#HackInBo

Bologna, Italy

2019

The web is broken

Let's fix it!

Roberto Clapis

Michele Spagnuolo

HackInBo Winter 2019 Edition

13ª EDIZIONE

Roberto Clapis

Software Engineer
(Security)

Michele Spagnuolo

Senior Information Security
Engineer

We work in a focus area of the **Google** security team (ISE) aimed at **improving product security** by targeted proactive projects to **mitigate whole classes of bugs**.

HackInBo Winter 2019 Edition
13ª EDIZIONE

What is Cross-site scripting (XSS)?

A web vulnerability that enables attackers to **run malicious scripts** in users' browsers in the **context** of the vulnerable origin

- **Server-side**
  - **Reflected XSS**: an attacker can change parts of an HTML page displayed to the user via sources they control, such as request parameters
  - ...
- **Client-side**
  - **DOM-based XSS**: using unsafe DOM methods in JS when handling untrusted data
  - ...

Manual escaping is not a solution

- **Not secure-by-default**

- **Hard** and **error-prone**
  - Different rules for different contexts
    - HTML
    - CSS
    - JS
    - XML-like (SVG, ...)

- **Unsafe DOM APIs** are out there to be (ab)used
  - Not just `innerHTML`!

location.open
HTMLFrameElement.srcdoc
HTMLMediaElement.src
HTMLScriptElement.InnerText
HTMLInputElement.formAction
document.write
location.href
HTMLSourceElement.src
HTMLAreaElement.href
HTMLInputElement.src
Element.innerHTML
HTMLFrameElement.src
HTMLBaseElement.href
HTMLTrackElement.src
HTMLButtonElement.formAction
HTMLScriptElement.textContent
HTMLImageElement.src
HTMLFormElement.action
location.assign
HTMLEmbededElement.src

A better solution: templating systems + safe APIs

- Templating systems with **strict contextual escaping**
    - **Java:** Google Closure Template/Soy
    - **Python:** Google Closure Template/Soy, recent Django (avoid `|safe`)
    - **Golang:** safehtml/template, html/template
    - **Angular** (Angular2+): TypeScript with ahead of time compilation (AoT)
    - **React:** very difficult (but not impossible) to introduce XSS
- Safe-by-default APIs
    - Use wrapping "**safe types**"
        - JS **Trusted Types** coming in Chromium

https://github.com/w3c/webappsec-trusted-types/
https://github.com/w3c/webappsec-trusted-types/wiki/Integrations

The idea behind Trusted Types

Source → ... → Policy → Trusted Type → ... → DOM sink

When Trusted Types are **enforced**:

```
Content-Security-Policy: trusted-types myPolicy
```

DOM sinks **reject strings**:

```
element.innerHTML = location.hash.slice(1); // a string
```

```
Uncaught TypeError: Failed to set the 'innerHTML' property on 'Element': This document requires `TrustedHTML` assignment.    demo2.html:9
    at demo2.html:9
```

DOM sinks **accept only typed objects**:

```
element.innerHTML = aTrustedHTML; // created via a TrustedTypes policy
```

The need for Defense-in-Depth

- **XSS** in its various forms is still a big issue
- The web platform is **not secure-by-default**
- Some XSS (especially DOM-based) are **very hard to prevent**
- **Defense-in-depth** is very important in case primary security mechanisms fail

Mitigation ≠ Mitigation

Reducing the attack surface

vs

- Measurable security improvement
- Disable unsafe APIs
- Remove attack vectors
- Target classes of bugs
- Defense-in-depth (Don't forget to fix bugs!)

