Google | Privacy, Safety & Security

#LocoMocoSec 2024

A Recipe for Scaling (Web) Security

Lessons from Google's Frontlines

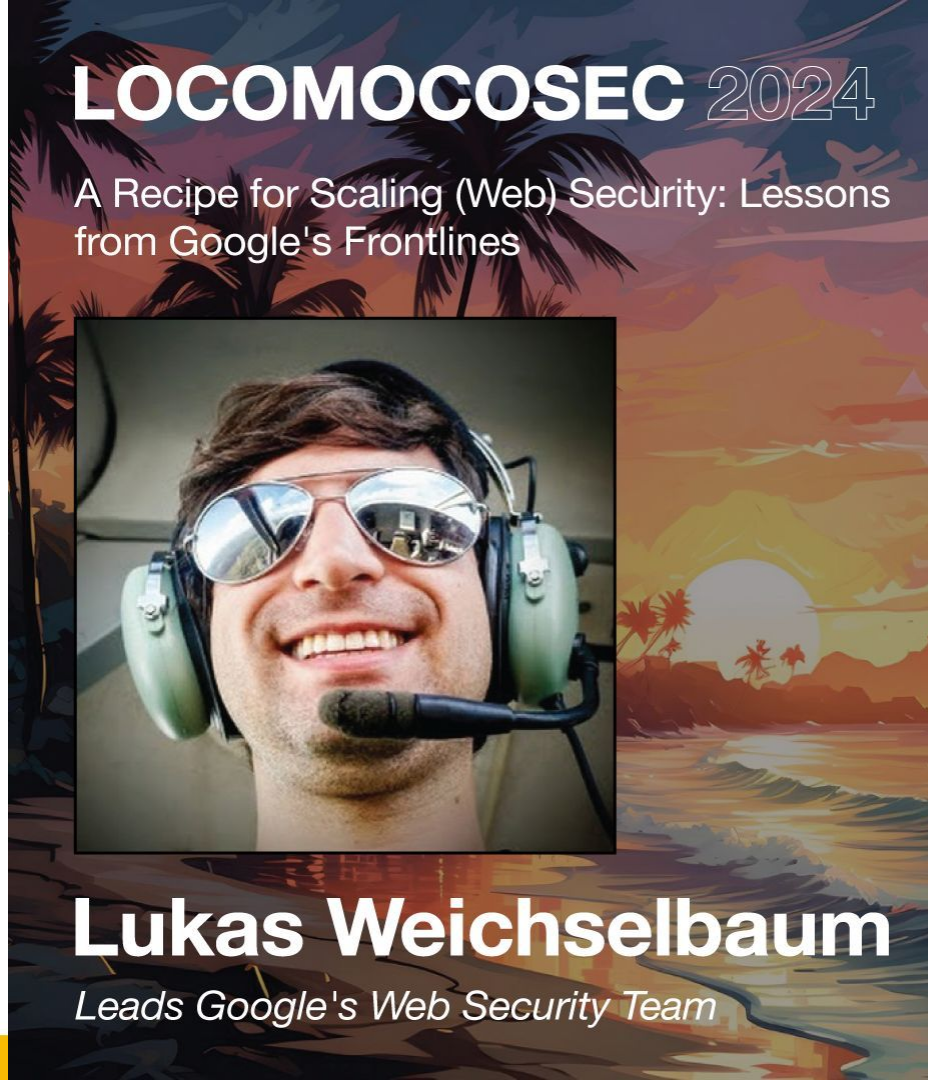
Lukas Weichselbaum

Meet your Chef 🧑‍🍳

Lukas Weichselbaum

Senior Staff Web Security Enthusiast

- Leads Google's Web Security Team (ISE Web)
- Deployment of web security features and safe defaults across Google's web application portfolio
- Passionate about addressing and mitigating entire classes of web vulnerabilities in the web platform with CSP, Trusted Types, Fetch Metadata, etc.



