)
  - access (read) to network response

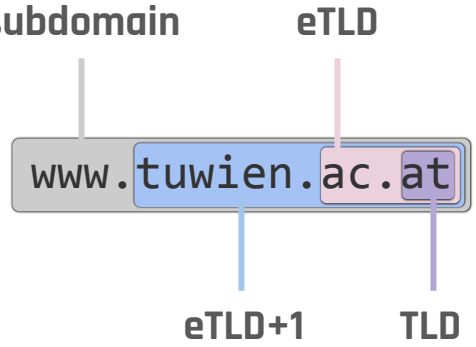
We will talk about cookies and DOM later!



- `https://evil.com` and `https://bank.com` are different origins
- If a user visits `evil.com` in one tab, and `bank.com` in another, evil cannot access the bank account
- Same applies to iframes, etc.

Origins != Sites

- eTLDs (Effective Top Level Domains) are defined by the Public Suffix List (PSL) 🔗publicsuffix.org
- eTLDs+1 are also called registrable domains
- 2 domains belong to the same site if they share a common registrable domains
- For e.g., cookies the protocol also matters




```
// GitHub, Inc.
// Submitted by Patrick Toomey <security@github.com>
githubusercontent.com
githubpreview.dev
github.io
```

https://www.tuwien.ac.at
https://old-project.tuwien.ac.at
http://test.tuwien.ac.at
http://test.tuwien.ac.at:8080

Same site

https://lavish.github.io
https://wert310.github.io

Cross site

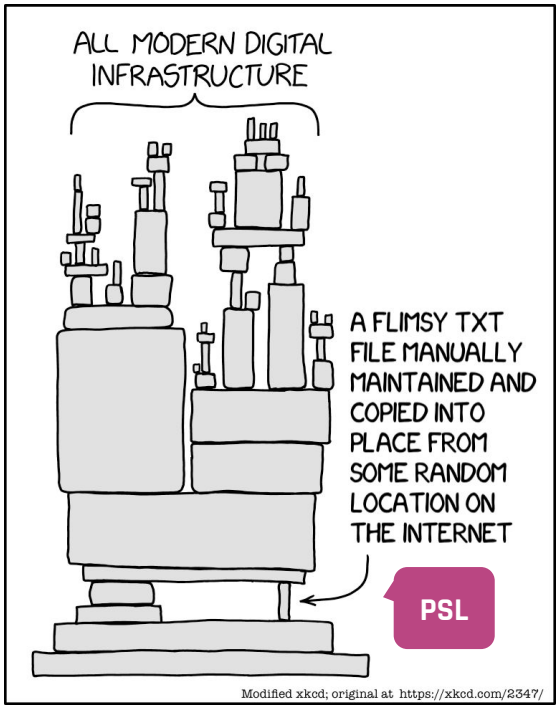
Same-Site as a Security Boundary

https://github.com/sleevi/psl-problems

- Browsers and Web security mechanisms place some **trust in same-site resources**
- **Example:** Protection against Spectre attacks
  Site Isolation in Chromium and Project Fission in Firefox

  "cross-origin attacks within a site are not mitigated"
  – from the original Site Isolation paper, USENIX'19
- **Problem:** Attackers can control same-site resources, e.g., via a **subdomain takeover**! In this case, they can escalate their privileges against the target

  affects cookies, CORS, CSP, postMessages, etc...



Same-Site as a Security Boundary

https://canitakeyoursubdomain.name/

Can I Take Your Subdomain? Exploring Same-Site Attacks in the Modern Web

Marco Squarcina[1] Mauro Tempesta[1] Lorenzo Veronese[1] Stefano Calzavara[2] Matteo Maffei[1]
[1] *TU Wien* [2] *Università Ca' Foscari Venezia & OWASP*

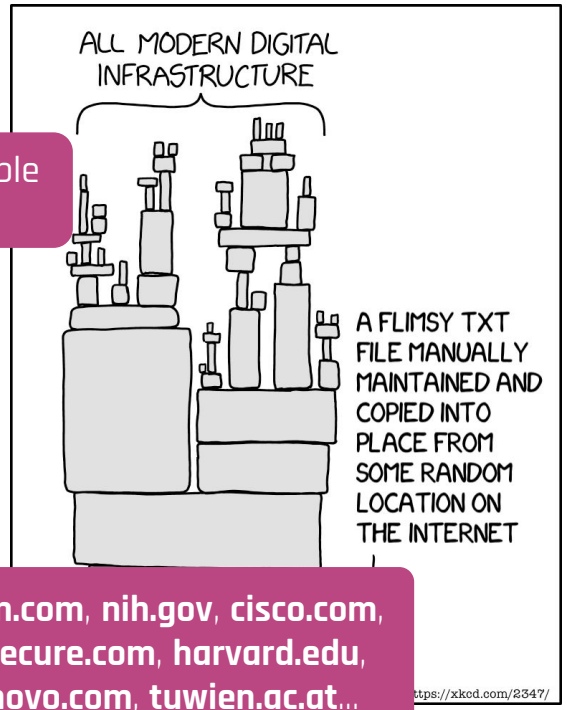
Abstract

Related-domain attackers control a sibling domain of their target web application, e.g., as the result of a subdomain takeover. Despite their additional power over traditional web attackers, related-domain attackers received only limited attention from the research community. In this paper we define and quantify for the first time the threats that related-domain attackers pose to web application security. In particular, we first clarify the capabilities that related-domain attackers can acquire through different attack vectors, showing that different instances of the related-domain attacker concept are worth attention. We then study how these capabilities can be abused to compromise web application security by focusing on different angles, including cookies, CSP, CORS, postMessage, and domain relaxation. By building on this framework, we report on a large-scale security measurement on the top 50k domains from the Tranco list that led to the discovery of vulnerabilities in 887 sites, where we quantified the threats posed by related-domain attackers to popular web applications.

attacker is traditionally defined as a web attacker with an extra twist, i.e., its malicious website is hosted on a sibling domain of the target web application. For instance, when reasoning about the security of `www.example.com`, one might assume that a related-domain attacker controls `evil.example.com`. The privileged position of a related-domain attacker endows it, for instance, with the ability to compromise cookie confidentiality and integrity, because cookies can be shared between domains with a common ancestor, reflecting the assumption underlying the original Web design that related domains are under the control of the same entity. Since client authentication on the Web is mostly implemented on top of cookies, this represents a major security threat.

Despite their practical relevance, related-domain attackers received much less attention than web attackers and network attackers in the web security literature. We believe there are two plausible reasons for this. First, related-domain attackers might sound very specific to cookie security, i.e., for many security analyses they are no more powerful than traditional

1520 vulnerable subdomains



cnn.com, **nih.gov**, **cisco.com**, **f-secure.com**, **harvard.edu**, **lenovo.com**, **tuwien.ac.at**...

Cross-Origin Communication

Cross-Origin Resource Sharing (CORS)

- The SOP does not forbid cross-origin requests, but prevents cross-origin data from being read
- **Cross-Origin Resource Sharing** (**CORS**) provides a controlled way to **relax the SOP**
- JavaScript can access the response content if the `Origin` header in the request matches the `Access-Control-Allow-Origin` header in the response (or if the value is the wildcard *)

Try to execute this from example.com. Then try to fetch https://tuwien.at

```
const res = await fetch('https://minimalblue.com');
const html = await res.text();
console.log(html);
```

Cross-Origin Resource Sharing (CORS)

- The SOP does not forbid cross-origin requests, but prevents cross-origin data from being read
- **Cross-Origin Resource Sharing** (**CORS**) provides a controlled way to **relax the SOP**
- JavaScript can access the response content if the `Origin` header in the request matches the `Access-Control-Allow-Origin` header in the response (or if the value is the wildcard `*`)

Try to execute this from example.com. Then try to fetch https://tuwien.at

Further reading
CORS is way more complicated than this!
See https://jakearchibald.com/2021/cors/

```
const res = await fetch('https://minimalblue.com');
const html = await res.text();
console.log(html);
```

Client-Side Communication with postMessage

- postMessage is a web API that enables cross-origin message exchanges between windows, e.g., a.com can embed a page at b.com as an iframe and communicate with it

```
<script>
window.addEventListener('message', (evt) => {
 if (evt.origin === 'http://b.com') {
   console.log(evt.data);
 }
})
</script>
<iframe src="http://b.com"></iframe>
```

a.com

```
<script>
window.parent.postMessage('hello!',
  'http://a.com');
</script>
```

b.com

Client-Side Communication with postMessage

- postMessage is a web API that enables cross-origin message exchanges between windows, e.g., a.com can embed a page at b.com as an iframe and communicate with it

```
<script>
window.addEventListener('message', (evt) => {
 if (evt.origin === 'http://b.com') {
   console.log(evt.data); // prints hello!
 }
})
</script>
<iframe src="http://b.com"></iframe>
```

a.com

```
<script>
window.parent.postMessage('hello!',
  'http://a.com');
</script>
```

b.com

Client-Side Communication with postMessage

- postMessage is a web API that enables cross-origin message exchanges between windows, e.g., a.com can embed a page at b.com as an iframe and communicate with it

```
<script>
window.addEventListener('message', (evt) => {
 if (evt.origin === 'http://b.com') {
   console.log(evt.data); // prints hello!
 }
})
</script>
<iframe src="http://b.com"></iframe>
```

a.com

```
<script>
window.parent.postMessage('hello!',
  'http://a.com');
</script>
```

b.com

- Message handlers must validate the origin field of incoming messages to communicate only with intended parties
- Failure to do so may result in security vulnerabilities!

Cookies

Cookies

- **No inherent state in HTTP**
  The server does not have a way to re-identify the client across multiple requests
- For static sites, not an issue
- However, dynamic sites may be required to preserve state across requests
  - **Authentication**: Login / User sessions
  - **Personalization**: Site preferences (e.g., language, dark mode)
  - **Tracking**: follow the user from site to site, learn their browsing behavior, etc (same-site and cross-site tracking)
- **Cookies** were introduced in 1994 to go around this limitation!

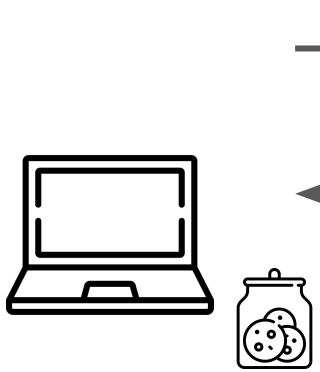
Standard at https://datatracker.ietf.org/doc/html/draft-ietf-httpbis-rfc6265bis

Cookies

- Initially sent by the Web server
- Stored by the Web browser in the so-called **cookie jar** (1 per site)
- Sent on every request to matching domain

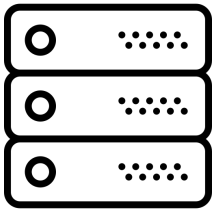
Standard at https://datatracker.ietf.org/doc/html/draft-ietf-httpbis-rfc6265bis

Cookies



Visit `https://www.example.com`

```
HTTP/2.0 200 OK
Content-type: text/html
Set-Cookie: yummy_cookie=choco;
            Path=/; Secure
Set-Cookie: tasty_cookie=pear
...
```



```
GET /page.html HTTP/2.0
Host: www.example.org
Cookie: yummy_cookie=choco;
        tasty_cookie=pear
```

Only the cookie name/value is attached to HTTP requests. Attributes and flags are only specified when the cookie is set.

Attributes					Flags	
Expires	Max-Age	Domain	Path	SameSite	Secure	HttpOnly

Cookies

Attributes					Flags	
Expires	Max-Age	Domain	Path	SameSite	Secure	HttpOnly

Max-Age and **Expires** define when the cookie expires

- If they are not set, the cookie is delete when the browser is closed
- The browser deletes the cookie when `Max-Age` is a negative number or `Expires` is a date in the past. `Max-Age` takes precedence

Path can be used to restrict the scope of a cookie, i.e., the cookie is attached to a request only if its path is a prefix of the path of the request's URL

- If the attribute is not set, the path is that of the page setting the cookie

Cookies

Attributes					Flags	
`Expires`	`Max-Age`	`Domain`	`Path`	`SameSite`	`Secure`	`HttpOnly`

When the **`Secure`** attribute is set, the cookie

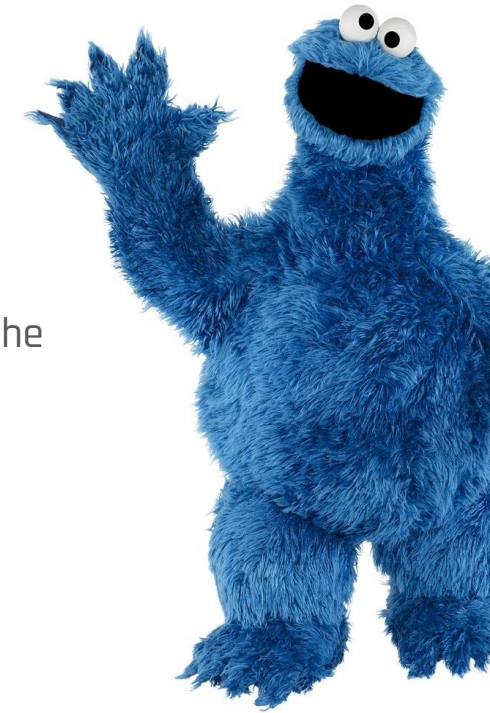
- is attached only to HTTPS requests (**confidentiality**)
- can not be set or overwritten by HTTP requests (**integrity**)
- protection against network attackers

The **`HttpOnly`** attribute prevents JavaScript from reading the value of the cookie via `document.cookie`

- additional protection in case of XSS vulnerabilities: the attacker cannot obtain the session cookie of the victim

Cookies

Attributes					Flags	
Expires	Max-Age	Domain	Path	SameSite	Secure	HttpOnly



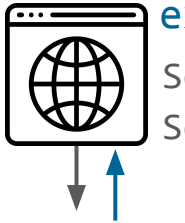
If the **Domain** attribute is not set, the cookie is attached only to requests to the domain that set the cookie (port and protocol don't matter)

When the **Domain** attribute is set, the cookie is attached to requests to the specified domain and all its subdomains