Example:

- block eval() or javascript: URI
  ⟶ all XSS vulnerabilities using that sink will stop working
- nonce-based CSP

"raising the bar"

- Increase the "cost" of an attack
- Slow down the attacker

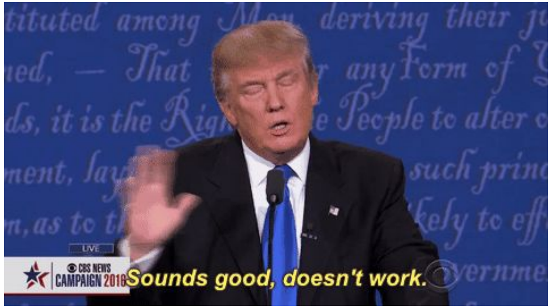
Example:

- whitelist-based CSP
  ⟶ sink isn't closed, attacker needs more time to find a whitelist bypass
  ⟶ often there is no control over content hosted on whitelisted domains (e.g. CDNs)

CSP is also hardening!

- Refactor inline event handlers
- Refactor uses of eval()
- Incentive to use contextual templating system for auto-noncing

Why NOT a whitelist-based CSP?




```
script-src 'self' https://www.google.com;
```

TL;DR Don't use them! They're almost always trivially bypassable.

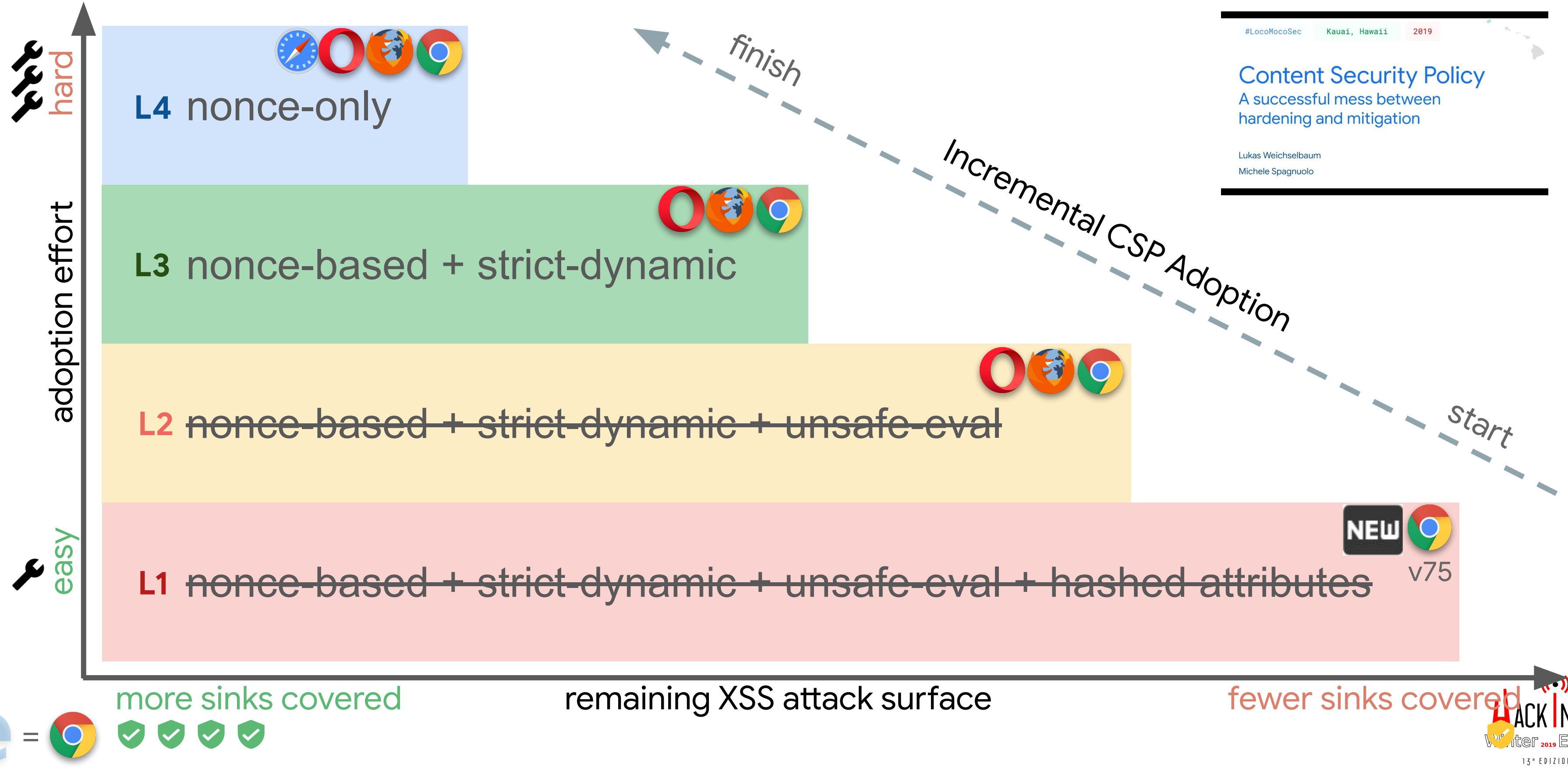
- >95% of the Web's whitelist-based CSP are bypassable automatically
  - Research Paper: https://ai.google/research/pubs/pub45542
  - Check yourself: http://csp-evaluator.withgoogle.com
  - The remaining 5% might be bypassable after manual review
- Example: JSONP, AngularJS, ... hosted on whitelisted domain (esp. CDNs)
- Whitelists are hard to create and maintain → breakages

