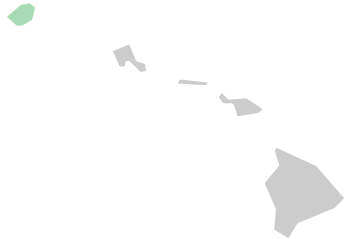
#LocoMocoSec Kauai, Hawaii 2019



# Content Security Policy

## A successful mess between hardening and mitigation

Lukas Weichselbaum

Michele Spagnuolo

**Lukas Weichselbaum**

Staff Information Security Engineer

**Michele Spagnuolo**

Senior Information Security Engineer

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

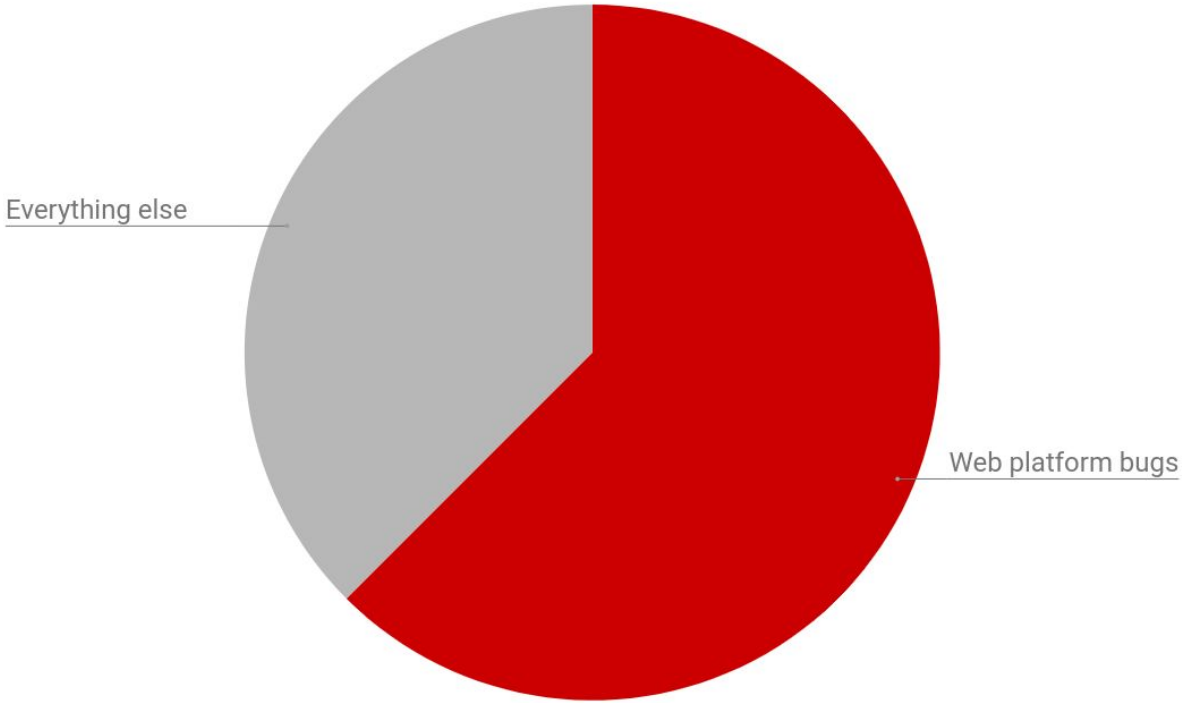
# Agenda

- Why CSP - aka XSS is still an issue
- Google CSP stats - how many XSS got mitigated in 2018
- CSP building blocks - mapping XSS sinks to CSP properties
- Rolling out a nonce-based CSP
- Advanced CSP Kung Fu
- Productionizing CSP