Menu

Appetizer

The (Google) Web Security Challenge

Main

Addressing Root Causes

Safe Coding

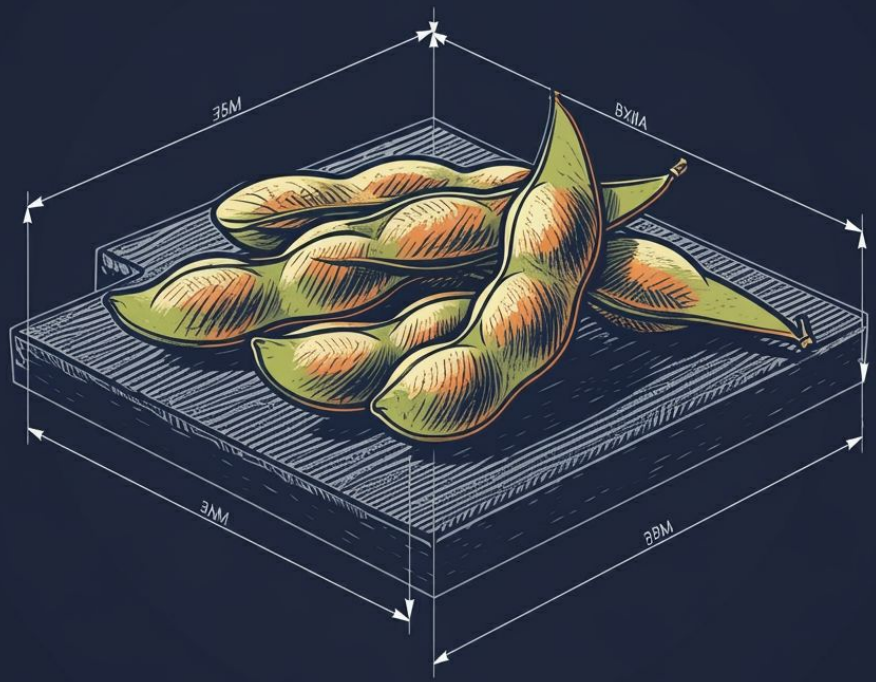
Fixing Legacy Applications

Dessert

Key Takeaways

Appetizer

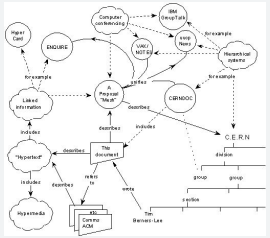
The (Google) Web Security Challenge



Web Originally vs Today

The Web Originally

"We should work toward a universal linked information system, in which generality and portability are more important than fancy graphics techniques and complex extra facilities. (...) The aim would be to allow a place to be found for any information or reference which one felt was important, and a way of finding it afterwards. The result should be sufficiently attractive to use that it the information contained would grow past a critical threshold, so that the usefulness the scheme would in turn encourage its increased use."



Tim Berners-Lee, [March 1989]

The Web Today

- Powers millions of applications, billions of users, trillions in market cap, zettabytes of data
- Data processed & stored is often highly sensitive
- Web applications can trigger highly sensitive actions
  - Payments, install apps on devices, control infrastructure/cloud/machines, ...

Google makes heavy use of the web platform

Google has one of the largest web ecosystems

>600

*.google.com Subdomains

>900

Sensitive User-Facing Web Applications

>1 Trillion

Requests

Many highly sensitive web applications

Google makes heavy use of the web platform

Google has one of the largest web ecosystems

What makes a web application "sensitive"?

>600

*.google.com Subdomains

>900

Sensitive User-Facing Web Applications

>1 Trillion

Requests

Many highly sensitive web applications

A Framework for Classifying Domain Sensitivity

Google classifies "sensitivity" of web applications with a concept called **Domain Tiers***

Tier0 Domains where a critical vulnerability (e.g. XSS or authorization bypass) could lead to a compromise of a user's account or execution of code on their system.

Tier1 Domains where a vulnerability could disclose particularly sensitive user data.

Tier2 Domains where the impact of a vulnerability may be damaging, but is less likely to lead to the disclosure of highly sensitive data.