More about CSP whitelists:
ACM CCS '16, IEEE SecDev '16, AppSec EU '17, Hack in the Box '18,

Reducing the attack surface with CSP

In-depth talk:

[Content Security Policy - A successful mess between hardening and mitigation]



What is a CSP nonce?

Content-Security-Policy:

```
script-src 'nonce-r4nd0m' 'strict-dynamic';
object-src 'none'; base-uri 'none';
```

Execute only scripts with the correct *nonce* attribute

```
✓ <script nonce="r4nd0m">kittens()</script>
✗ <script nonce="other-value">evil()</script>
```

Trust scripts added by already trusted code

```
✓<script nonce="r4nd0m">
   var s = document.createElement('script')
   s.src = "/path/to/script.js";
✓  document.head.appendChild(s);
 </script>
```

The Easy Way: nonce-based + strict-dynamic

soon

```
script-src 'nonce-r4nd0m' 'strict-dynamic';
object-src 'none'; base-uri 'none';
```

Refactoring steps:

```
<html>
 <a href="javascript:void(0)">a</a>
 <a onclick="alert('clicked')">b</a>
 <script src="stuff.js"/>
 <script>
  var s =
    document.createElement('script');
  s.src = 'dynamicallyLoadedStuff.js';
  document.body.appendChild(s);
  var j = eval('(' + json + ')');
 </script>
</html>
```

```
<html>
 <a href="#">a</a>
 <a id="link">b</a>
 <script nonce="r4nd0m" src="stuff.js"/>
 <script nonce="r4nd0m">
  var s = document.createElement('script');
  s.src = 'dynamicallyLoadedStuff.js'
  document.body.appendChild(s);
  document.getElementById('link')
    .addEventListener('click', alert('clicked'));
 var j = JSON.parse(json);
 </script>
</html>
```

soon

The Easy Way: nonce-based + strict-dynamic

```
script-src 'nonce-r4nd0m' 'strict-dynamic';
object-src 'none'; base-uri 'none';
```

TL;DR Good trade off between refactoring and covered sinks.