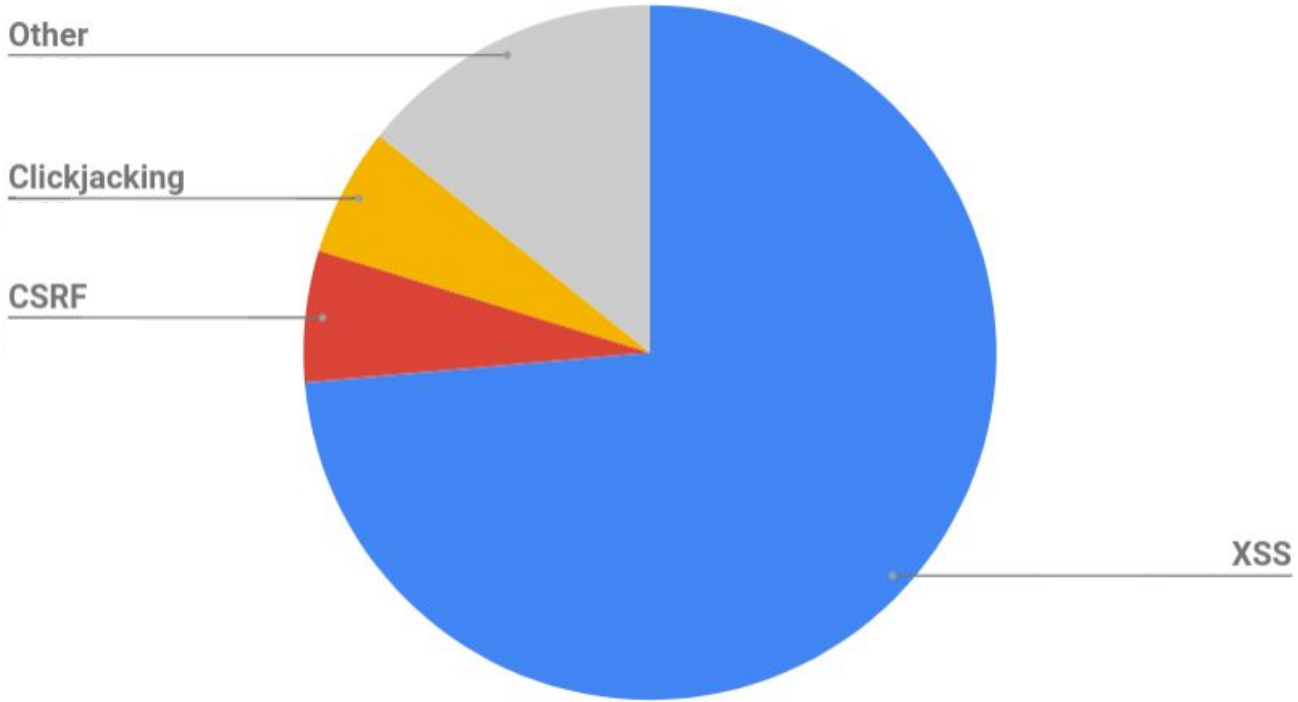
# Vulnerability Trends

## Why you should care about XSS

# Total Google VRP Rewards (since 2014)



# Google VRP Rewards for Web Platform Bugs



## VULNERABILITIES BY INDUSTRY

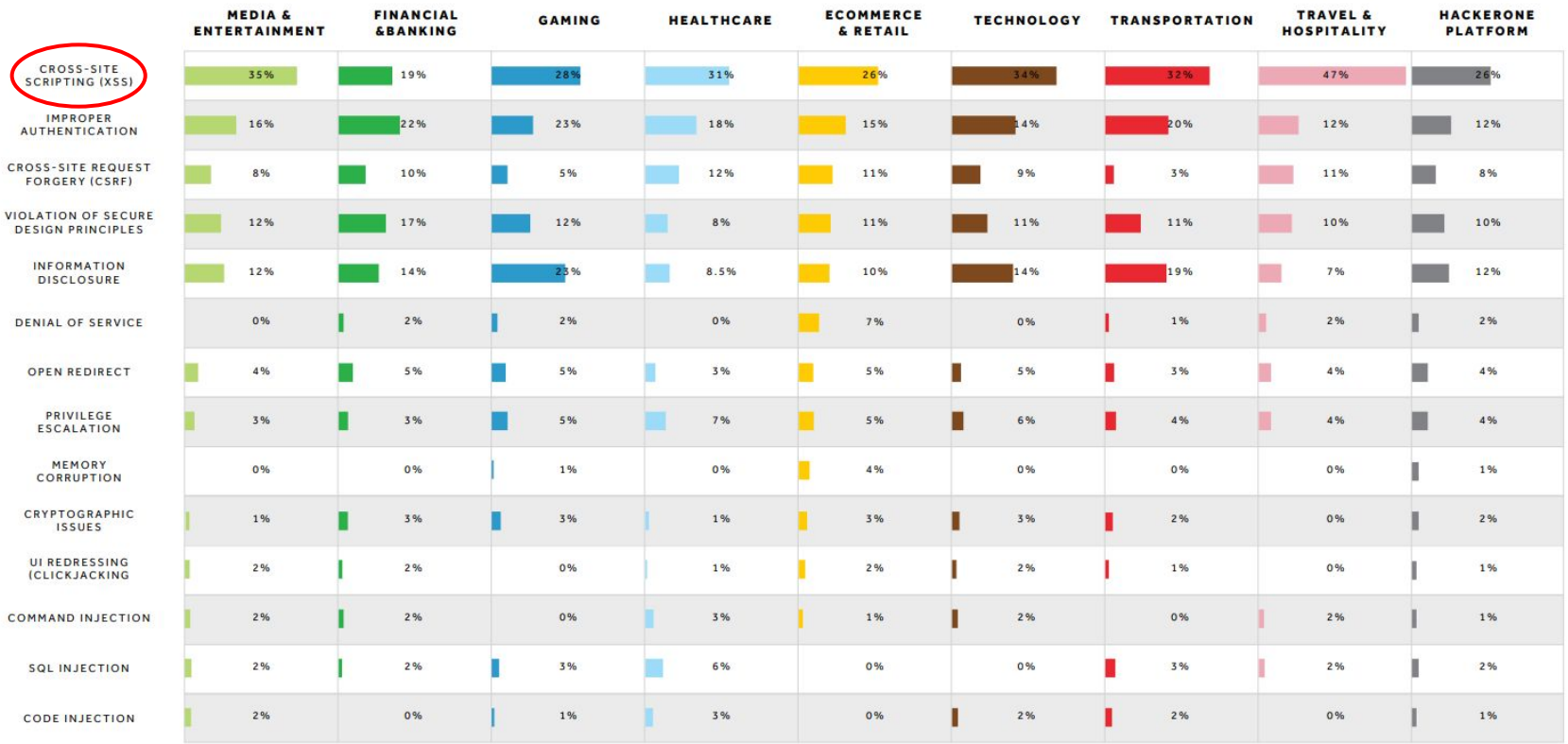
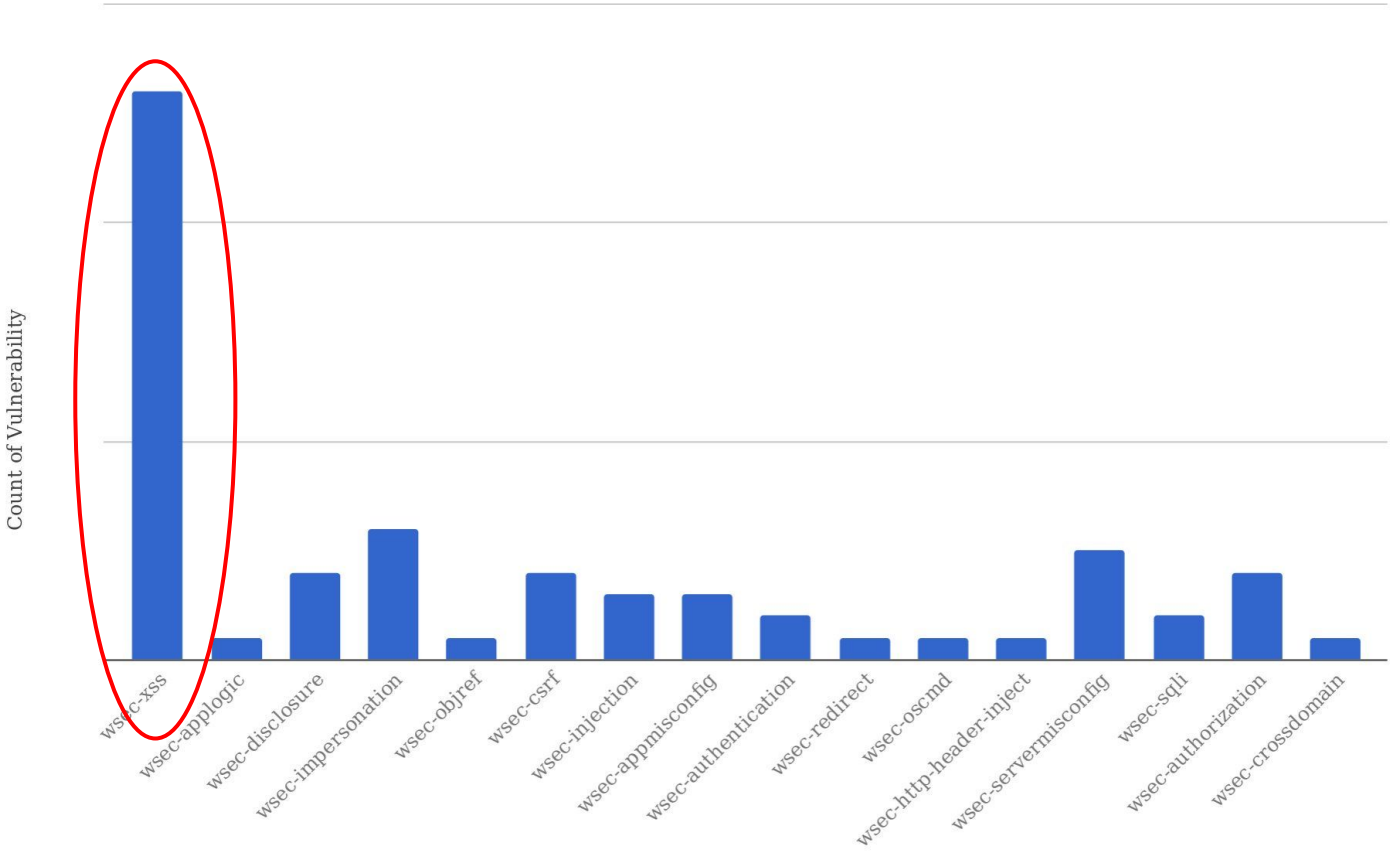



**Figure 2:** *Percentage of vulnerability type by industry from 2013 to May 2017.*

Source: HackerOne report, 2017

Paid bounties by vulnerability on Mozilla websites in 2016 and 2017




Source: @jvehent, Mozilla (legend)

# The Need for Defense-in-Depth

- The majority of application vulnerabilities are web platform issues
- **XSS** in its various forms is still a big issue
- The web platform is not secure by default
- Especially for sensitive applications, defense-in-depth mechanisms such as CSP are very important in case primary security mechanisms fail

# Mitigation ≠ Mitigation

## Reducing the attack surface

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

Example:

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

VS

## "raising the bar"

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

Example:

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

## Hardening Steps induced by CSP

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

# XSS blocked by CSP @Google

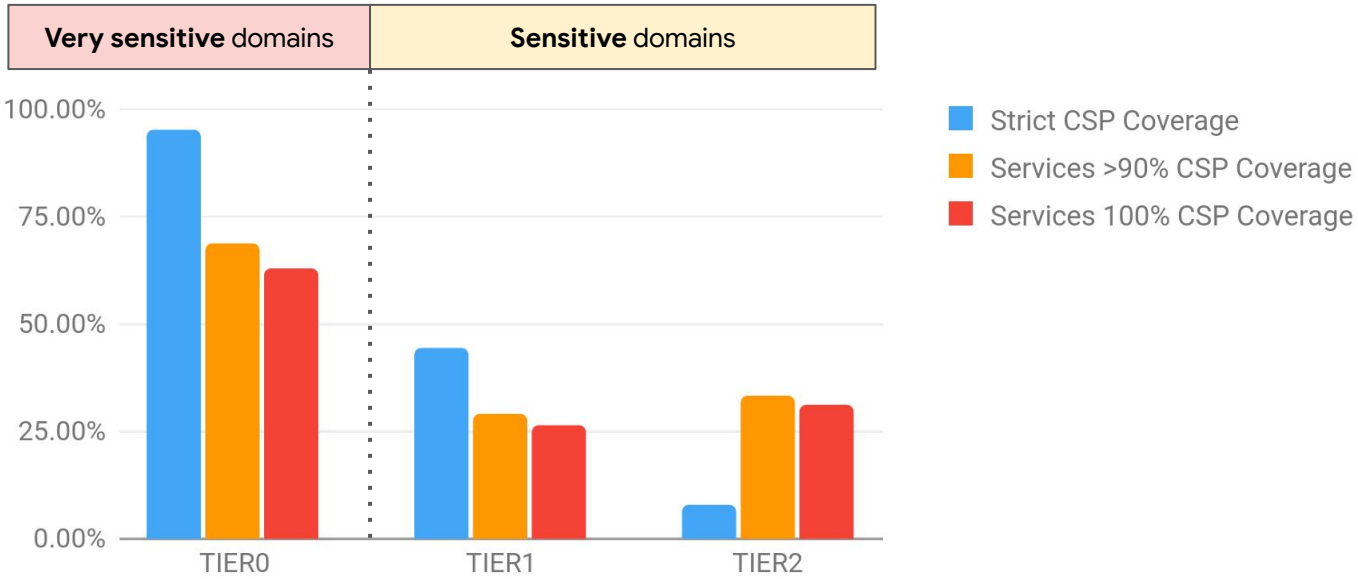
An analysis of externally reported XSS in 2018

# CSP Coverage at Google

**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



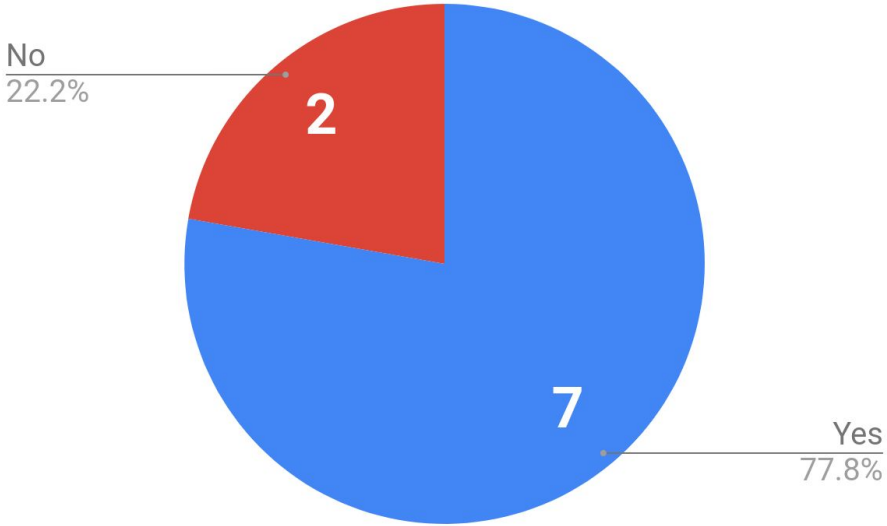
# Google Case Study: >60% of XSS Blocked by CSP

- Externally reported XSS in 2018
- Among 11 XSS vulnerabilities on very sensitive domains
  - 9 were on endpoints with strict CSP deployed, in 7 of which (**78%**) CSP successfully prevented exploitation

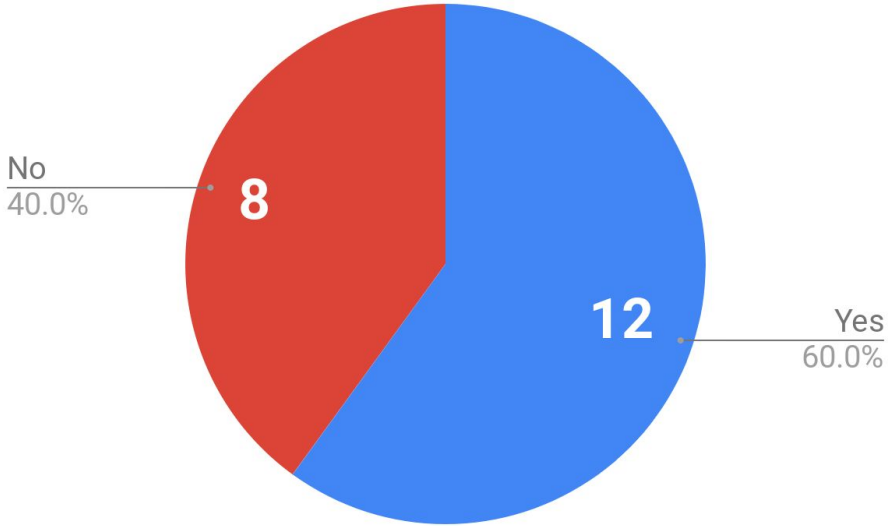
- Among all valid 69 XSS vulnerabilities on sensitive domains
  - 20 were on endpoints with strict CSP deployed
  - in 12 of which (**60%**) CSP successfully prevented exploitation