- The value can be set up to the eTLD+1 of the current domain
- Same-site attackers can read domain cookies!
- Same-site attacker can also set domain cookies!
- **NEVER USE THE DOMAIN ATTRIBUTE IF POSSIBLE**

Used in multi-origin applications

Cookies

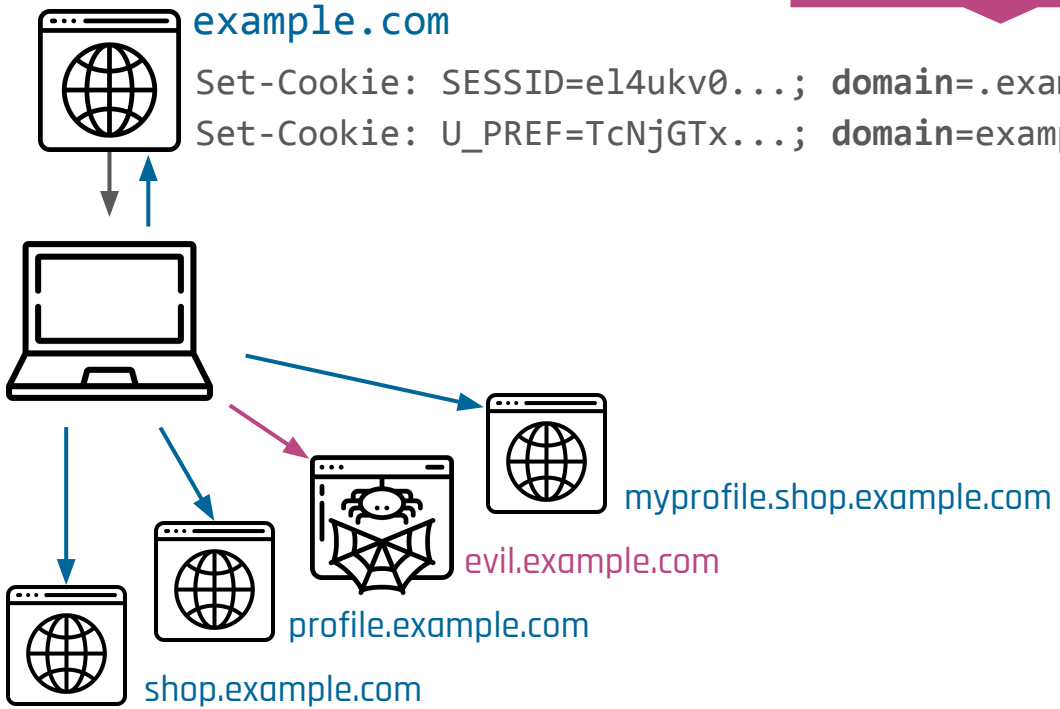


example.com

```
Set-Cookie: SESSID=el4ukv0...; domain=.example.com
Set-Cookie: U_PREF=TcNjGTx...; domain=example.com
```

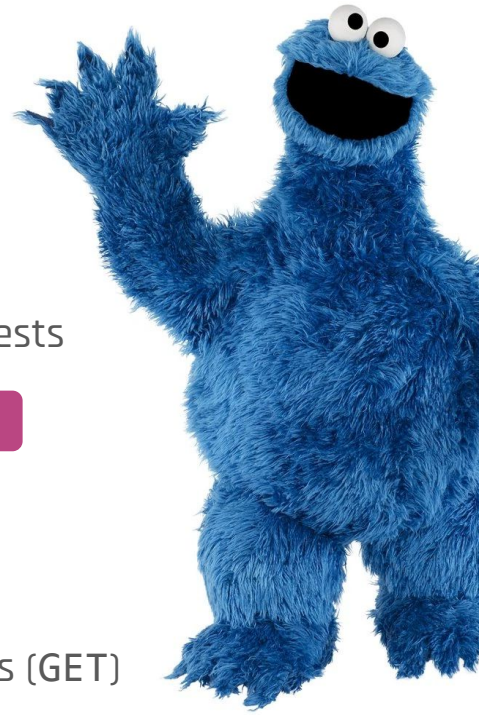
Cookies

The “dot” makes no difference



Cookies

Attributes					Flags	
`Expires`	`Max-Age`	`Domain`	`Path`	`SameSite`	`Secure`	`HttpOnly`



The **`SameSite`** attribute determines if cookies are attached to cross-site requests

> The user has a cookie on `pics.com`
> `example.com` includes an image from `pics.com`
> does the browser send the cookie to `pics.com`?

cross-site request

There are **4 possible values**

- **None** attach 🍪 to cross-site requests. `Secure` must be enabled
- **`Lax`** attach 🍪 only to top-level cross-site navigations using safe methods (`GET`)
- **`Strict`** never attach 🍪 to cross-site requests
- **`Unspecified`** defaults to `Lax` after 2 minutes. Before, attach 🍪 to top-level cross-site `POST` request (a hack for SSO, still dangerous)

browser-dependent behavior

Cookies

Attributes					Flags
Expires	Max-Age	Domain	Path	SameSite	Sec

Lax+POST 2mins hack got standardized

5.4.7.2. "Lax-Allowing-Unsafe" enforcement

As discussed in Section 8.8.6, compatibility concerns may necessitate the use of a "Lax-allowing-unsafe" enforcement mode that allows cookies to be sent with a cross-site HTTP request if and only if it is a top-level request, regardless of request method. That is, the "Lax-allowing-unsafe" enforcement mode waives the requirement for the HTTP request's method to be "safe" in the `SameSite` enforcement step of the retrieval algorithm in Section 5.6.3. (All cookies, regardless of `SameSite` enforcement mode, may be set for top-level navigations, regardless of HTTP request method, as specified in Section 5.5.)

"Lax-allowing-unsafe" is not a distinct value of the `SameSite` attribute. Rather, user agents MAY apply "Lax-allowing-unsafe" enforcement only to cookies that did not explicitly specify a `SameSite` attribute (i.e., those whose same-site-flag was set to "Default" by default). To limit the scope of this compatibility mode, user agents which apply "Lax-allowing-unsafe" enforcement SHOULD restrict the enforcement to cookies which were created recently. Deployment experience has shown a cookie age of 2 minutes or less to be a reasonable limit.

quests

est

ods (GET)

op-level

dependent behavior

Cookies (Examples)

Set-Cookie header	Set by	Action	
`s1=v1; Secure; SameSite=Lax`	`https://a.com`	Clicking the link to `https://a.com:8443` from `http://b.com`	
`s2=v2; HttpOnly; SameSite=Strict`	`https://a.com`	Form submission via POST to `https://a.com` from `https://foo.bar.a.com`	
`s3=v3; SameSite=Lax`	`https://a.com`	Page at `https://b.com` includes an iframe with `src=https://a.com/user`	
`s4=v4; SameSite=Lax`	`https://a.com`	Page at `https://b.com` opens a popup (`window.open`) to `https://a.com`	

Cookies (Examples)

Set-Cookie header	Set by	Action	
`s1=v1; Secure; SameSite=Lax`	`https://a.com`	Clicking the link to `https://a.com:8443` from `http://b.com`	✅
`s2=v2; HttpOnly; SameSite=Strict`	`https://a.com`	Form submission via POST to `https://a.com` from `https://foo.bar.a.com`	✅
`s3=v3; SameSite=Lax`	`https://a.com`	Page at `https://b.com` includes an iframe with `src=https://a.com/user`	❌
`s4=v4; SameSite=Lax`	`https://a.com`	Page at `https://b.com` opens a popup (`window.open`) to `https://a.com`	✅

Cookies (Examples)

Set-Cookie header	Set by	Action	
`s5=v5; SameSite=None; HttpOnly`	`https://a.com`	Clicking the link to `https://a.com` from `https://b.com`	
`s6=v6; Domain=a.com; SameSite=Strict`	`http://bad.a.com`	Navigating directly to `https://www.a.com`	
`s7=v7; Domain=a.com; SameSite=Strict`	`http://bad.a.com`	Form submission via POST to `https://login.a.com` from `https://www.a.com`	
`s8=v8; Domain=a.com; SameSite=Strict`	`http://bad.a.com`	Clicking the link to `https://a.com` from `http://bad.a.com`	

Cookies (Examples)

Invalid, missing Secure

Set-Cookie header	Set by	Action	
`s5=v5; SameSite=None; HttpOnly`	`https://a.com`	Clicking the link to `https://a.com` from `https://b.com`	❌
`s6=v6; Domain=a.com; SameSite=Strict`	`http://bad.a.com`	Navigating directly to `https://www.a.com`	✅
`s7=v7; Domain=a.com; SameSite=Strict`	`http://bad.a.com`	Form submission via POST to `https://login.a.com` from `https://www.a.com`	✅
`s8=v8; Domain=a.com; SameSite=Strict`	`http://bad.a.com`	Clicking the link to `https://a.com` from `http://bad.a.com`	❌

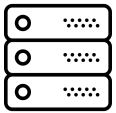
Different protocol = Different site for 🍪

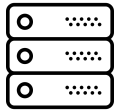
Sessions

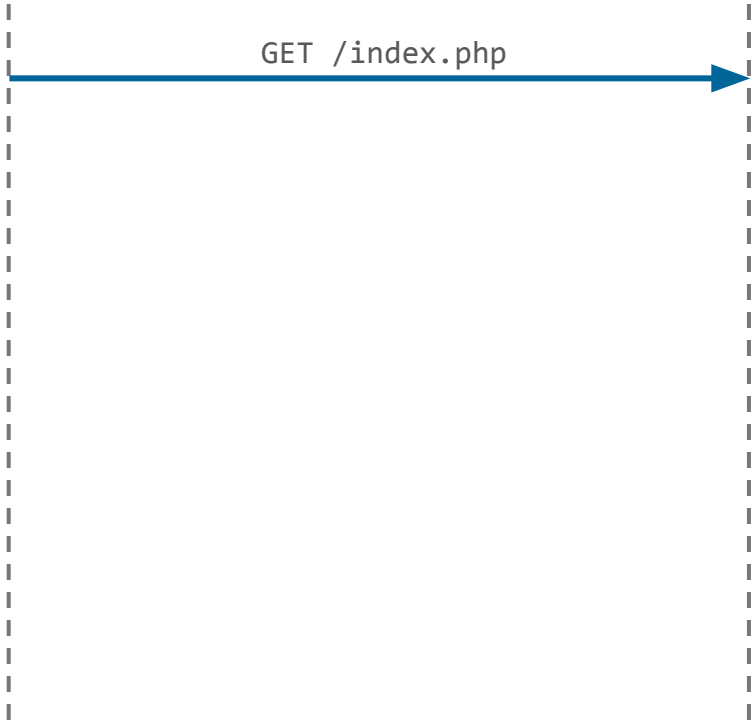
Server-Side Session (PHP)

```php
<?php
session_start();
if (isset($_SESSION['name'])) {
   echo "Welcome back, " . $_SESSION['name'];
} else if (isset($_GET['name'])) {
   $_SESSION['name'] = $_GET['name'];
   header('Location: index.php');
   die();
} else {
   echo "You are not logged in";
}
?>
```

index.php





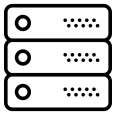


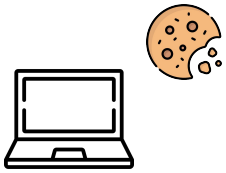
Server-Side Session (PHP)

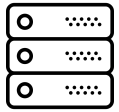
```php
<?php
session_start();
if (isset($_SESSION['name'])) {
   echo "Welcome back, " . $_SESSION['name'];
} else if (isset($_GET['name'])) {
   $_SESSION['name'] = $_GET['name'];
   header('Location: index.php');
   die();
} else {
   echo "You are not logged in";
}
?>
```

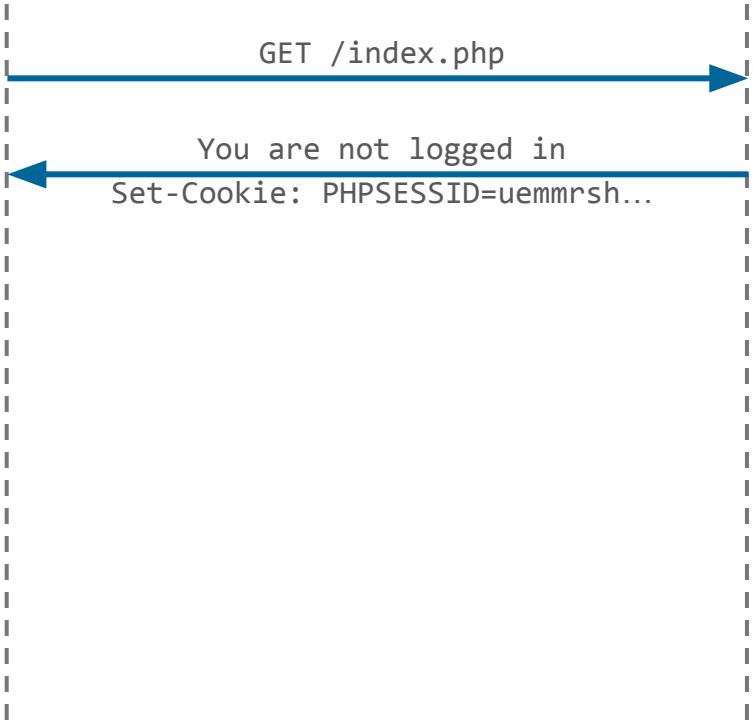
index.php

/var/lib/php/sessions/sess_uemmrsh...