PROs:

+ Reflected/stored XSS mitigated
+ Little refactoring required
  - `<script>` tags in initial response must have a valid `nonce` attribute
  - inline event handlers and javascript: URIs must be refactored
+ Works if you don't control all JS
+ Good browser support

CONs:

- DOM XSS partially covered
  - e.g. injection in dynamic script creation possible

The Better Way: nonce-only

soon

```
script-src 'nonce-r4nd0m';
object-src 'none'; base-uri 'none';
```

Refactoring steps:

```html
<html>
 <a href="javascript:void(0)">a</a>
 <a onclick="alert('clicked')">b</a>
 <script src="stuff.js"/>
 <script>
  var s =
    document.createElement('script');
  s.src = 'dynamicallyLoadedStuff.js';
  document.body.appendChild(s);
 </script>
</html>
```

```html
<html>
 <a href="#">a</a>
 <a id="link">b</a>
 <script nonce="r4nd0m" src="stuff.js"/>
 <script nonce="r4nd0m">
  var s = document.createElement('script');
  s.src = 'dynamicallyLoadedStuff.js'
  s.setAttribute('nonce', 'r4nd0m');
  document.body.appendChild(s);
  document.getElementById('link')
    .addEventListener('click', alert('clicked'));
 </script>
</html>
```

soon

The Better Way: nonce-only

```
script-src 'nonce-r4nd0m';
object-src 'none'; base-uri 'none';
```

TL;DR Holy grail! All traditional XSS sinks covered, but sometimes hard to deploy.

PROs:

+ Best coverage of XSS sinks possible in the web platform
+ Supported by all major browsers
+ Every running script was explicitly marked as trusted

CONs:

- Large refactoring required
  - **ALL** `<script>` tags must have a valid `nonce` attribute
  - inline event-handlers and javascript: URIs must be refactored
- You need be in control of all JS
  - all JS libs/widgets must pass nonces to child scripts

Nonce-only is great!

soon

```
script-src 'nonce-r4nd0m';
object-src 'none'; base-uri 'none';
```

XSS Sinks Covered:

javascript: URI	✓
data: URI	✓
(inner)HTML context	✓
inline event handler	✓
eval	✓
script#text	✓ (✘ if untrusted script explicitly marked as trusted)
script#src	✓ (✘ if untrusted URL explicitly marked as trusted)

CSP in brief

Use a **nonce-based CSP with strict-dynamic:**

```
script-src 'nonce-r4nd0m' 'strict-dynamic';
object-src 'none'; base-uri 'none';
```

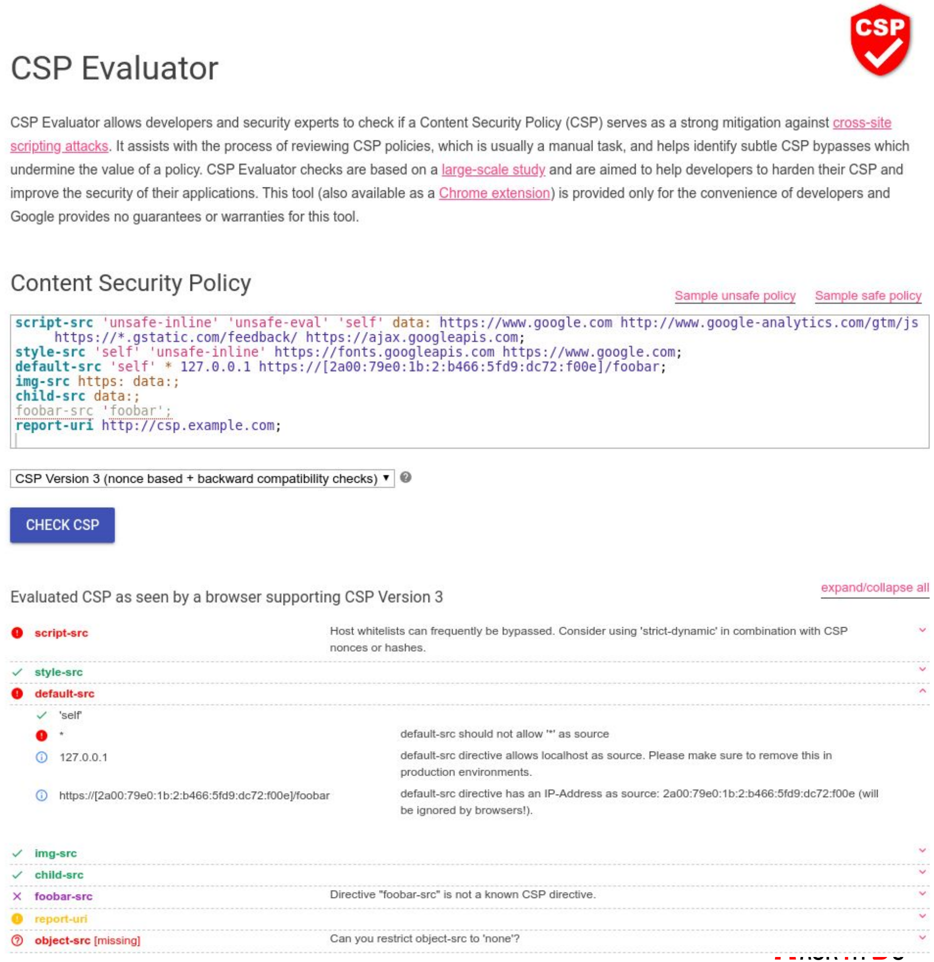
If possible, upgrade to a **nonce-only CSP**:

```
script-src 'nonce-r4nd0m';
object-src 'none'; base-uri 'none';
```