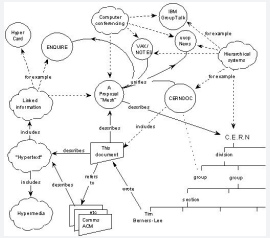
Tier3+ Domains where the impact of a vulnerability is low.

* goo.gle/domain-tiers

Web Originally vs Today

The Web Originally

"We should work toward a universal linked information system, in which generality and portability are more important than fancy graphics techniques and complex extra facilities. (...) The aim would be to allow a place to be found for any information or reference which one felt was important, and a way of finding it afterwards. The result should be sufficiently attractive to use that it the information contained would grow past a critical threshold, so that the usefulness the scheme would in turn encourage its increased use."



Tim Berners-Lee, March 1989

The Web Today

The Web wasn't built for this

- Powers millions of applications, billions of users, trillions in market cap, zettabytes of data
- Data processed & stored is often highly sensitive
- Web applications can trigger highly sensitive actions
  - Payments, install apps on devices, control infrastructure/cloud/machines, ...

The web platform isn't safe by default

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

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

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.

The web platform isn't safe by default

Historically, there were three *origin* [illegible] an application platform:

Mostly solved by the web platform

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

Difficult to fully address in the web platform. Powerful opt-ins available!

2. **Injections**: C[illegible] 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 b[illegible]

Join tomorrow's 3PCD talk to hear more :)

One of the Original Web Platform Sins: Injections

Everything is code

- Any text within an HTML page can turn into code (`<script>...</script>`)
- Any URL can be executable (`javascript:...`)

Injected code executes with privileges of the application and receives unrestricted access to all data & functionality in the application's origin.

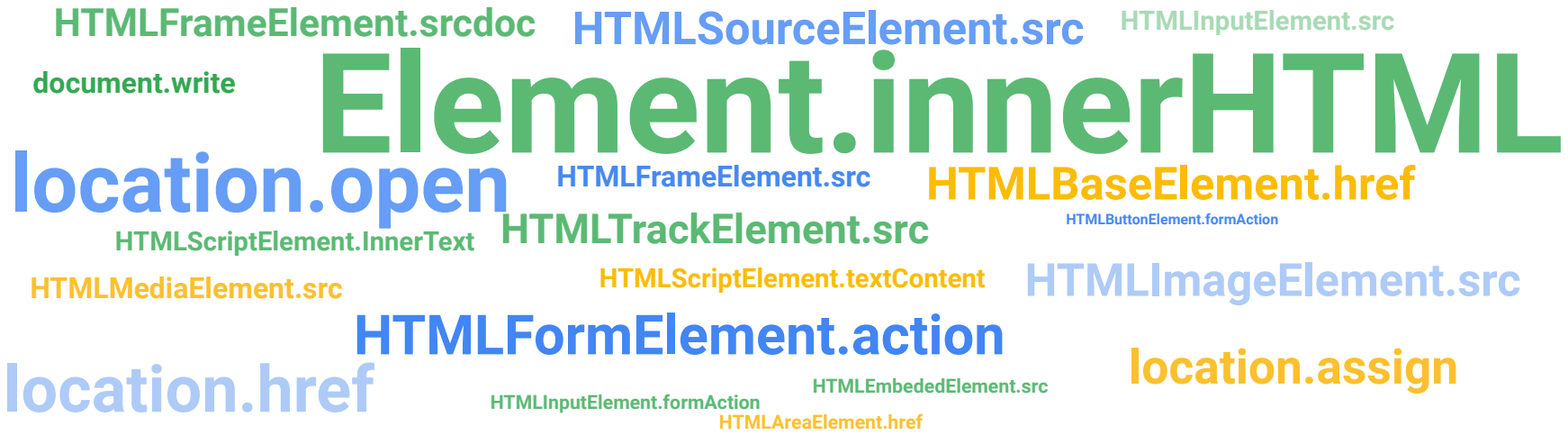
Vulnerabilities: Cross-site scripting (XSS)