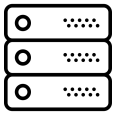


Server-Side Session (PHP)

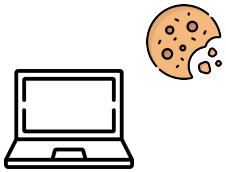
```php
<?php
session_start();
if (isset($_SESSION['name'])) {
   echo "Welcome back, " . $_SESSION['name'];
} else if (isset($_GET['name'])) {
   $_SESSION['name'] = $_GET['name'];
   header('Location: index.php');
   die();
} else {
   echo "You are not logged in";
}
?>
```

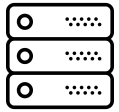
index.php

/var/lib/php/sessions/sess_uemmrsh...



name|s:5:"marco";



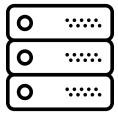


Server-Side Session (PHP)

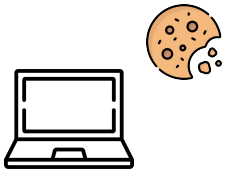
```php
<?php
session_start();
if (isset($_SESSION['name'])) {
   echo "Welcome back, " . $_SESSION['name'];
} else if (isset($_GET['name'])) {
   $_SESSION['name'] = $_GET['name'];
   header('Location: index.php');
   die();
} else {
   echo "You are not logged in";
}
?>
```

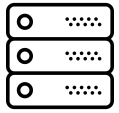
index.php

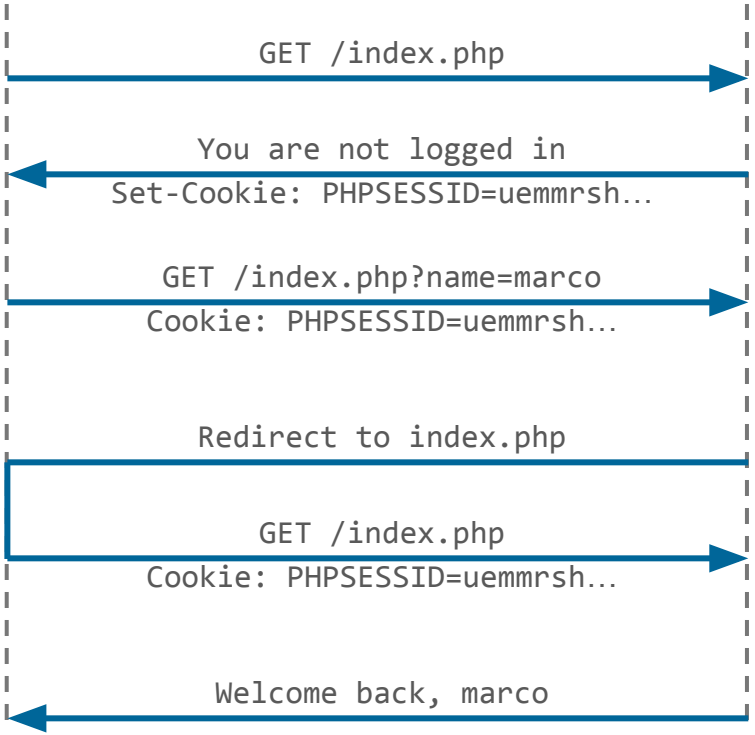
/var/lib/php/sessions/sess_uemmrsh...



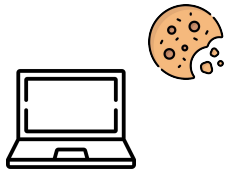
```
name|s:5:"marco";
```

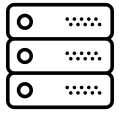






Client-Side Session (Flask)

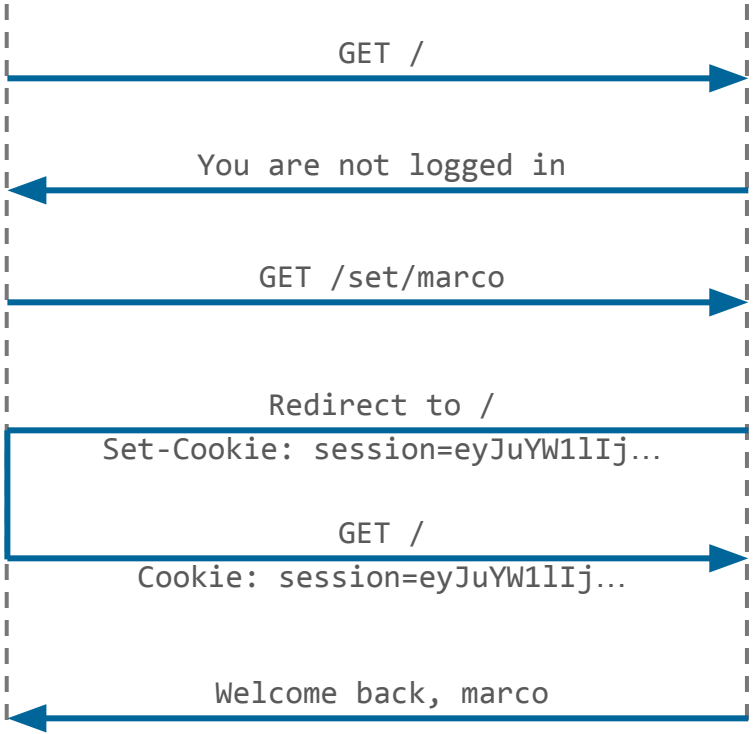




```python
from flask import Flask, session, url_for, redirect

app = Flask(__name__)
app.secret_key = 'afDlNaKCtpexin0DTC'

@app.route("/set/<username>")
def set_user(username):
   session['name'] = username
   return redirect(url_for('home'))

@app.route("/")
def home():
   if 'name' in session:
       return f"Welcome back, {session['name']}"
   return 'You are not logged in'
```

app.py

Nothing is saved on the server!



Client-Side Session (Flask)

Example of a session cookie

`eyJuYW1lIjoibWFyY28ifQ.Zi4y0g.SEit7OXA8MPRHbH9WF8dpwoWQGk`

Base64-serialized session data
- `{'name': 'marco'}`

Compressed if it starts with `.`

HMAC signature of the data computed with `URLSafeTimedSerializer`
- Includes a timestamp
- Uses `app.secret_key` as the HMAC key

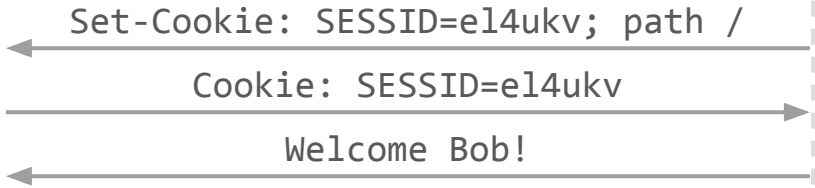
- Users/attackers can read the session content, but cannot forge valid sessions without knowing the value of `secret_key`. Provides **integrity but not confidentiality**!
- Data can be optionally encrypted

Attacks on Cookies

Cookie Tossing (example Session Swapping)

evil.bank.com

bank.com



Cookie Tossing (example Session Swapping)

evil.bank.com

bank.com

Session cookie issued to the attacker

```
Set-Cookie: SESSID=1337;
domain=bank.com; path /account/
```

```
Set-Cookie: SESSID=el4ukv; path /
Cookie: SESSID=el4ukv
Welcome Bob!
```

Cookie Tossing (example Session Swapping)

evil.bank.com

bank.com

Session cookie issued to the attacker

```
Set-Cookie: SESSID=1337;
domain=bank.com; path /account/
```

Can also be set via JavaScript!

```
Set-Cookie: SESSID=el4ukv; path /
Cookie: SESSID=el4ukv
Welcome Bob!

GET /account/index.html HTTP/2.0
Cookie: SESSID=1337; SESSID=el4ukv
Welcome Attacker!
```

Cookie Tossing (example Session Swapping)

- In this case, the attacker logged the victim in their session (**session swapping**)
- This attack can be used, e.g., to track the victim's activity or perform more sophisticated attacks

- Subdomains can **force domain cookies** to all other related-domains, including the apex domain
- Cookies are keyed in the browser by `<name, domain, path>`. When cookies are sent to the server, only the name/value pair is sent by the browser and **attributes are not included**
  - **Servers have no way to tell** which cookie is for which domain/path
  - Most **servers accept the 1st occurrence** of cookies with the same name in the `Cookie:` header
  - Most **browsers place cookies created earlier first**
  - Most **browsers place cookies with most specific paths before** cookies with shorter paths

Preventing Cookie Tossing: Cookie Prefixes

```
Set-Cookie: __Host-sid=honestsession; Secure; Path=/
```

- If a cookie name has the `__Host-` prefix, it is accepted by the browser in a `Set-Cookie` directive only if
  - is marked `Secure`
  - was sent from a secure origin
  - does not include a `Domain` attribute
  - and has the `Path` attribute set to `/`

Another valid prefix is `__Secure-` to lock cookies to HTTPS origins

- This **prevents same-site attackers from forcing a cookie** to the registrable domain since these cookies can be seen as **host-locked**

Cookie Jar Overflow (Eviction)

- Browsers are limited on the number of cookies a **site** can have (~180)
- When there is no space left, **older cookies are deleted**
- Attackers can thus overflow the cookie jar to **evict HttpOnly cookies** or to **bypass cookie tossing protections** on servers that block requests with multiple cookies having the same name

Name	Value	Domain	Path	Expire...	Size ▲	HttpO...	Secure	Same...
session	legit	minimalb...	/	Sessi...	12	✓		

Cookie Jar Overflow (Eviction)

- Browsers are limited on the number of cookies a **site** can have (~180)
- When there is no space left, **older cookies are deleted**
- Attackers can thus overflow the cookie jar to **evict HttpOnly cookies** or to **bypass cookie tossing protections** on servers that block requests with multiple cookies having the same name

Name	Value
session	legit

```
> 03:18:48.407 document.cookie
< 03:18:48.422 ""
> 03:19:02.661 var i; for(i=0; i<200; i++) { document.cookie = "overflow_" + i + "=x"; }
< 03:19:02.784 "overflow_199=x"
> 03:19:38.750 document.cookie = "session=1337"
< 03:19:38.765 "session=1337"
>
```

Cookie Jar Overflow (Eviction)

Fun fact: Safari has no limits

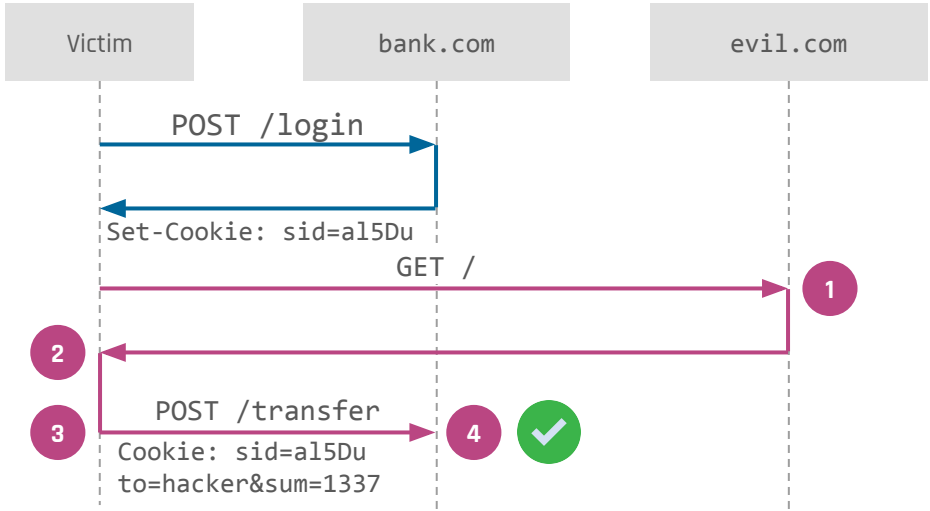
- Browsers are limited on the number of cookies a **site** can have (~180)
- When there is no space left, **older cookies are deleted**
- Attackers can thus overflow the cookie jar to **evict HttpOnly cookies** or to **bypass cookie tossing protections** on servers that block requests with multiple cookies having the same name

Cross-Site Request Forgery (CSRF)

Session integrity violation: attacker performs unwanted actions within the victim's authenticated session

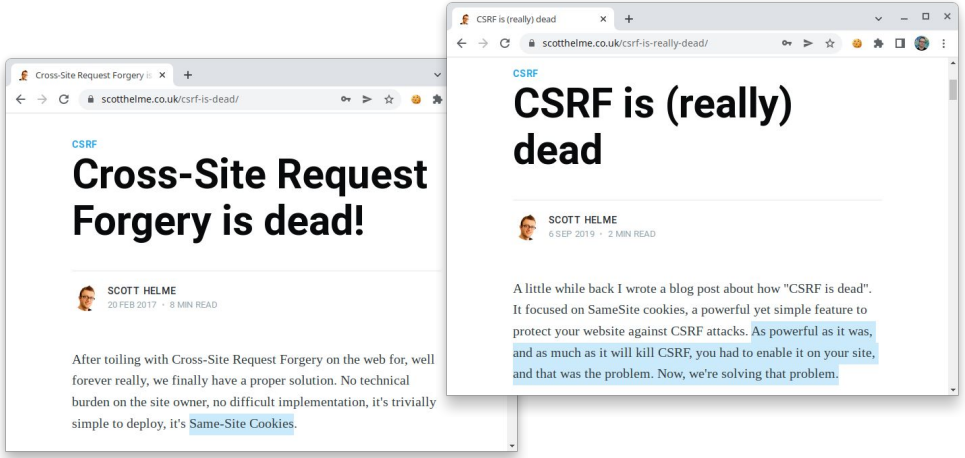
- **Cookies** are **automatically attached** to cross-origin/cross-site requests (e.g., a form submission from `https://example.com` to `https://bank.com`). This is great for **usability**
- Assume that the victim is authenticated on `bank.com`, the attack flow is:

1. The victim visits the attacker's website at `evil.com`
2. The page at `evil.com` contains an HTML form prefilled with a request for a money transfer to the attacker's account at `bank.com`
3. The form is automatically submitted via JavaScript without the user realizing
4. The victim's session cookie for `bank.com` is attached to outgoing POST request and the unwanted money transfer succeeds



Protections Against CSRF (Same-Site Cookies)