CSP tools & resources

- How to adopt an effective CSP in your web app: csp.withgoogle.com

- Always double check your CSP with the **CSP Evaluator:** csp-evaluator.withgoogle.com



XSS done, everything else to go...

HackInBo Winter 2019 Edition
13ª EDIZIONE

Cross site request forgery (CSRF/XSRF)

- Client-side example form:

```
<form enctype="application/x-www-form-urlencoded" method="POST"
        action="https://store.google.com">
  <input type="text" name="action" value="buy_product">
  <input type="text" name="quantity" value="1000">
  <input type="submit" value="https://store.google.com">
</form>
```

- What the server sees when user submits:
  - cookies
  - `action=buy_product`
  - `quantity=1000`
- There is no secure notion of **web origin**

HackInBo® Winter 2019 Edition 13ª EDIZIONE

Cross site request forgery (CSRF/XSRF)

- It's been there since the beginning
- **It's clumsy to address**
- Requires developers to add custom protections **on top of the platform**
- Normally addressed by adding tokens in hidden forms parameters
- It is not clear what to protect, so even using frameworks might lead to issues

Example: GET requests are usually not protected by frameworks but developers might decide to have **state-changing** APIs that use **GET** parameters, or some libraries might automatically parse GET forms and treat them as POST. If this happens **after the CSRF middleware runs** the vulnerability is still there.

Same Site Cookies

- Simple **server-side** CSRF mitigation mechanism

```
Set-Cookie: <name>=<value>; SameSite=(Lax|Strict);
```

- **Lax** allows cross-site navigation (default since Chromium 80)
- **Strict** prevents cookies from being sent in any cross-site action

Cross site leaks (XS-Leaks)

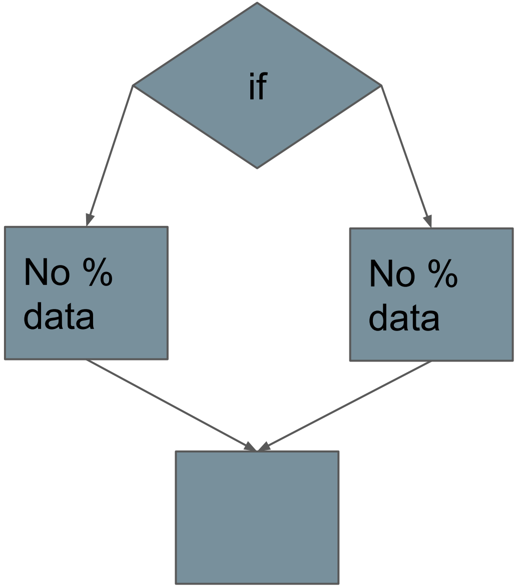
- Extract bits of information via **side channels**
- The attacking page doesn't need to see the cross-origin content, just the **time** it took to load, or the **error** that happened while trying to load
- **Same-origin policy does not protect** against this kind of attacks

For example, **login detection**: loading a frame errors if user is not logged in.