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

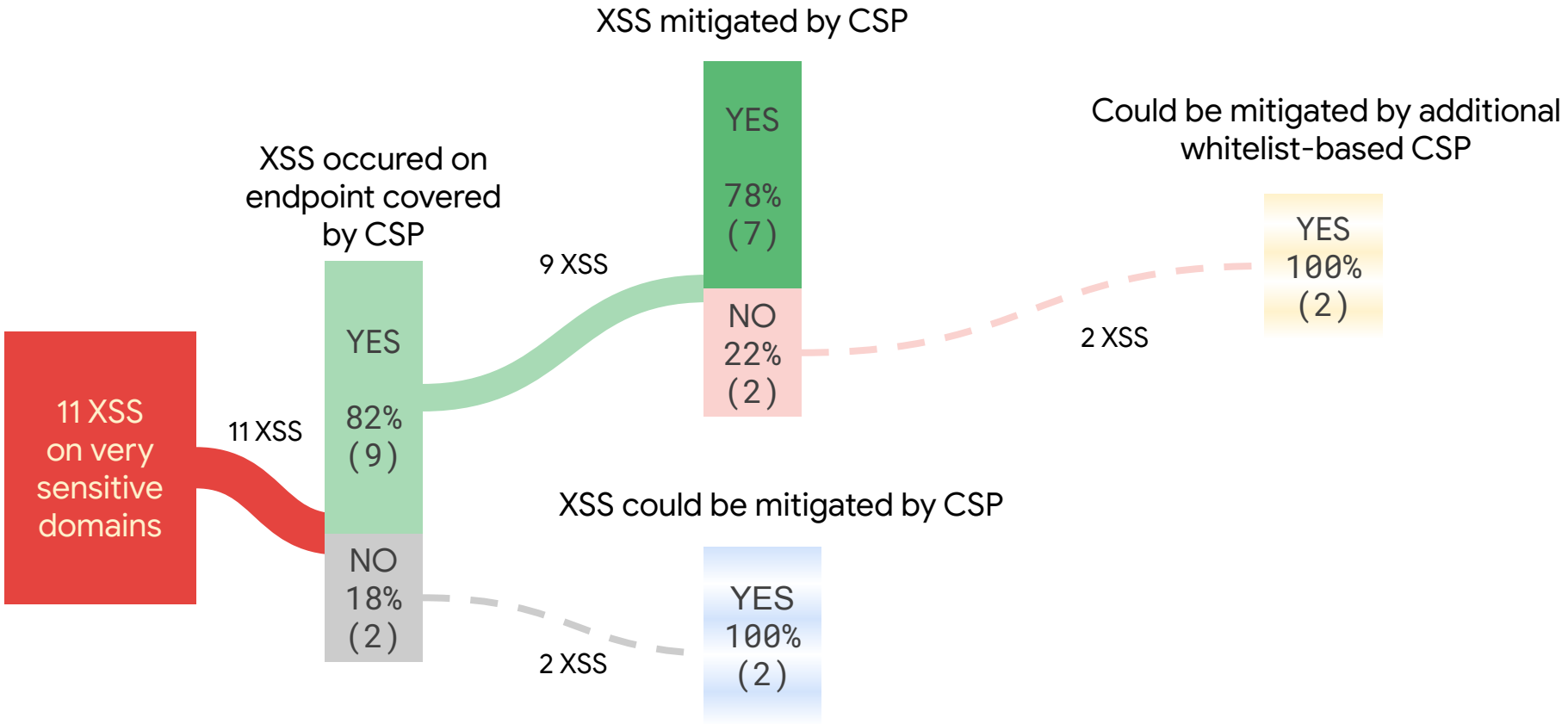
**Very sensitive** domains **with CSP**



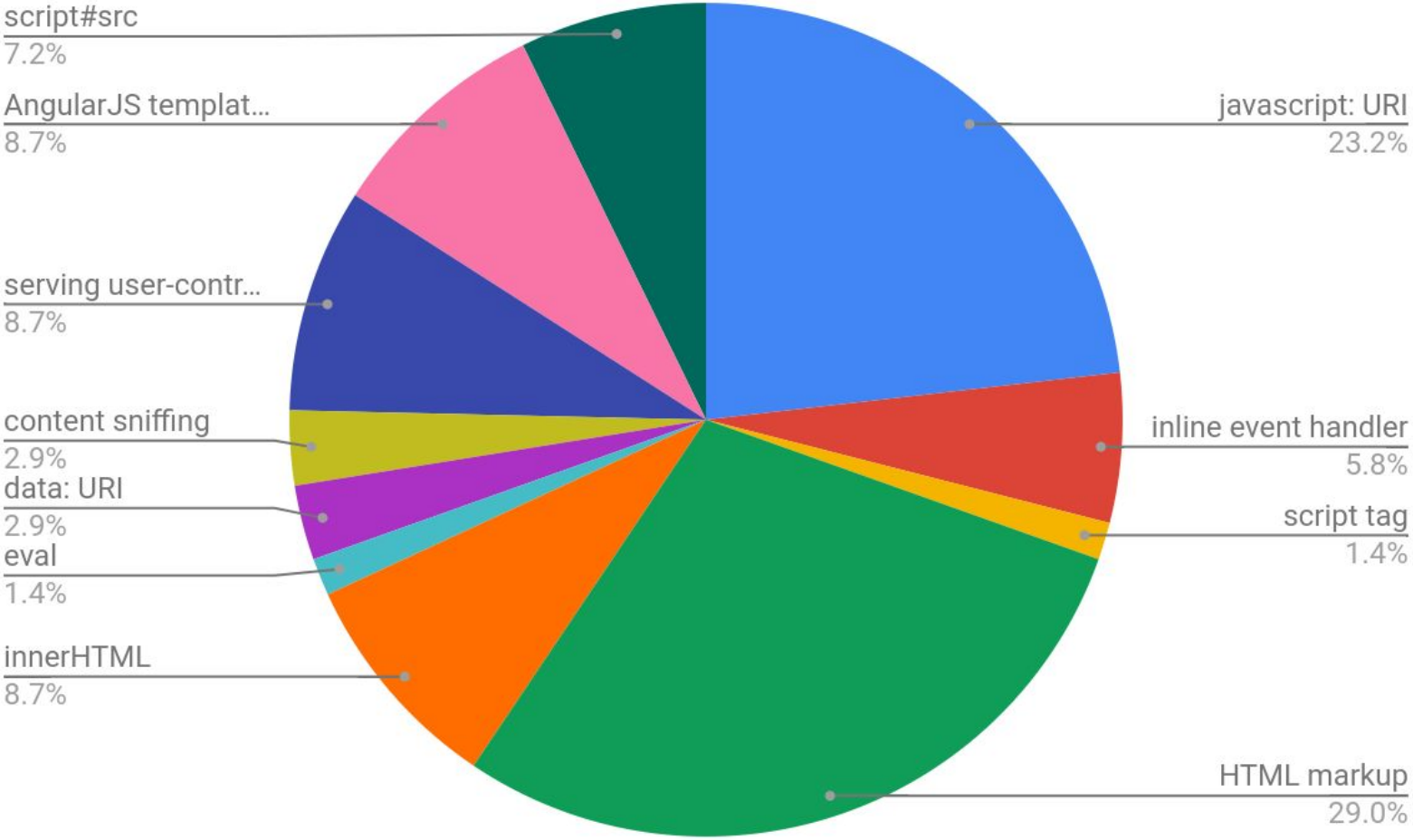

All sensitive domains **with CSP**



# On Very Sensitive Domains: ~80% of XSS Blocked by CSP



# Externally Reported XSS Exploited via



# Mapping Common XSS Sinks to CSP Features

| XSS sink (*injection into...*) | CSP blocks if... |
|---|---|
| javascript: URI<br>(i.e., javascript:**alert(1)**) | ~~'unsafe-inline'~~ |
| data: URI<br>(i.e., **data:text/html,<script>alert(1)</script>**) | ~~'unsafe-inline'~~ |
| (inner)HTML context<br>(i.e., **<script>alert(1)</script>** ) | ~~'unsafe-inline'~~ |
| inline event handler<br>(i.e., onerror=**alert(1)**) | ~~'unsafe-inline'~~ |
| eval()<br>(i.e., eval('**alert(1)**') | ~~'unsafe-eval'~~ |
| script#text<br>(i.e., var s = createElement('script'); s.innerText = '**alert(1)**';) | 'sha256-...'<br>'nonce-...' ~~'strict-dynamic'~~<br>(if scripts are not blindly nonced) |
| script#src<br>(i.e., var s = createElement('script'); s.src = '**attacker.js**';) | 'nonce-...' ~~'strict-dynamic'~~<br>(if scripts are not blindly nonced) |
| AngularJS-like template injection<br>(i.e., **{{constructor.constructor('alert(1)')()}}**) | Must be addressed in the framework.<br>e.g. upgrade AngularJS to Angular 2+ |