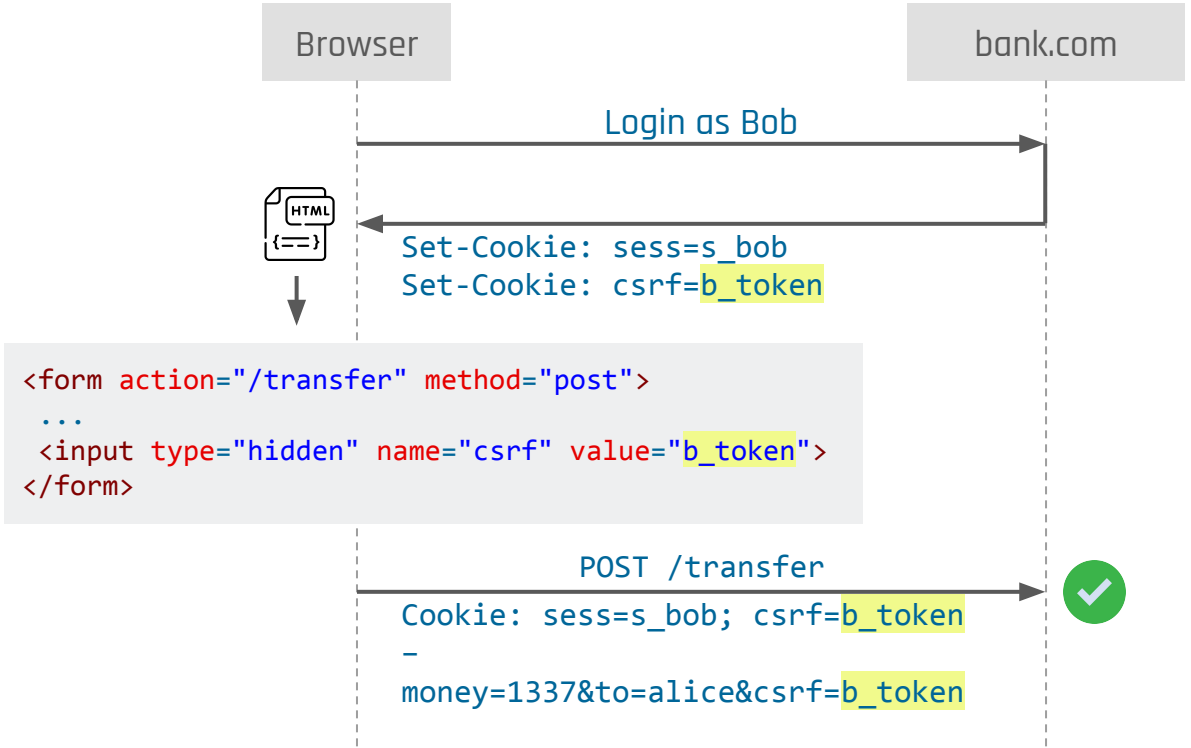
- **SameSite cookies**: do not attach cookies on cross-site requests
- **Problems**
  - not uniform adoption by different browsers, **unsafe defaults**
  - **Cross-Origin Request Forgery** (**CORF**) attacks are not mitigated



- **Tokenization** and other countermeasures are still important: https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html

```
<form action="/transfer" method="post">
 <input type="text" name="to" value="alice">
 <input type="text" name="money" value="1337">
 <input type="hidden" name="csrf" value="r4nd0m">
</form>
```

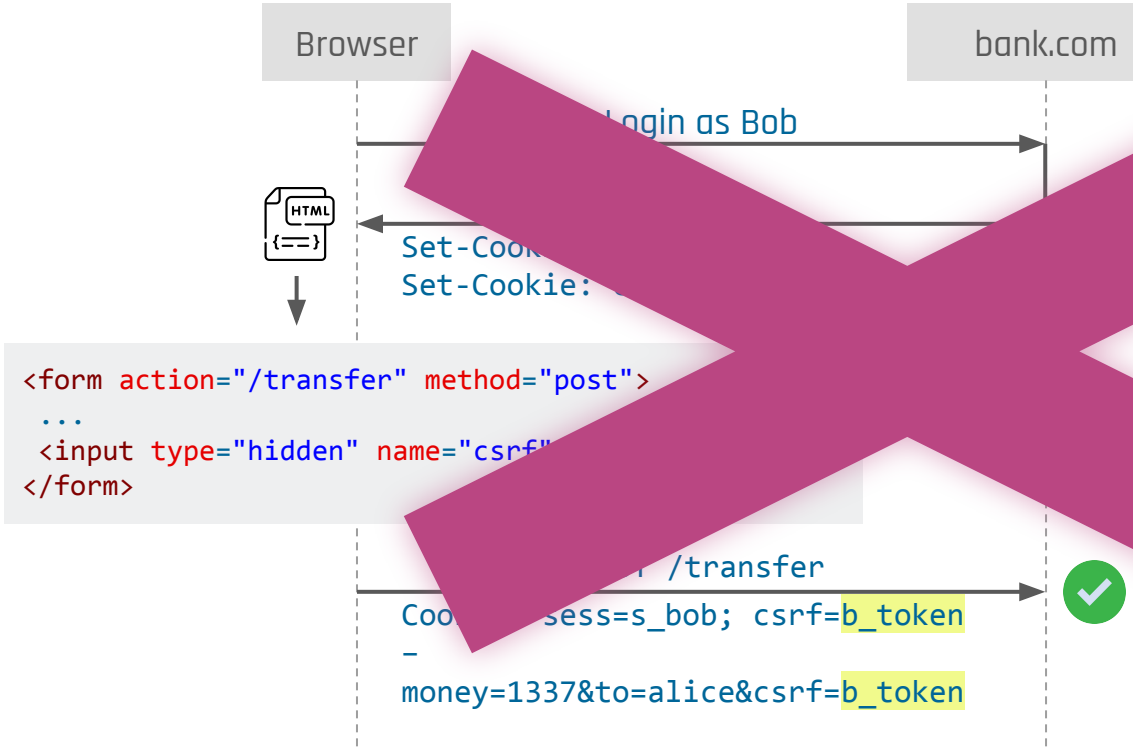
Protections Against CSRF (Double Submit Pattern)



- **CSRF token** sent back to the server as a **cookie** and **POST parameter**
- If the 2 values match, the server accepts the request

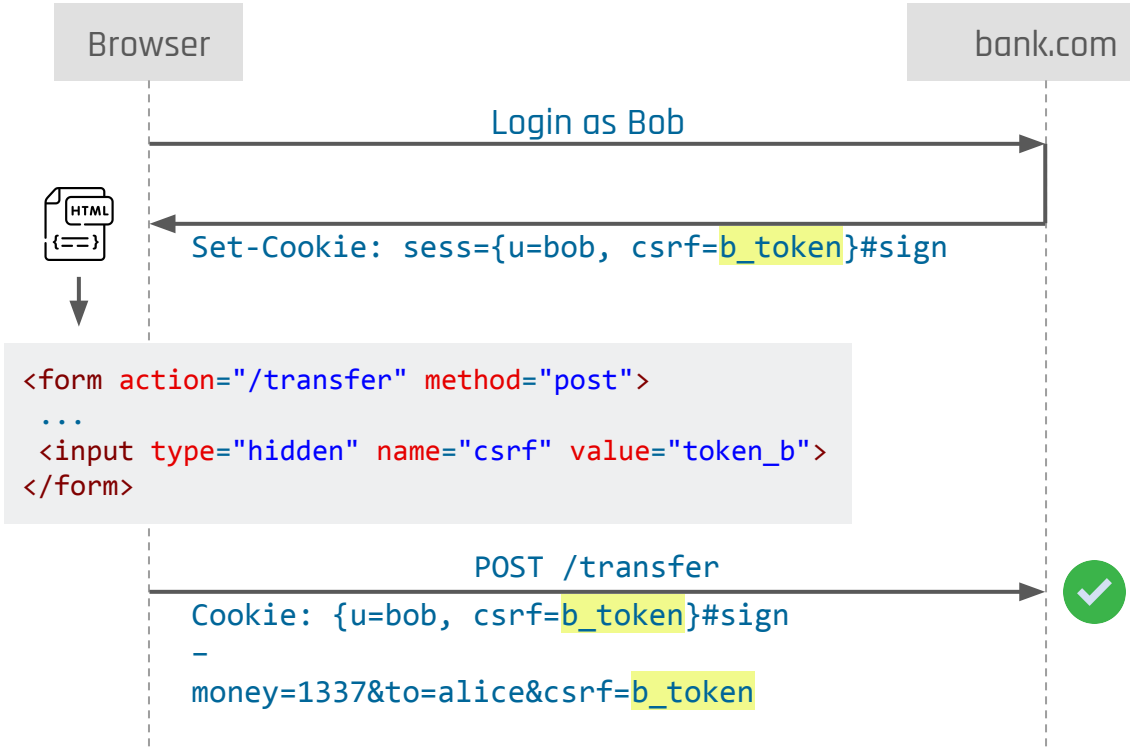
- Assumption
  The attacker can forge a cross-origin request with any POST parameter, but cannot set a cookie for `bank.com`
- True for cross-site attacks
- **False for same-site attacks!** (cookie tossing)

Protections Against CSRF (Double Submit Pattern)



- **CSRF token** sent back to the r as a **cookie** and **POST eter**
  e 2 values match, the server accepts the request
- Assumption
  The attacker can forge a ross-origin request with any arameter, but cannot cookie for `bank.com`
- for cross-site attacks
- **False for same-site attacks!** (cookie tossing)

Protections Against CSRF (Synchronizer Token Pattern)

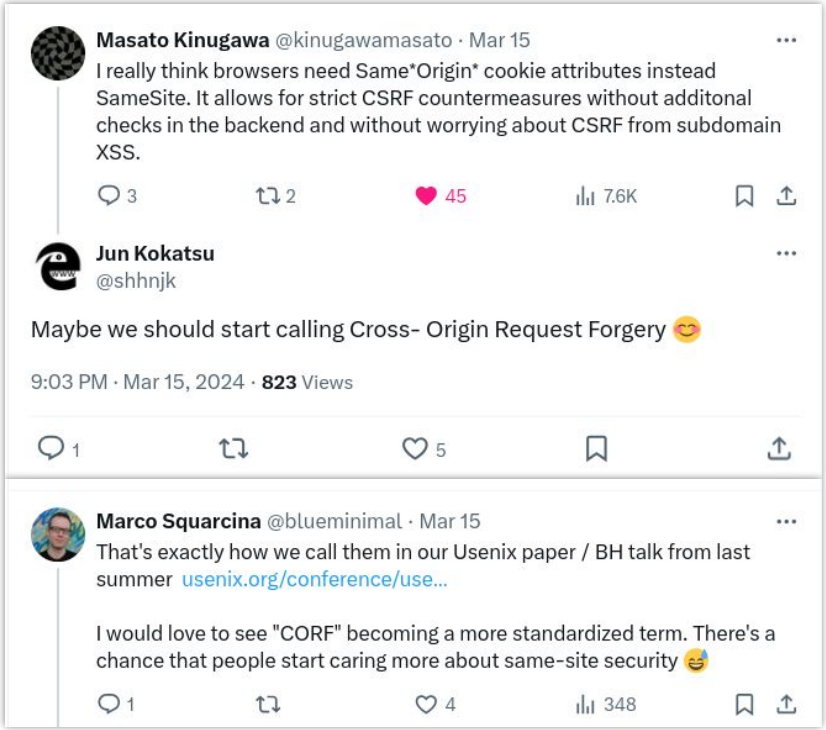


- **CSRF token** is saved in the session (server- or client-side) and sent as a **POST parameter**
- If the 2 values match, the server accepts the request
- Assumption
  If the attacker tries to overwrite the session cookie, the victim gets deauthenticated & attack fails
- If implemented correctly, **robust against cross- and same-site attacks**

Protections Against CSRF (Synchronizer Token Pattern)

- **CSRF token** is saved in the session (server- or client-side) and sent as a **POST parameter**
- If the 2 values match, the server accepts the request
- Assumption
  If the attacker tries to overwrite the session cookie, the victim gets deauthenticated & attack fails
- If implemented correctly, **robust against cross- and same-site attacks**

CSRF? No, Cross-Origin Request Forgery (CORF)!

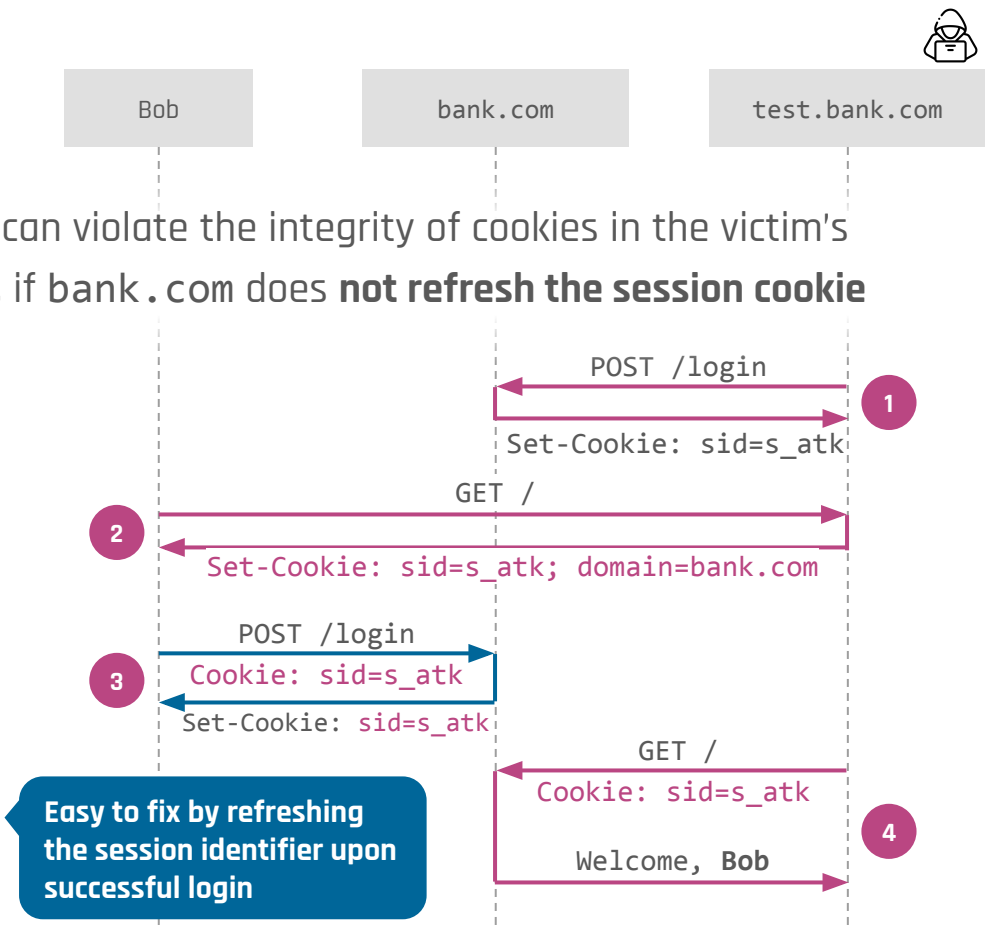


Session Fixation

- Full **session hijacking** when the attacker can violate the integrity of cookies in the victim's browser. Assuming a **same-site attacker**, if `bank.com` does **not refresh the session cookie** after successful login:
  1. The attacker performs a login on `bank.com` and obtains a valid session cookie `S_atk` for their account
  2. The victim visits `test.bank.com` that sets a domain cookie in the victim's browser with value `S_atk`
  3. The victim authenticates on `bank.com`. Notice that domain cookies are attached, so the cookie `S_atk` is sent and promoted to the victim's session identifier
  4. The attack has access to the victim's session since they know `S_atk`