Spectre

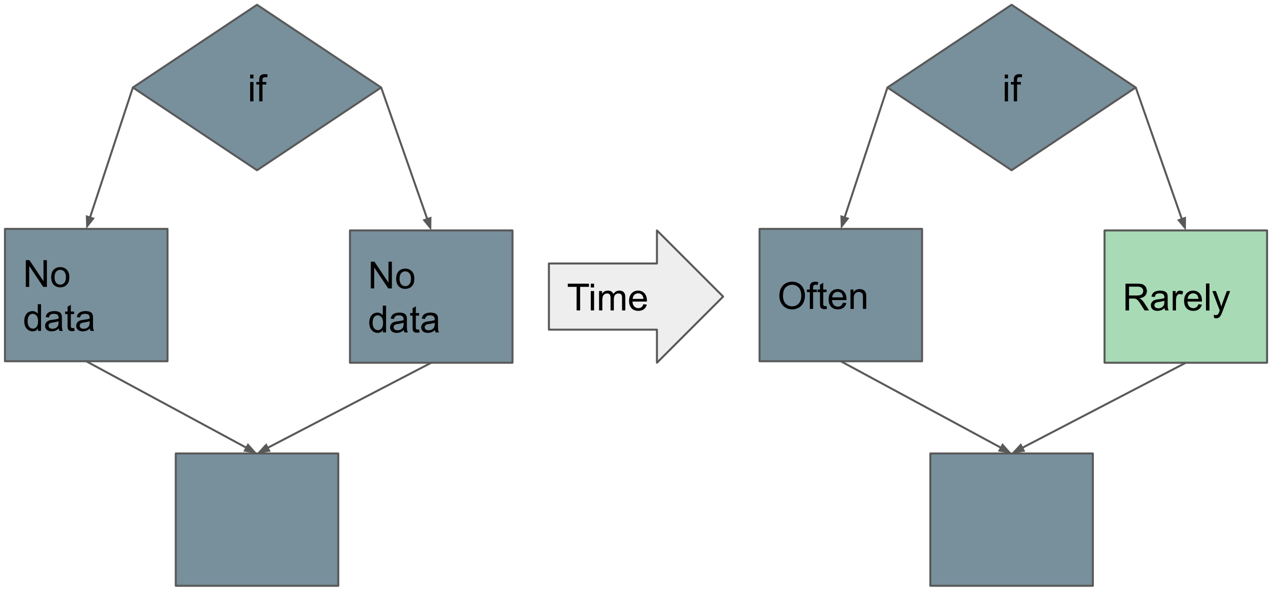
- Extract bits of information via hardware issues
- Get around Same-Origin policy because the memory is in the same process, and it can be accessed via **side-channels**
- Requires precise timers, but they can be crafted

Spectre



First execution

Spectre



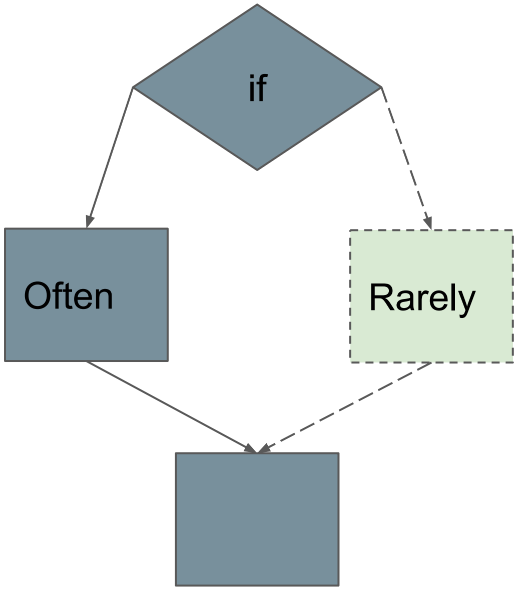
First execution

Many executions

Spectre

- After many executions the CPU will start **spec**ulating which **branch** should be taken, and will execute it **before the if conditions computed**