# Step-by-Step Towards a Stronger CSP

Incremental rollout of a nonce-based CSP

# Why NOT a whitelist-based CSP?

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


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

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

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

# Recap: What is a nonce-based CSP

**Content-Security-Policy:**

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

Execute only scripts with the correct *nonce* attribute

✔ `<script nonce="r4nd0m">kittens()</script>`

✘ `<script nonce="other-value">evil()</script>`

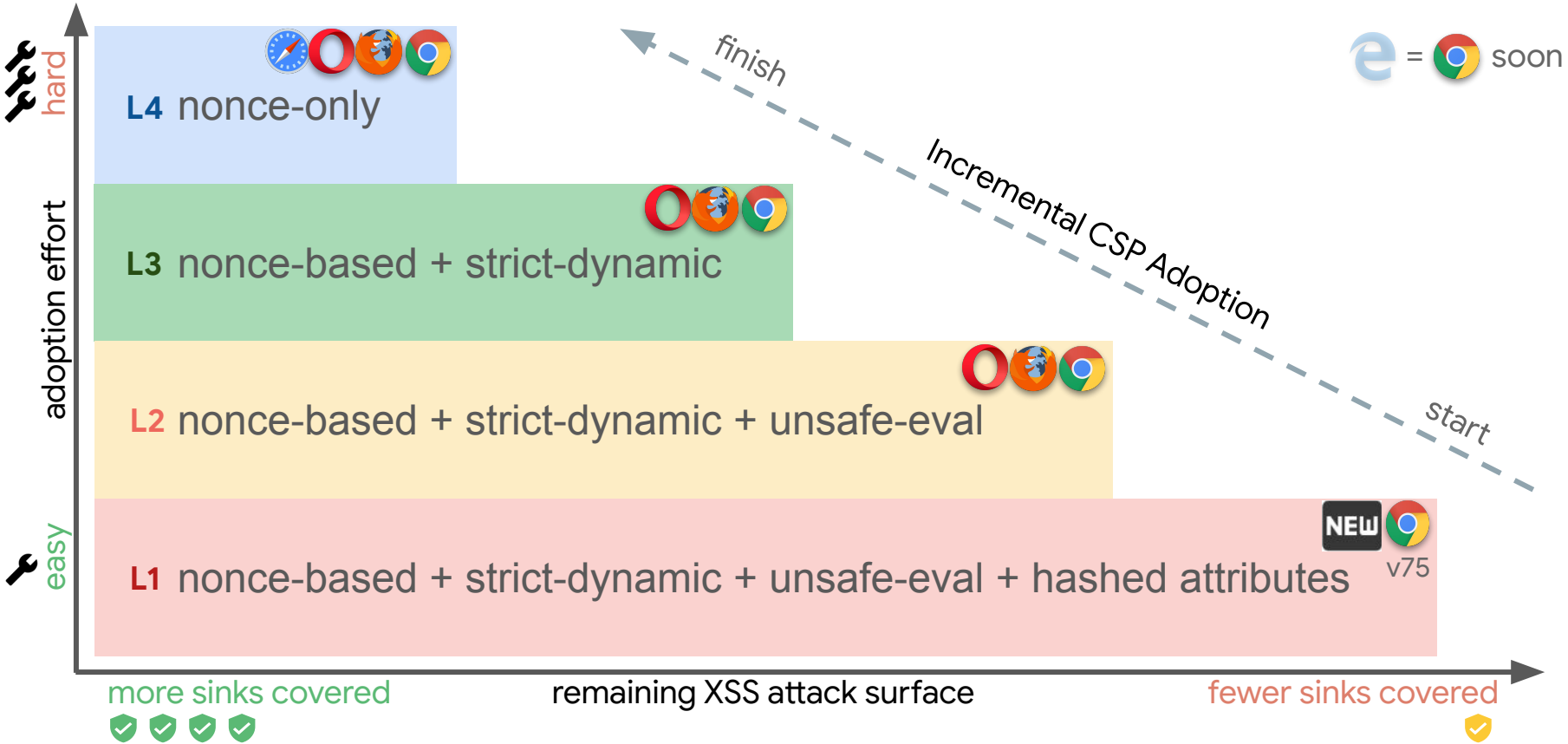
Trust scripts added by already trusted code

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

# Incremental Rollout of a nonce-based CSP

- Trade-off between **covered XSS sinks** vs. **ease of deployment**
- CSP security guarantees are not binary
  - Aim for actual reduction of attack surface instead of "raising the bar"
  - Trivial example: CSP w/o 'unsafe-eval' will block **all** eval-based XSS
- Refactoring work mostly varies based on
  - Type of CSP
  - Application (e.g. how many inline event handlers, use of eval(), size, etc.)

# Towards a Stronger nonce-based CSP (Level 1-4)



# L2: nonce-based + strict-dynamic + unsafe-eval

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

**TL;DR** Sweet spot! Good trade off between refactoring and covered sinks.

**PROs:**

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

**CONs:**

- eval() sink not covered
- DOM XSS partially covered
  - e.g. injection in dynamic script creation possible

# L2: nonce-based + strict-dynamic + unsafe-eval

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

**XSS Sinks Covered:**

| | |
|---|---|
| javascript: URI | ✓ |
| data: URI | ✓ |
| (inner)HTML context | ✓ |
| inline event handler | ✓ |
| eval | ✘ |
| script#text | ✘ |
| | ✓ if script is hashed |
| script#src | ✘ |
| AngularJS-like template injection | ✘ (✓ if upgraded to Angular 2+ or similar) |

# L2: nonce-based + strict-dynamic + unsafe-eval

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

Common Refactoring Steps:

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

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

# L3: nonce-based + strict-dynamic

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

**TL;DR** Sweet spot! Good trade off between refactoring and covered sinks.

**PROs:**

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

**CONs:**

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

# L3: nonce-based + strict-dynamic

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

**XSS Sinks Covered:**

| | |
|---|---|
| javascript: URI | ✓ |
| data: URI | ✓ |
| (inner)HTML context | ✓ |
| inline event handler | ✓ |
| eval | ✓ |
| script#text | ✘ |
| | ✓ if script is hashed |
| script#src | ✘ |
| AngularJS-like template injection | ✘ (✓ if upgraded to Angular 2+ or similar) |

# L3: nonce-based + strict-dynamic

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

Common Refactoring Steps:

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

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

# L3.5: **hash**-based + strict-dynamic

```
script-src 'sha256-avWk...' 'strict-dynamic';
object-src 'none'; base-uri 'none';
```

Refactoring steps for **static/single-page apps**:

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

```
<html>
 <a href="#">a</a>
 <a id="link">b</a>
 <script> // sha256-avWk...
  var urls = ['stuff.js',''dynLoadedStuff.js'];
  urls.map(url => {
    var s = document.createElement('script');
    s.src = url;
    document.body.appendChild(s); });
  document.getElementById('link')
    .addEventListener('click', alert('clicked'));
 </script>
</html>
```


# L4: nonce-only

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

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

**PROs:**

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

**CONs:**

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


# L4: nonce-only

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

## XSS Sinks Covered:

| | |
|---|---|
| javascript: URI | ✓ |
| data: URI | ✓ |
| (inner)HTML context | ✓ |
| inline event handler | ✓ |
| eval | ✓ |
| script#text | ✓ (✘ iff untrusted script explicitly marked as trusted) |
| | ✓ if script is hashed |
| script#src | ✓ (✘ iff untrusted URL explicitly marked as trusted) |
| AngularJS-like template injection | ✘ (✓ if upgraded to Angular 2+ or similar) |

# L4: nonce-only

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

Refactoring Steps:

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

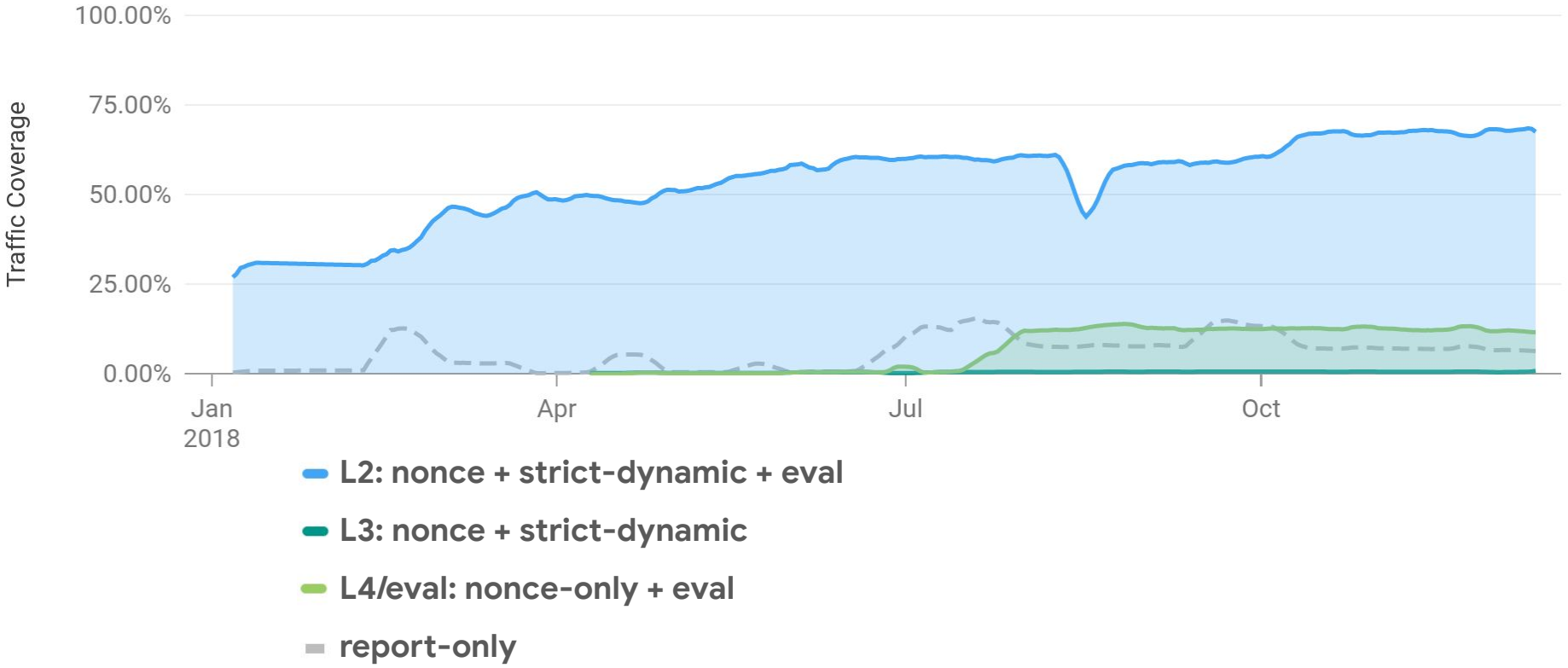
```
<html>
 <a href="#">a</a>
 <a id="link">b</a>
 <script nonce="r4nd0m" src="stuff.js"/>
 <script nonce="r4nd0m">
  var s = document.createElement('script');
  s.src = 'dynamicallyLoadedStuff.js'
  s.setAttribute('nonce', 'r4nd0m');
  document.body.appendChild(s);
  document.getElementById('link')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

# XSS Attack Surface by CSP Type

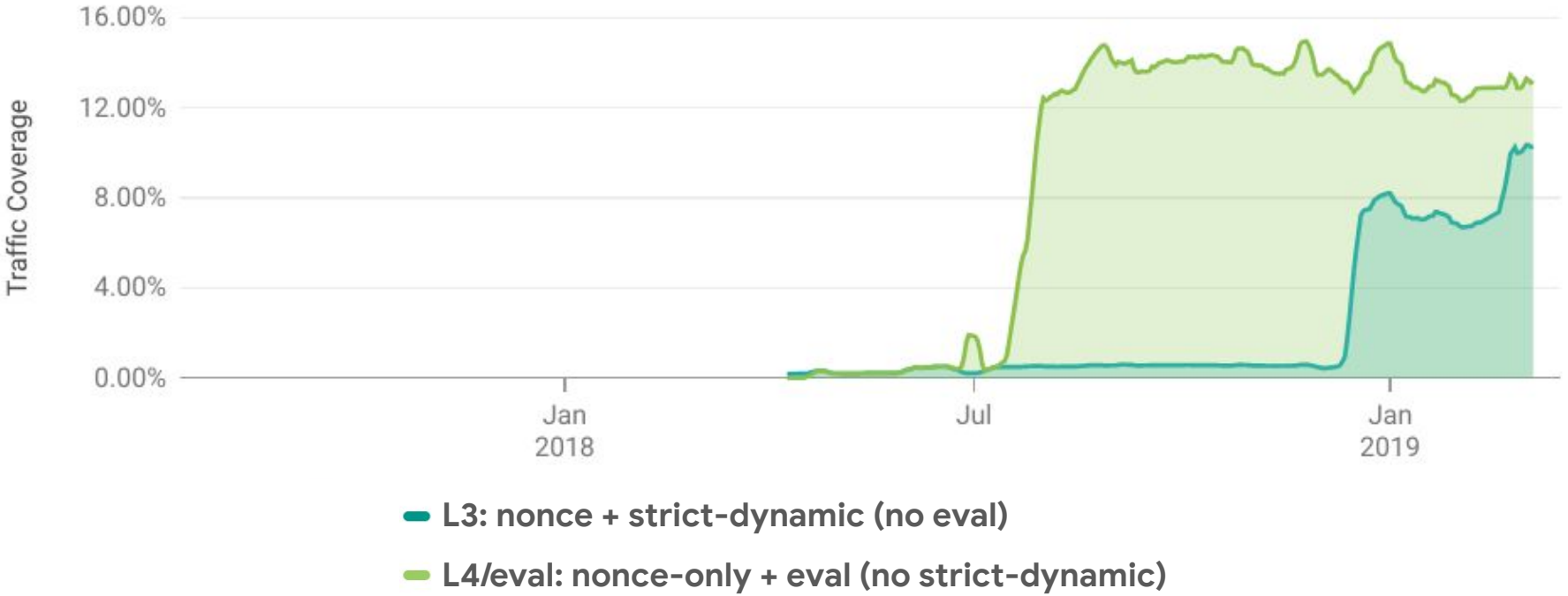
| | **L1** nonce-based, strict-dynamic, eval, hashed attributes | **L2** nonce-based, strict-dynamic, eval | **L3** nonce-based, strict-dynamic | **L4** nonce only | **L5** nonce only, whitelist | **Trusted Types** |
|---|---|---|---|---|---|---|
| javascript: URI | ✓ | ✓ | ✓ | ✓ | ✓ | **~(1)** |
| data: URI | ✓ | ✓ | ✓ | ✓ | ✓ | **~(1)** |
| (inner)HTML context | ✓ | ✓ | ✓ | ✓ | ✓ | **~(1)** |
| inline event handler | ~ | ✓ | ✓ | ✓ | ✓ | **~(1)** |
| eval | ✘ | ✘ | ✓ | ✓ | ✓ | ✓ |
| script#text | ✘ | ✘ | ✘ | ~ | ~ | ✓ |
| script#src | ✘ | ✘ | ✘ | ~ | ✓ | ✓ |
| AngularJS-like template injection | ✘ | ✘ | ✘ | ✘ | ✘ | ~ |

1) limited to DOM XSS

# CSP Coverage at Google by Type (2018)



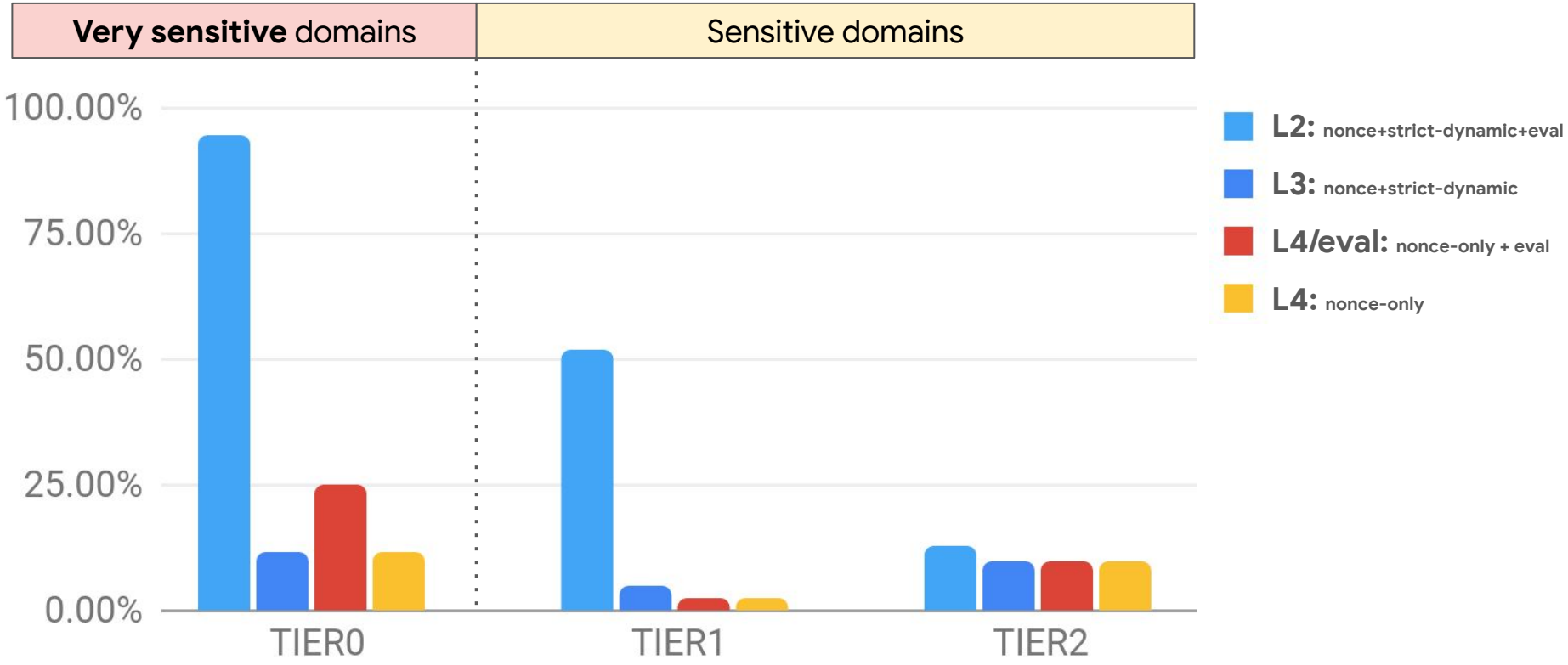
# CSP Coverage at Google by Type (excl. L2, 2019)



# CSP Types @Google (examples)

| | | L2: nonce+strict-dynamic+eval | L4/eval: nonce-only + eval | L3: nonce+strict-dynamic |
|---|---|---|---|---|
| | **Domain Tier** | **Strict CSP** ▼ | **Nonce-only** | **No Eval** |
| + accounts.google.com | TIER0 | 100.00% | 47.62% | 8.74% |
| + chrome.google.com | TIER0 | 100.00% | 99.97% | 0.45% |
| + myactivity.google.com | TIER1 | 100.00% | 13.90% | 13.90% |
| + meet.google.com | TIER1 | 100.00% | 21.10% | 0.03% |
| + passwords.google.com | TIER0 | 100.00% | 100.00% | 0.00% |
| + notifications.google.com | TIER1 | 100.00% | 100.00% | 100.00% |
| + takeout.google.com | TIER0 | 100.00% | 100.00% | 0.00% |
| + remotedesktop.google.com | TIER0 | 100.00% | 100.00% | 100.00% |
| + cloudsearch.google.com | TIER0 | 100.00% | 100.00% | 100.00% |
| + issuetracker.google.com | TIER0 | 100.00% | 100.00% | 0.00% |
| + script.google.com | TIER0 | 100.00% | 0.02% | 0.60% |
| + *.meet.google.com | TIER1 | 100.00% | 7.69% | 7.69% |
| + source.cloud.google.com | TIER1 | 100.00% | 0.00% | 100.00% |
| + dev.cloud.google.com | TIER1 | 100.00% | 0.00% | 100.00% |
| + lers.google.com | TIER1 | 100.00% | 0.00% | 100.00% |
| + console.actions.google.com | TIER0 | 100.00% | 0.00% | 0.00% |
| + console.cloud.google.com | TIER0 | 100.00% | 0.00% | 0.00% |

# CSP Types @Google by Domain Sensitivity (2019)



# Advanced CSP Techniques

Guru section ahead!

# New in CSP3 - script-src-elem and script-src-attr

**script-src-elem**

- applies to all script requests and inline script blocks.
- unlike script-src, this directive doesn't control attributes that execute scripts (inline event handlers)

**script-src-attr**

- controls attributes e.g. inline event handlers
- **'unsafe-hashes'** keyword allows the use of hashes for inline event handlers
- overrides the **script-src** directive for relevant checks.

(**style-src-elem** and **style-src-attr** are similar)

NEW v75

# L1: nonce-based + strict-dynamic + unsafe-eval + hashed attributes