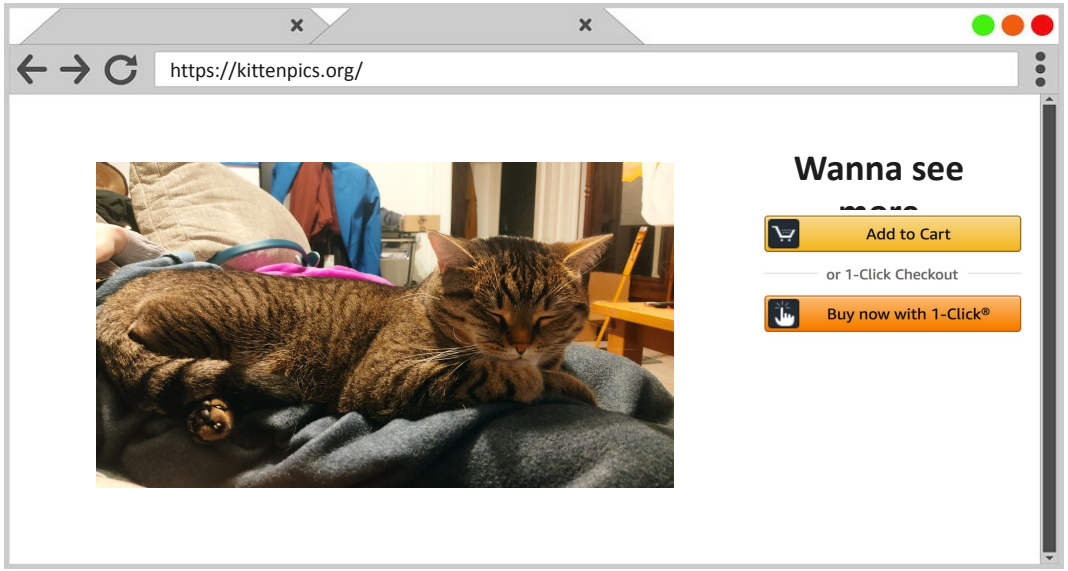
Session Fixation

- Full **session hijacking** when the attacker can violate the integrity of cookies in the victim's browser. Assuming a **same-site attacker**, if `bank.com` does **not refresh the session cookie** after successful login:
  1. The attacker performs a login on `bank.com` and obtains a valid session cookie `S_atk` for their account
  2. The victim visits `test.bank.com` that sets a domain cookie in the victim's browser with value `S_atk`
  3. The victim authenticates on `bank.com`. Notice that domain cookies are attached, so the cookie `S_atk` is sent and promoted to the victim's session identifier
  4. The attack has access to the victim's session since they know `S_atk`



Clickjacking Attacks

Attackers put an opaque iframe on their site that overlays legitimate buttons -> victim clicks on button in iframe!



Clickjacking Defense

Security Response Headers set by the server and enforced by the browser can be used to to control the framing of a Web application:

- Defence in Depth: The Content Security Policy (CSP) directive frame-ancestors:

```
Content-Security-Policy: frame-ancestors example.com partnersite.com;
```

Only allows partnersite.com to load this page in an iframe (more configuration details later)

- Legacy Solution: X-Frame-Options (XFO)

```
X-Frame-Options: DENY
```

Page can not be loaded in an iframe

- Additional mitigation: same-site cookies

XFO is deprecated nowadays due to multiple issues. Still, to also secure legacy clients (e.g. Internet Explorer) it might make sense to deploy it. Notably, modern clients ignore XFO when frame-ancestors is present!

Cross-Site Scripting (XSS)

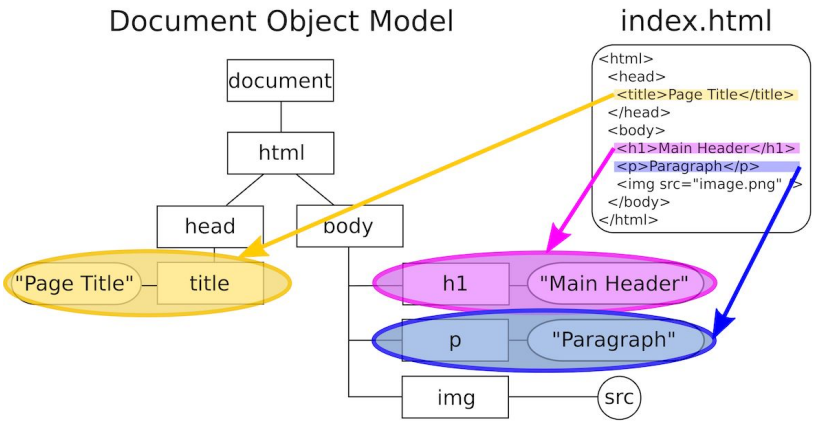
& Friends

Document Object Model (DOM)

- Living standard by WHATWG
  https://dom.spec.whatwg.org

- Tree-like, object-oriented data structure of the elements of an HTML page
- Properties: `document.forms`, `document.links`, …
- Methods: `document.createElement()`, `document.getElementsByTagName()`, …

- By interacting with the DOM, scripts can read and modify the content of the webpage



JavaScript Inclusion

- Inline in the page

```
<script>alert("Hello World!");</script>
```

- As an external file

```
<script type="text/javascript" src="foo.js"></script>
```

- As an event handler

```
<a href="http://www.bar.com" onmouseover="alert('hi');">
```

- Pseudo-URLs in links

```
<a href="javascript:alert('You clicked');">Click me</a>
```

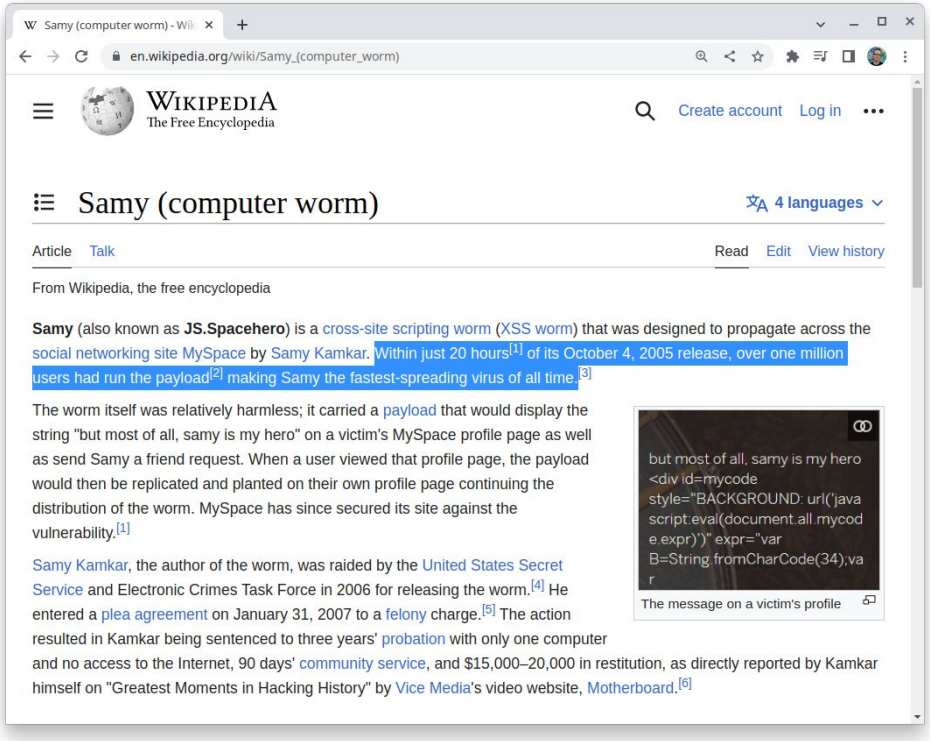
- Import statement (only in modules)

```
import 'https://foo.com/alert.js';
```

Cross-Site Scripting (XSS)

- Whenever an attacker is able to **inject JavaScript code** into a benign page visited by the user, the attacker has full control (read/write) over that DOM!
- **SOP bypass**!
- XSS vulnerabilities are caused by **mixing code and data**... on the client-side.

Samy Worm



Where it all started!

Cross-Site Scripting (XSS)

Reflected XSS

1. A website reads a parameter from an incoming HTTP request and **includes its value into a web page without proper sanitization**
2. User is **tricked into visiting** an honest website with an **URL forged by the attacker** (phishing email, redirect from the attacker's website, …)
3. The attacker's script is now **executed in the victim's browser**, on the target origin (e.g., bank.com) and can completely control the victim's session!

Example

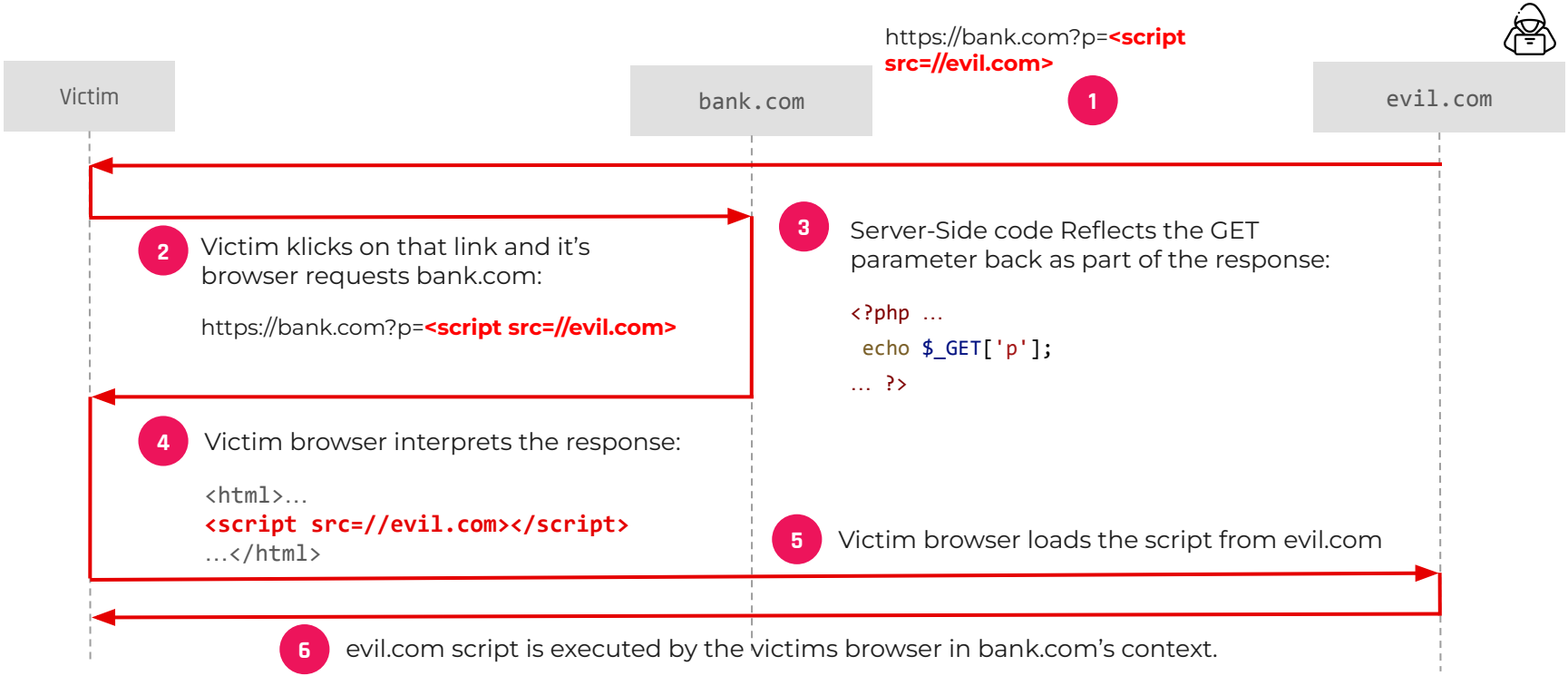
```
https://bank.com/index.php?
    search=<script>alert(“XSS”)</script>
```

```
<html>
   ...
   <div class="search">
       <p>You searched:
           <script>alert("XSS")</script>
       </p>
   </div>
</html>
```