- **Some side effects of this can be inspected**



Many executions

Spectre, an example

Run many times with small indexes, then with `controlled_index` > `max_index`

```
if (controlled_index < max_index) {
    secret_value = index_array[controlled_index];
    _ = data_array[secret_value*cache_block_size];
}
```

Measure access time to different blocks of data_array

The one in **`secret_value`** position will be **faster to access**

How do you get stuff in memory?

The legacy of Same Origin Policy

```
<script
   src=https://vulnerable.com/interesting_data>
</script>
```

```
<img
   src=https://vulnerable.com/interesting_data>
</img>
```

COR{B,P}

Cross Origin Read Blocking

On by default, but it is a heuristic

Cross-Origin-Resource-Policy

Enforces CORB and provides more protection

How do you **NOT** get stuff in memory?

Fetch Metadata

- Three `Sec-Fetch-*` **request headers**
  - `-Mode` (cors, navigate, no-cors, same-origin, websocket...)
  - `-Site` (cross-site, same-origin, same-site, none)
  - `-User` (boolean)
- **Servers** can now make **informed decisions** whether to provide the requested resource

Sample HTTP request headers

```
GET /?do=action HTTP/1.1
Sec-Fetch-Mode: no-cors
Sec-Fetch-Site: cross-site
```

The code

```
func Allowed(r *http.Request) bool {
    site := r.Header.Get("sec-fetch-site")
    mode := r.Header.Get("sec-fetch-mode")
    if site != "cross-site" {
        return true
    }
    if mode == "navigate" && req.Method == "GET" {
        return true
    }
    return false
}
```

Find a reference module here:
github.com/empijei/go-sec-fetch

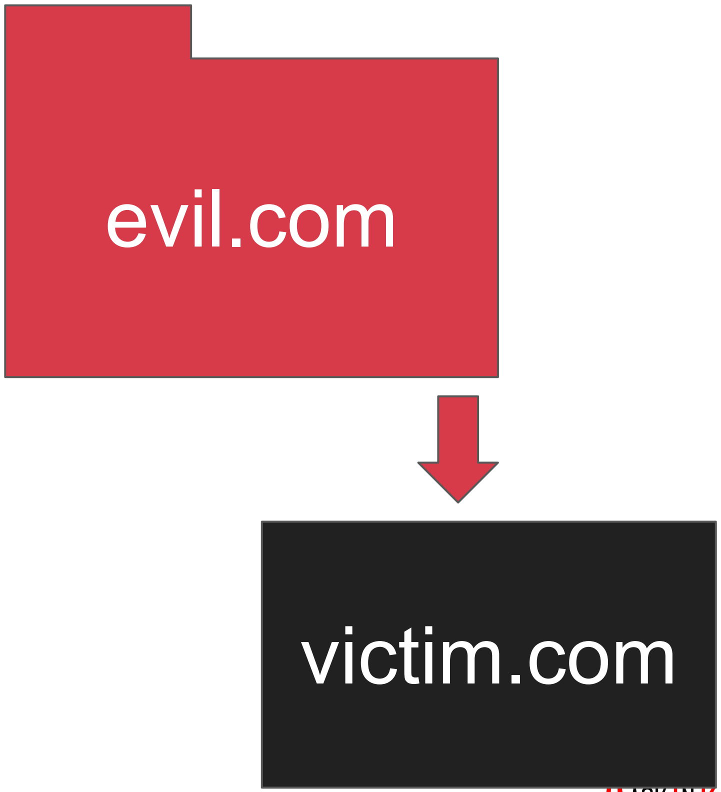
Once we block resources...