```
script-src-attr 'unsafe-hashes' 'sha256-....';
script-src-elem 'nonce-r4nd0m' 'strict-dynamic' 'unsafe-eval';
object-src 'none'; base-uri 'none';
```

**TL;DR** Only use if you can't refactor inline event handlers / javascript: URIs

**PROs:**

+ Almost no refactoring required
  - `<script>` tags in initial response must have a valid nonce attribute
+ Strictly better than no CSP
  → Good starting point

**CONs:**

- Many sinks not covered (see next slide)
- Currently only supported in Chrome v75+
- In case of HTML injection
  → hashed event-handlers can be chained (ROP-like)

PoC: https://poc.webappsec.dev/csp/hashed_attr_csp.html

NEW v75

# L1: nonce-based + strict-dynamic + unsafe-eval + hashed attributes

```
script-src-attr 'unsafe-hashes' 'sha256-....';
script-src-elem 'nonce-r4nd0m' 'strict-dynamic' 'unsafe-eval';
object-src 'none'; base-uri 'none';
```

## XSS Sinks Covered:

| | |
|---|---|
| javascript: URI | ✓ |
| data: URI | ✓ |
| (inner)HTML context | ✓ |
| inline event handler | ~ (all hashed event handlers can be reused) |
| eval | ✘ (✓ if 'unsafe-eval' removed from CSP) |
| script#text | ✘ |
| | ✓ if script is hashed instead of nonced |
| script#src | ✘ |
| AngularJS-like template injection | ✘ (✓ if upgraded to Angular 2+ or similar) |

NEW v75

# L1: nonce-based + strict-dynamic + unsafe-eval + hashed attributes

```
script-src-attr 'unsafe-hashes' 'sha256-jE1Jw...' 'sha256-rRMdk...';
script-src-elem 'nonce-r4nd0m' 'strict-dynamic' 'unsafe-eval';
object-src 'none'; base-uri 'none';
```

Required Refactoring:

```
<html>
 <a href="javascript:void(0)">
 <a onclick="alert('clicked')">
 <script>alert('hi')</script>
 <script src="stuff.js"/>
</html>
```

→

```
<html>
 <a href="javascript:void(0)">  // sha256-rRMdk...
 <a onclick="alert('clicked')"> // sha256-jE1Jw...
 <script nonce="r4nd0m">alert('hi')</script>
 <script nonce="r4nd0m" src="stuff.js"/>
</html>
```

NEW v75

# L1.5: hash-based + strict-dynamic + hashed attributes

```
script-src-attr 'unsafe-hashes' 'sha256-jE1Jw...' 'sha256-rRMdk...';
script-src-elem 'sha256-CXAtY...' 'strict-dynamic';
object-src 'none'; base-uri 'none';
```

Refactoring steps for static/single-page apps:

```
<html>
 <a href="javascript:void(0)">
 <a onclick="alert('clicked')">
 <script src="stuff.js"/>
</html>
```

```
<html>
 <a href="javascript:void(0)">  // sha256-rRMdk...
 <a onclick="alert('clicked')"> // sha256-jE1Jw...
 <script>  // sha256-CXAtY...
  var s = document.createElement('script');
  s.src = 'stuff.js'
  document.body.appendChild(s); // allowed by strict-dynamic
 </script>
</html>
```

# Double Policies - The Best of Both Worlds

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

```
script-src 'self';
```

- More than one CSP header per response!
- Every CSP is enforced independently of each other by the browser
  - Adding additional CSPs can only add constraints
  - e.g. in order to run a script has to pass **every** CSP on the response!
- This allows very advanced setups
  - e.g. instead of allowing a script to load if it's whitelisted **OR** has a nonce (single CSP), it is possible to enforce that the script is from a trusted origin **AND** has a nonce
- Multiple CSPs can either be set via
  - multiple response headers
  - or in a single response header split via , (comma) - RFC 2616

# Double Policies - Example

CSP#1 | CSP#2

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

**Allowed** - ✓ CSP#1, ✓ CSP#2 - script has nonce and is hosted on same domain

```
<html>
 ✓ <script nonce="r4nd0m" src="foo.js"></script>
</html>
```

Blocked - ✓ CSP#1, ✘ CSP#2 - missing nonce attribute

```
<html>
 ✘ <script src="foo.js"></script>
</html>
```

Blocked - ✘ CSP#1, ✓ CSP#2 - domain not whitelisted

```
<html>
  ✘ <script nonce="r4nd0m" src="example.org/foo.js"></script>
</html>
```

# L5: Double Policy: separate whitelist + nonce-only

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

**TL;DR** Very hard to deploy (approach also makes sense for 'strict-dynamic' CSPs)

**PROs:**

+ Can block XSS where
  - nonced/trusted scripts get redirected
  - injection into script#src

**CONs:**

- Large refactoring required
- Additional burden of creating/maintaining whitelist
- Complex approach

# L5: Double Policy: separate whitelist + nonce-only

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

## XSS Sinks Covered:

| | |
|---|---|
| javascript: URI | ✓ |
| data: URI | ✓ |
| (inner)HTML context | ✓ |
| inline event handler | ✓ |
| eval | ✓ |
| script#text | ✓ (✘ iff untrusted script explicitly marked as trusted) |
| | ✓ if script is hashed |
| script#src | ✓ (only scripts from whitelisted domains, due to double policy usual whitelist bypasses don't apply!) |
| AngularJS-like template injection | ✘ (✓ if upgraded to Angular 2+ or similar) |

# CSP Beyond XSS - What About <style> Injections?

```
style-src-elem 'nonce-r4nd0m';
style-src-attr 'unsafe-inline';
```

- Aims to block CSS attacks by requiring CSP nonces for <style> tags:
  - CSS Keylogger - https://github.com/maxchehab/CSS-Keylogging
  - @import-based - https://medium.com/@d0nut/better-exfiltration-via-html-injection-31c72a2dae8b
- <style> tags are more powerful (CSS selectors!) than inline style attributes
- Reduces refactoring effort to noncing of <style> blocks
- `style-src 'nonce-r4nd0m'` would be better (stricter)
  - but much harder to deploy, because **all** inline styles would need to get refactored
- Can be combined with `script-src` CSP directives

# Productionizing CSP

## Better reporting and browser fallbacks

# Meaningful CSP Reports

```
script-src 'nonce-r4nd0m' 'strict-dynamic' 'report-sample'; report-uri /csp;
object-src 'none'; base-uri 'none';
```

- Add the **'report-sample'** keyword to the script-src directive
  ⟶ inline violations will contain a sample of the blocked expression
- Allows to differentiate between blocked inline scripts and inline event handlers
- Allows to identify which script was blocked
  ⟶ Possible to identify false positives (e.g. noise due to browser extensions)
- Example report:

```
csp-report:
    blocked-uri:"inline"
    document-uri:"https://f.bar/foo"
    effective-directive:"script-src"
    script-sample:"hello(1)"
```

# Overview of CSP Fallbacks

| ignored | in presence of | since version |
|---|---|---|
| `'unsafe-inline'` | `'nonce-...'` | CSP v2 |
| | `'sha256-...'` | CSP v2 |
| https:, http:, any.whitelist.com | `'strict-dynamic'` | CSP v3 |
| `script-src` (for elements) | `script-src-elem` | CSP v3 |
| `script-src` (for attributes) | `script-src-attr` | CSP v3 |
| `style-src` (for elements) | `style-src-elem` | CSP v3 |
| `style-src` (for attributes) | `style-src-attr` | CSP v3 |

# Fallbacks for Old Browsers

```
script-src 'nonce-r4nd0m' 'strict-dynamic' https: 'unsafe-inline';
object-src 'none'; base-uri 'none';
```

ignored
not supported

CSP as seen by CSP3 Browser

```
script-src 'nonce-r4nd0m' 'strict-dynamic' https: 'unsafe-inline';
object-src 'none'; base-uri 'none';
```

CSP as seen by CSP2 Browser

```
script-src 'nonce-r4nd0m' 'strict-dynamic' https: 'unsafe-inline';
object-src 'none'; base-uri 'none';
```

CSP as seen by CSP1 Browser

```
script-src 'nonce-r4nd0m' 'strict-dynamic' https: 'unsafe-inline';
object-src 'none'; base-uri 'none';
```

# Conclusions

Enough! What should I remember of this talk?

# Wrapping up

- Nonce-based CSPs cover the classical reflected/stored XSS very well
- A nonce-based CSP with 'strict-dynamic'
  - is a good trade-off between security and adoption effort
  - covers classical reflected/stored XSS very well
  - has limitations when it comes to DOM XSS
  - was able to block 60%-80% of externally reported XSS at Google
- If possible upgrade to nonce-only
- CSP is a defense-in-depth mechanism
  - it's meant to protect the user when primary security mechanisms (e.g. escaping) fail
  - it's not an excuse to not fix underlying bugs
- Always double check your CSP with the CSP Evaluator: csp-evaluator.withgoogle.com

# In Brief

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

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

L3

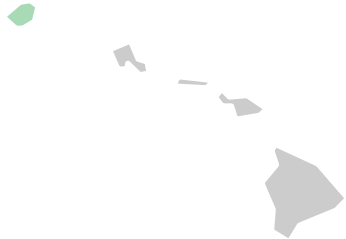
If possible, upgrade to a **nonce-only CSP**:

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

L4

Recommended reading: csp.withgoogle.com

#LocoMocoSec Kauai, Hawaii 2019



# Mahalo! 🤙

## Questions?

You can find us at:


{lwe,mikispag}@google.com

**@we1x**, **@mikispag**