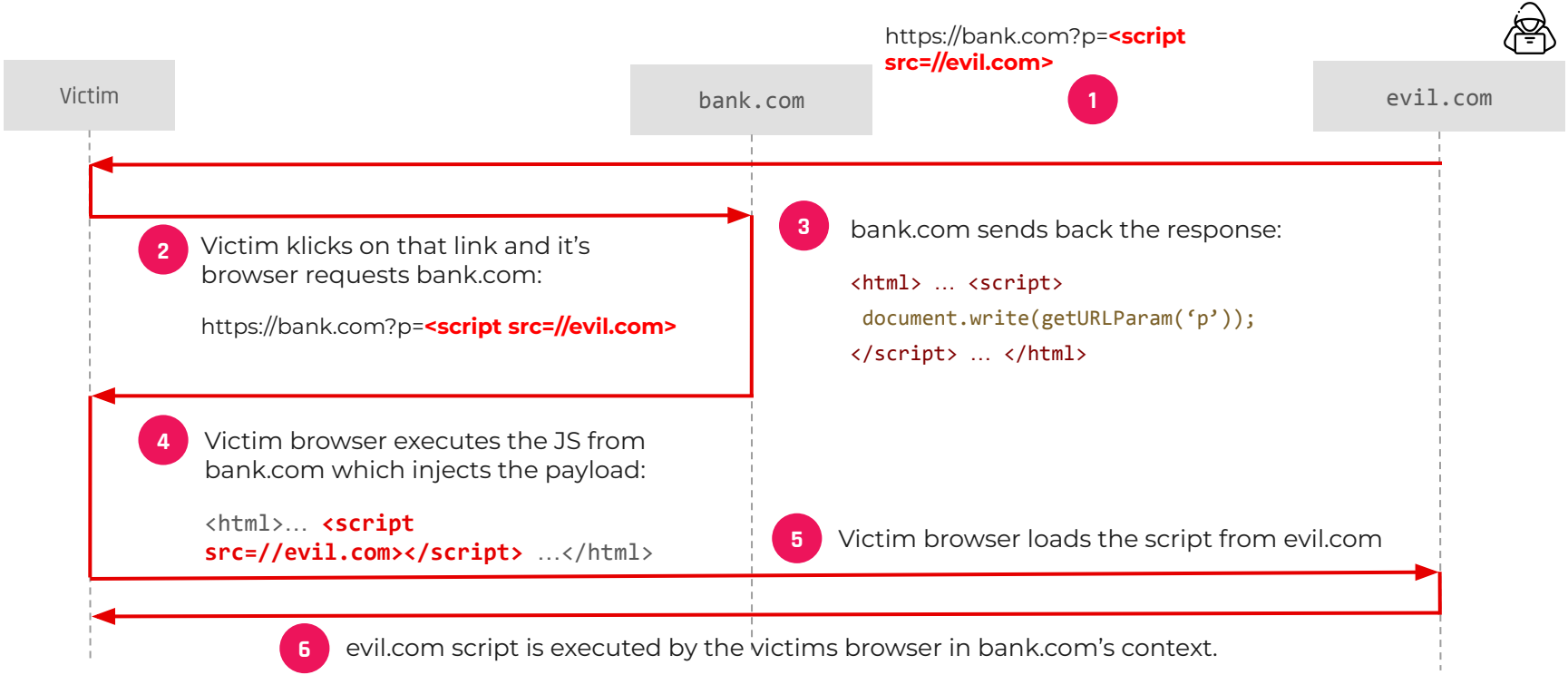
XSS Dimensions

	Server-side	Client-side
Reflected	• Victim must visit a malicious link • No persistent change to the server	• Victim must visit a malicious link • No persistent change to the client • Not visible in the server logs
Stored	• Attacker stores the malicious payload on the server-side • **Every** user is affected on **every visit**	• User must visit malicious link **once** • Single user affected on **every visit**

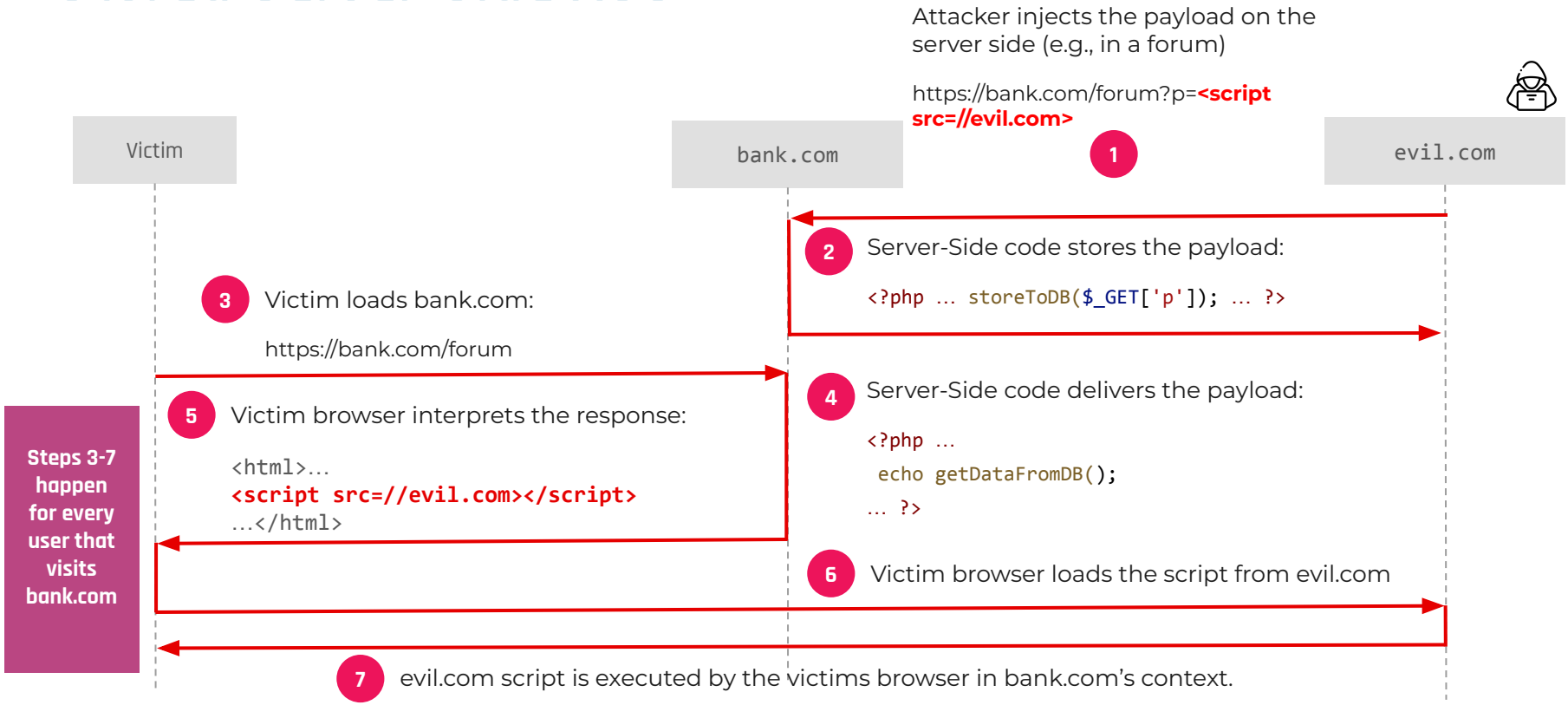
Reflected Server-Side XSS



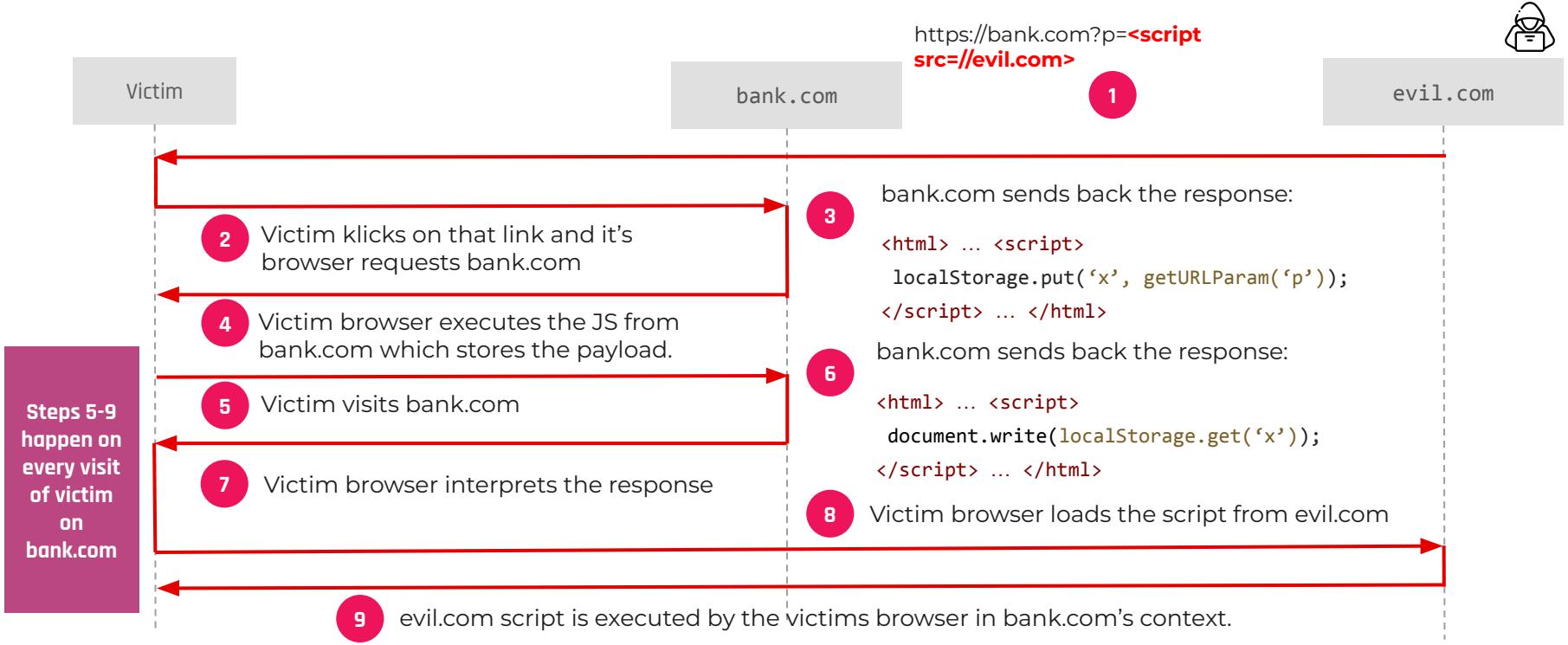
Reflected Client-Side XSS



Stored Server-Side XSS



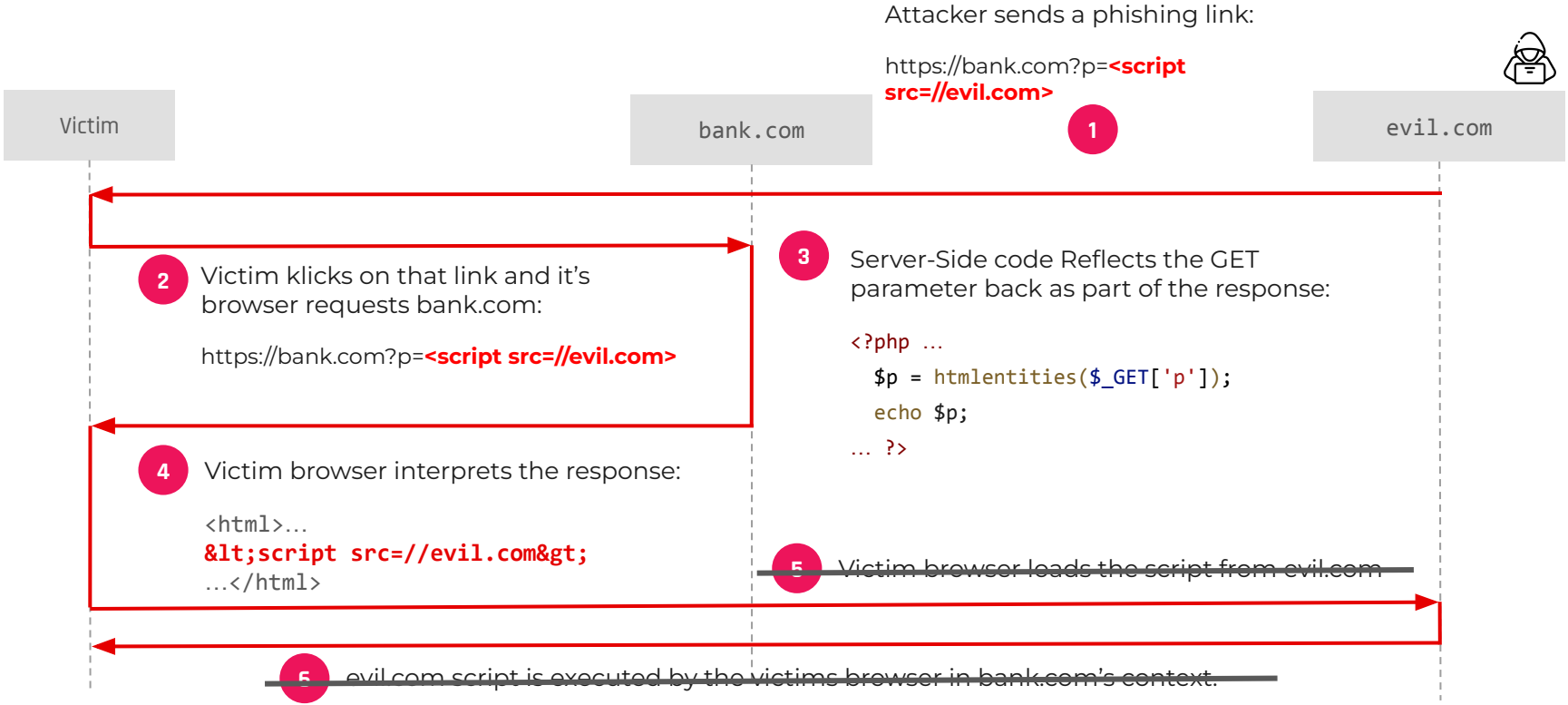
Stored Client-Side XSS



XSS Protection (1/2): Sanitization / Encoding

- **Sanitization on the server-side**, when the context is easy to infer
- Usage of **frameworks and templating libraries** with safe defaults
- **Tricky** against **client-side** attacks
  - **Sanitization** should take place on the **client-side**, where sanitization libraries have the same context/parser as the victim's browser
  - **DOMPurify** is an example, **HTML Sanitizer API** in development https://developer.mozilla.org/en-US/docs/Web/API/HTML_Sanitizer_API
- **Sanitization is HARD, even Google Search was vulnerable to XSS!** https://youtu.be/lG7U3fuNw3A

Reflected Server-Side XSS (with encoding)