XS-Leaks: Cross site search (XSSearch)

- A notable example of cross-site leaks
- Extract bits of information from **the time it takes to load search results**
- In 2016 this affected **GMail** and **Bing** to a point where **credit cards could be stolen** in less than 45s and the **full search history** in less than 90s

Cross-site search

- Open a window to `victim.com/?q=search_term`
- Navigate it many times with different search terms and measure timing, or count frames, or read history length...
- Leak data



We could you CSRF tokens but...

Very complicated to add to GETs

Would break some functionalities

Bookmarks would stop working

Lowers caches efficacy

Even if we did...

Tabnabbing

- **Phishing attack** that relies on navigations that the user does not expect
- Example:
    - User clicks on a link on GMail
    - The link opens a new tab
    - The originating page (**gmail.com**) gets redirected to a phishing clone (**gmai1.com**) asking for credentials
    - When the user closes the new tab, they will go back to the previous context and **expect it to still be GMail**
    - User inputs credentials in **gmai1.com**

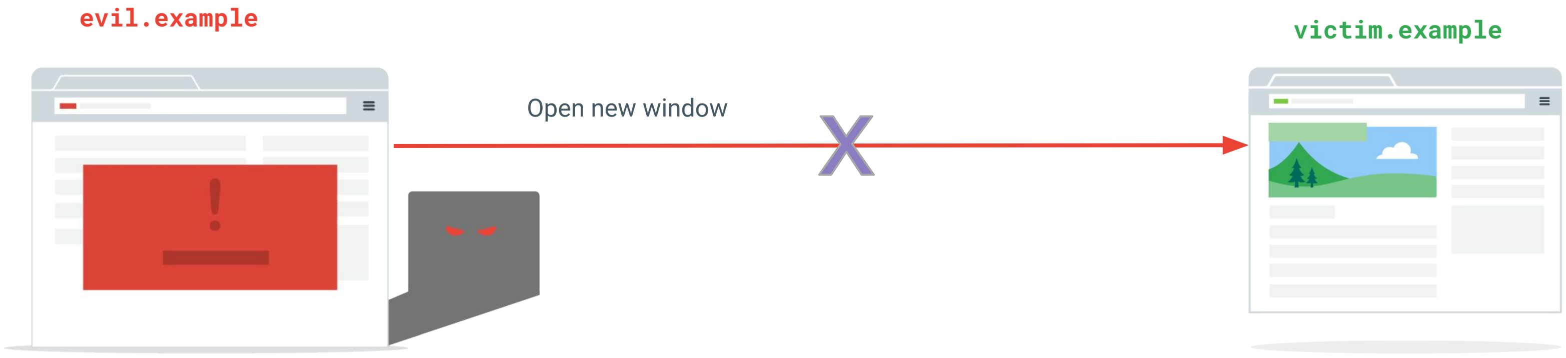
HackInBo Winter 2019 Edition 13ª EDIZIONE

How do we fix it?

Cross Origin Opener Policy

- Dictates top-level navigation cross-origin behavior
- Addresses attacks that rely on cross-window actions
- **Severs the connection between windows** during navigation

```
Cross-Origin-Opener-Policy: "same-origin"
```



What about the first navigation?

Double-Keyed Caches

Navigations can still leak bits of information, even with

Vary: Sec-Fetch-Site

If a resource is loaded by a page (e.g. profile picture) it is brought in cache, and it is thus measurably faster to load

This could identify Twitter users by using a divide-and-conquer approach (silhouette attack)

Double-Keyed-Caches use the origin that **requested the data** as secondary key.

Recap

```
Content-Security-Policy:
    script-src 'nonce-r4nd0m' 'strict-dynamic'; object-src
'none'; base-uri 'none';

Cross-Origin-Opener-Policy: same-origin

Cross-Origin-Resource-Policy: same-origin
```

+

a Fetch Metadata policy

Mahalo! 🤙

Questions?

You can find us at:

{clap,mikispag}@google.com

@empijei, @mikispag

Slides:
clap.page.link/fixtheweb