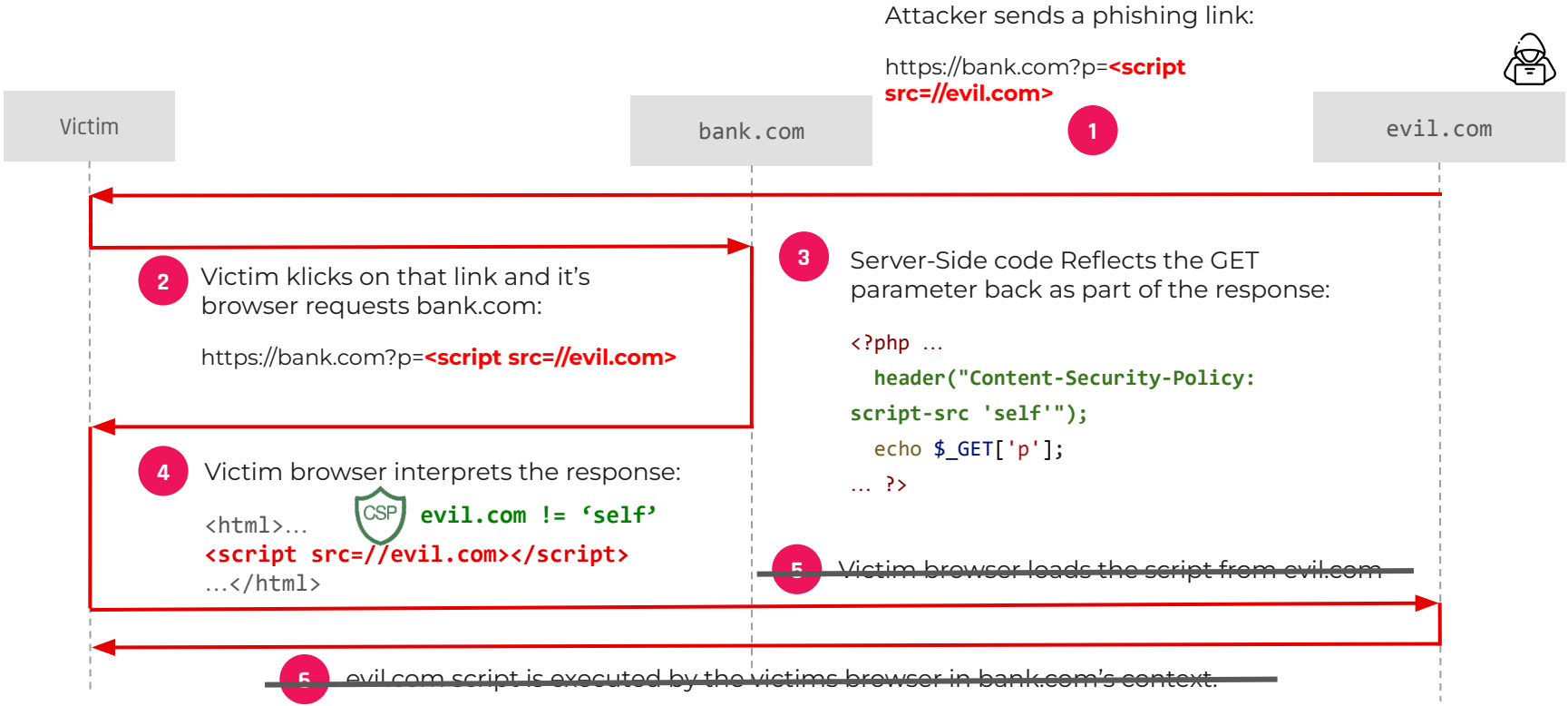
XSS Protection (2/2): Content Security Policy (CSP)

- Defense in depth mechanism: **Content Security Policy** (**CSP**)
  - **HTTP response header** that specifies a **list of allowed resources**, including scripts, styles, and more (original paper: **Reining in the Web with Content Security Policy**)
  - Originally a XSS mitigation, now can restrict framing, mixed-content, form submission, navigations (sort of…), etc.
  - The policy is **enforced by the browser**
  - Example:

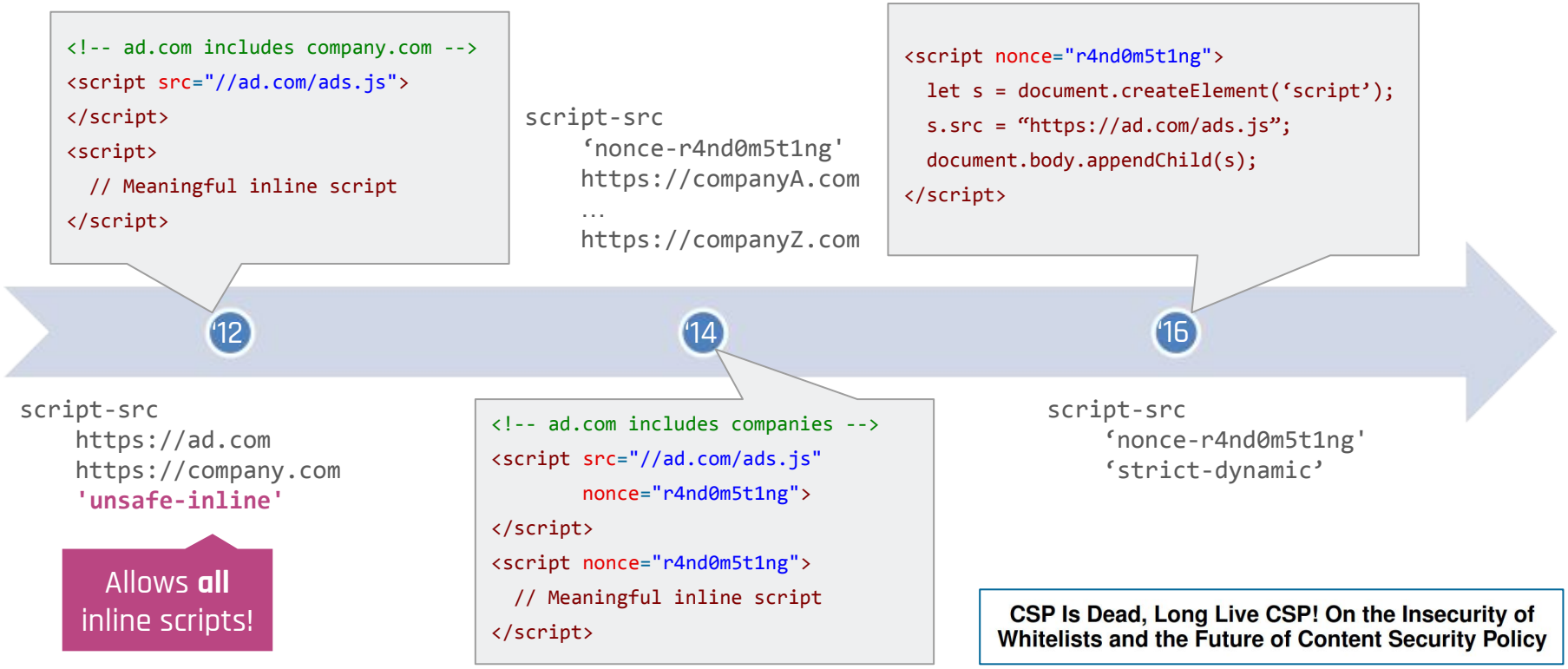
```
Content-Security-Policy:

    script-src
        'self'
        advertisement.com
        'nonce-a7b4f9420'
        'sha256-3i[…]FQ=';
        'strict-dynamic'
```

- Allow scripts from same-origin
- Allow scripts with src hostname advertisement.com
- Allow scripts with nonce attribute set to a7b4f9420
- Allow scripts where the content hash matches
- Allowed scripts can propagate their trust (disables all host-based entities (e.g., 'self', advertisement.com))

Reflected Server-Side XSS (with a CSP)



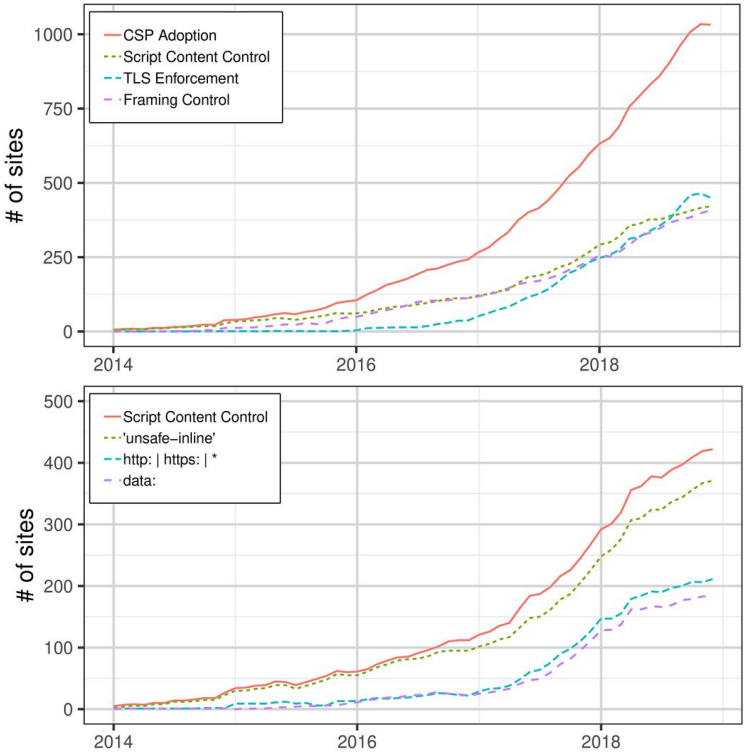
History of the Content Security Policy (CSP)



CSP Is Dead, Long Live CSP! On the Insecurity of Whitelists and the Future of Content Security Policy

Deployed CSPs in the wild

Complex Security Policy?
A Longitudinal Analysis of Deployed Content Security Policies



- The overall adoption of CSP in growing (was at ~10% at the end of 2018).
- The vast majority of all deployed policies is, and has always been, trivially bypassable by an attacker.
- In practice many third parties are mandating the usage of unsafe-inline and unsafe-eval (see

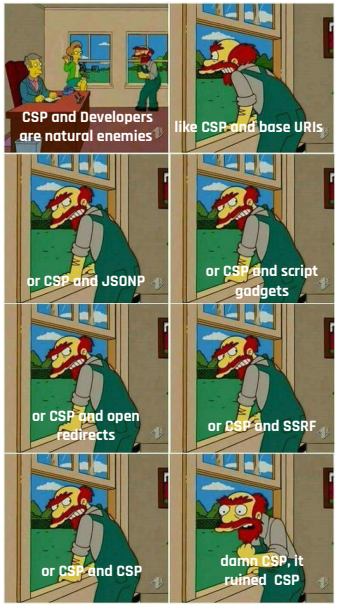
Who's Hosting the Block Party?
Studying Third-Party Blockage of CSP and SRI

- There are a plethora of different roadblocks in the real-world deployment process of CSP (see

12 Angry Developers
A Qualitative Study on Developers' Struggles with CSP

Bypassing a CSP

- There are several ways how even a non-trivially bypassable CSP can be bypassed:
  - Hijacking the Base URI
  - **JSON with padding (JSONP)**
  - Nonce stealing attacks
  - **Code Reuse / Script Gadgets**
  - DOM Clobbering
  - Unrestricted file uploads
  - Missing object-src
  - Open Redirects (seen earlier)
  - SSRF (seen earlier)
  - ...



Hardening your CSP:

- CSP spec says: "[...] when multiple policies are present, each must be enforced or reported, according to its type."
- Usually in case of a CSP directive one of the source-expressions need to match. With multiple policies we can enforce the match of multiple source-expressions
  Example: "scripts need to be nonced and originate from same origin"

```
Content-Security-Policy: script-src nonce-r4nd0m
Content-Security-Policy: script-src 'self'
```

- RFC 2616 says: "It MUST be possible to combine the multiple header fields into one 'field-name: field-value' pair, without changing the semantics of the message, by appending each subsequent field-value to the first, each separated by a comma."

```
Content-Security-Policy: script-src nonce-r4nd0m, script-src 'self'
```

Advanced Exploitation Techniques

postMessage XSS

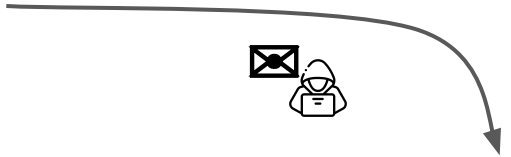
The Postman Always Rings Twice:
Attacking and Defending postMessage in HTML5 Websites

PMForce: Systematically Analyzing
postMessage Handlers at Scale

If the a postMessage handler calls dangerous APIs and only insufficiently (or not at all) checks for the origin of the message, attackers that are loading the page e.g. in an iframe can send messages to inject markup or execute JavaScript:

```
<iframe id="attk" src="https://b.com/></iframe>
<script>
window.attk.postMessage('alert(123)', 'http://b.com');
</script>
```

a.com



```
<script>
window.addEventListener('message', (evt) => {
   eval(evt.data);
})
</script>
```

b.com

postMessage XSS

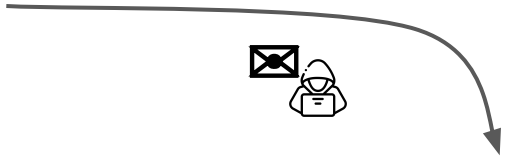
The Postman Always Rings Twice:
Attacking and Defending postMessage in HTML5 Websites

PMForce: Systematically Analyzing
postMessage Handlers at Scale

If the a postMessage handler calls dangerous APIs and only insufficiently (or not at all) checks for the origin of the message, attackers that are loading the page e.g. in an iframe can send messages to inject markup or execute JavaScript:

```
<iframe id="attk" src="https://b.com/></iframe>
<script>
window.attk.postMessage('alert(123)', 'http://b.com');
</script>
```

b.com.a.com



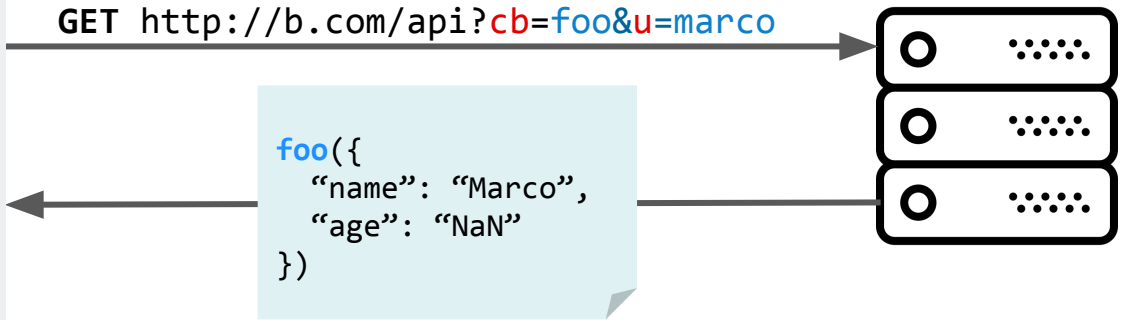
```
<script>
window.addEventListener('message', (evt) => {
   if (evt.origin.startsWith('https://b.com'))
       eval(evt.data);
})
</script>
```

b.com

JSON with Padding (JSONP)

- **JSONP** is a technique to enable cross-origin read. Use **CORS** now please.
- Works by exploiting the fact that **script inclusion is not subject to the SOP!**

```
<!DOCTYPE html>
<body>
   <script>
       function foo(data) {
           console.log(data);
           // and much more
       }
   </script>
   <script
   src="https://b.com/api?cb=foo&
        u=marco"></script>
</body>
</html>
```



JSON with Padding (JSONP)

```
Content-Security-Policy: script-src accounts.google.com
```

```
<body>
   <h1>JSONP Injection</h1>
   Hello <script src='https://accounts.google.com/o/oauth2/revoke?callback=alert(1)'>!
   </body>
```

Code Reuse Attacks / Script Gadgets

- Many websites use very popular (and complex) JS frameworks like AngularJS, React, Vue.js, Aurelia, jQuery, etc.
- These frameworks contain **script gadgets**, pieces of JavaScript that **react** to the presence of specifically crafted DOM elements
- Script Gadgets convert otherwise safe HTML tags and attributes into **arbitrary JavaScript code execution**, turning **any markup injections into full XSS**!

Example:

```
// framework.js
...
var btns = document.querySelectorAll("[data-role=button]");
for (var b of btns) {
      // Style the button
      b.innerHTML = b.getAttribute("data-text")
}
...
```

```
<div data-role="button" data-text="Submit!"></div>
```

```
<div data-role="button"
     data-text="<img onerror=alert(1)>"></div>
```

Code Reuse Attacks / Script Gadgets

```
Content-Security-Policy:
script-src gstatic.com ‘unsafe-eval’
```

```
<body>
   Hello <script
            src="https://gstatic.com/angular.js">
   </script>
   <div ng-app>
      {{constructor.constructor('alert(“XSS”)')()}}
   </div>
</body>
```

Code-Reuse Attacks for the Web: Breaking Cross-Site Scripting Mitigations via Script Gadgets

- **Script Gadgets** are based on **different execution methods**
  - `eval()`
  - `innerHTML`, …
  - Non-eval based expression parsers that tokenize, parse & evaluate the expressions on their own

[POCs] >
https://github.com/google/security-research-pocs/tree/master/script-gadgets

Trusted Types

- New API to **obliterate DOM XSS**
- **Idea:**
  - Lock down dangerous **injection sinks** so that they **cannot be called with strings**
  - **Interaction** with those functions is only permitted via **special (trusted) typed objects**
  - Those objects can be created only inside a **Trusted Type Policy**, created in the JavaScript code part of an web application)
  - **Policies are enforced** by setting the `trusted-types` directive in the **CSP**
  - Ideally, TT-enforced applications are **secure by default** and the only code that could introduce a DOM XSS vulnerability is in the policies

Trusted Types

- Identified >60 different injection sinks
- 3 possible Trusted Types
  - **`TrustedHTML`**
    strings that can be safely inserted into injection sinks (e.g., innerHTML) and rendered as HTML. Constructed via the `createHTML` method.
  - **`TrustedScript`**
    string with a script body that a developer can safely pass into an injection sink (e.g., eval) that may execute that script. Constructed via the `createScript` method.
  - **`TrustedScriptURL`**
    string with a URL that a developer can safely pass into an injection sink that will parse it as a URL of an external script resource. Constructed via the `createScriptURL` method.

Trusted Types

Problem: If a third party mandates unsafe-eval we are doomed -> **Solution:** Trusted Types

```
Content-Security-Policy: trusted-types ttpolicy; require-trusted-types-for 'script';
```

vulnerable.js

```
window.addEventListener('load', function () {
    let d = document.createElement('div');
    var name = unescape(location.hash.slice(1));
    d.innerHTML = ttpolicy.createHTML(name);
    document.body.appendChild(d);
});
```

trusted-types.js

```
if (window.trustedTypes && trustedTypes.createPolicy) {
    trustedTypes.createPolicy('ttpolicy', {
        createHTML: function(html_string) {
            return sanitizeHTML(html_string);
        },
        createScript: function(js_string) {
            return sanitizeJS(js_string);
        },
        createScriptUrl: function(url) {
            return checkURL(url);
        },
    });
}
```

Problem: WebApps have a huge codebase and also run third party code that can not be changed...

Trusted Types

Solution: To avoid refactoring (and also secure third party code) Trusted Types supports a default sanitizer!

```
Content-Security-Policy: trusted-types default; require-trusted-types-for 'script';
```

vulnerable.js

```
window.addEventListener('load', function () {
    let d = document.createElement('div');
    d.innerHTML = unescape(location.hash.slice(1));
    document.body.appendChild(d);
});
```

inherently calls the responsible sanitizer!

trusted-types.js

```
if (window.trustedTypes && trustedTypes.createPolicy) {
    trustedTypes.createPolicy('default', {
        createHTML: function(html_string) {
            return sanitizeHTML(html_string);
        },
        createScript: function(js_string) {
            return sanitizeJS(js_string);
        },
        createScriptUrl: function(url) {
            return checkURL(url);
        },
    });
}
```

Problem: Developers are responsible for writing the sanitizer functions -> Hard to deploy (see Trust Me *If You Can* How Usable Is Trusted Types In Practice?)

Trusted Types: Pitfalls

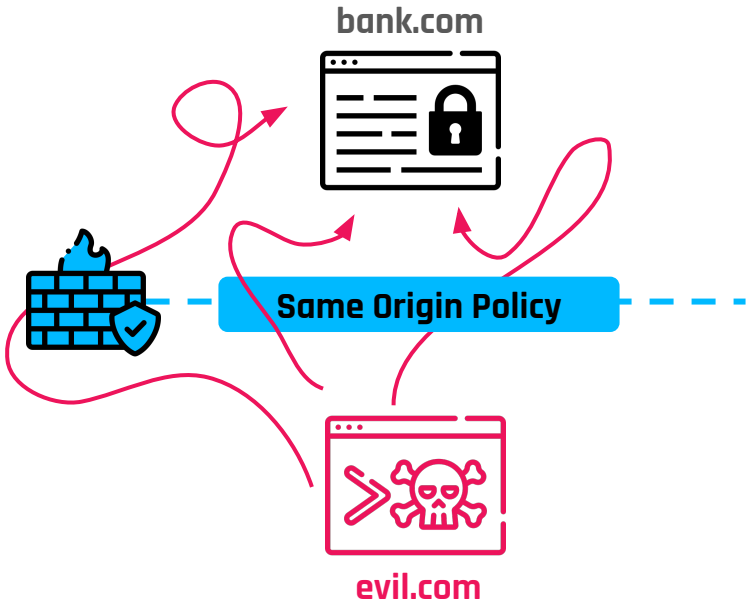
- Sanitisation is left as an exercise to the policy writers
  - W3C is working on the **Sanitizer API**! However, it's only for HTML and currently it blindly removes all occurrences if JS in the HTML code (unusable in practice).
- None client-side XSS could lead to a bypass of the policy restrictions
  - … so you have to combine Trusted Types with a proper CSP
- Third Party behaviour might be incompatible with Trusted Types
- Policies are custom JavaScript code that may depend on the global state
- Colluding same-origin pages: complete bypass of Trusted Types
- For more Roadblocks of Trusted types see

Trust Me *If You Can*
How Usable Is Trusted Types In Practice?

XS-Leaks

Gaps in the SOP = Cross-Site Leaks

- Thanks to the **SOP**, attackers cannot access cross-origin resources directly
- But they can exploit **browser side-channel techniques** to infer and gather information about users, usually via a **boolean oracle**
- These oracles are built by exploiting **subtleties in the web platform**



Gaps in the SOP = Cross-Site Leaks

Browser state changes can be monitored using techniques to infer the size and status of HTTP responses

- `https://mail.google.com/mail/u/0/#search/credit+card+5400` QUICK (or 404)
- `https://mail.google.com/mail/u/0/#search/credit+card+5401` QUICK (or 404)
- `https://mail.google.com/mail/u/0/#search/credit+card+5402` **SLOW!** (or 200)

Most dangerous setting for an XS-Leak vulnerability: **XS-Search**

XS-Leak Example: Error Events

Attacker-controlled page visited by the victim

```
<body>
   <script>
       function probeError(url) {
           let script = document.createElement('script');
           script.src = url;
           script.onload = () => console.log('Onload event triggered');
           script.onerror = () => console.log('Error event triggered');
           document.head.appendChild(script);
       }
       // because google.com/404 returns HTTP 404, the script triggers error event
       probeError('https://google.com/404');
       // because google.com returns HTTP 200, the script triggers onload event
       probeError('https://google.com/');
   </script>
</body>
```

Recently fixed in Chrome, FF is still vulnerable

Can be used to **detect whether a user is logged in to a service** by checking if the user has access to resources only available to authenticated users.

XS-Leak Example: Frame Counting

```
// Get a reference to the window
var win = window.open('https://www.linkedin.com');
// Wait for the page to load
setTimeout(() => {
 // Read the number of iframes loaded
 console.log("%d iframes detected", win.length);
}, 2000);
```

- Also the **number of iframes** can reveal the authentication state of a user on a website

Lax cookies are still sent, since `window.open` is a top-level navigation!

- But we have **SameSite cookies**, right?
- More advanced attacks on **https://xsleaks.dev/**

XS-Leaks Defenses

- **Defending against XS-Leaks is difficult**
  - Countless attack vectors
  - Most powerful defenses are opt-in and not trivial to deploy
  - Lack of browser support for all the protections

- So difficult that many companies are not even paying bug bounties for this class of vulnerabilities, despite its impact

- The Web platform is **insecure by default**. Shifting towards secure defaults is a long process that requires to break **backward compatibility**.

XS-Leaks Defenses

- **Fetch Metadata**
  request headers sent by the browser explaining why a request was initiated (e.g., is the request **same-site or same-origin**?) Servers can take informed decisions & block

- **Framing Protections**
  (via XFO, CSP) applications can define what sites are allowed to frame them

- ...

More at **https://xsleaks.dev/**

XS-Leaks Defense: Frame Counting + COOP

- **Cross-Origin-Opener-Policy (COOP)**
  prevent other **origins** from interacting with an application via `window.open/window.opener`

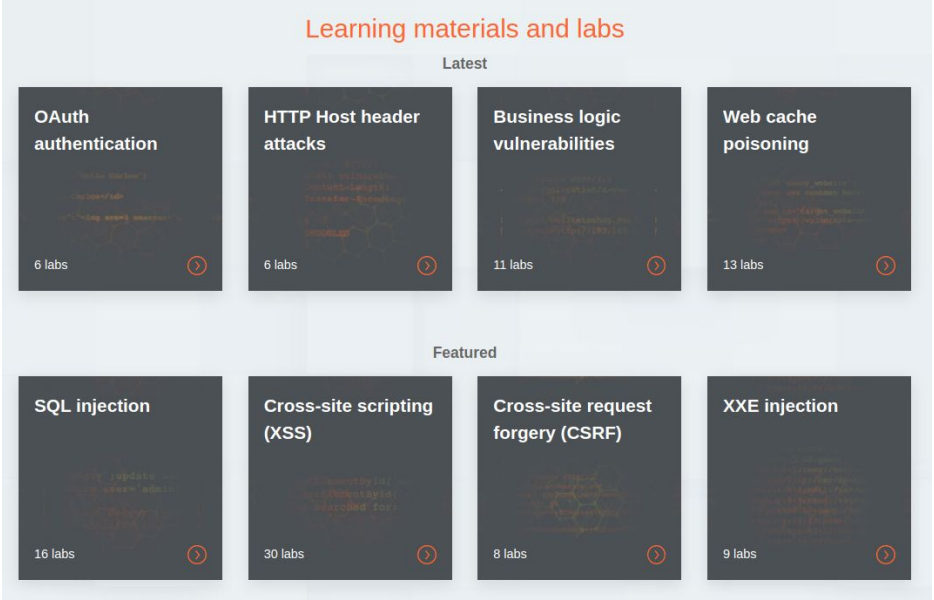
```
// Get a reference to the window
var win = window.open('https://example.org');
// Wait for the page to load
setTimeout(() => {
  // Read the number of iframes loaded
  console.log("%d iframes detected", win.length);
}, 2000);
```

- **`win.opener`** is set to **`null`**
- **`win.length`** returns always **`0`**

Assume `example.org` to ship with **`Cross-Origin-Opener-Policy: same-origin`**

Train Yourself

Most of these attacks are covered by
https://portswigger.net/web-security/all-labs



Takeaways (or The Cursed Web, again)

- Securing the Web is a complex task
- Browser are becoming the new Operating Systems
- Web developers are (still) provided with too many footguns and security knobs
- Removing legacy insecure mechanisms from the Web is difficult (<0.0X%)
- Understanding what fixes what under which conditions is far from trivial
- Complex standards, sometimes contradictory, or overlapping
- Huge gaps between standards and real-world implementations
- How do standards combine? Lack of formal methods
- Advancement in the Web platform in the hands of a few companies
- Lack of competition, e.g., iOS

Thank You!
Q&A

Sebastian **Roth** <sebastian.roth@tuwien.ac.at>
Marco **Squarcina** <marco.squarcina@tuwien.ac.at>

Note: In this course we discussed around 1% of Web security