Tackling XSS will be our canonical example for today's scaling recipes

The web platform isn't safe by default

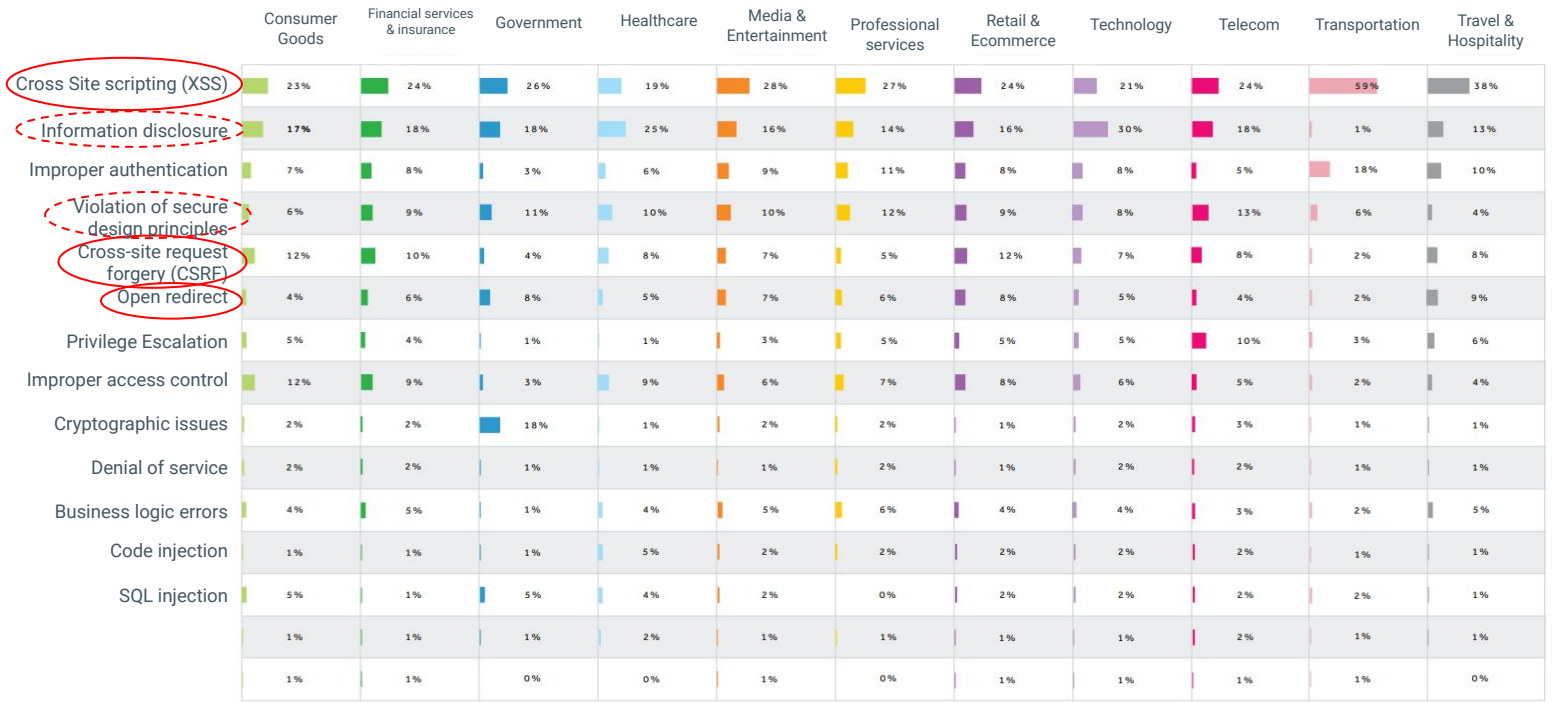
Example: DOM XSS

- User controlled strings get converted into code via dangerous DOM APIs
- Caused by **60+ DOM APIs that are not safe by default**



Web vulnerabilities are prevalent across the industry

HackerOne: Report 2018



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

Web vulnerabilities are prevalent across the industry

HackerOne

Global vulnerability trends 2020

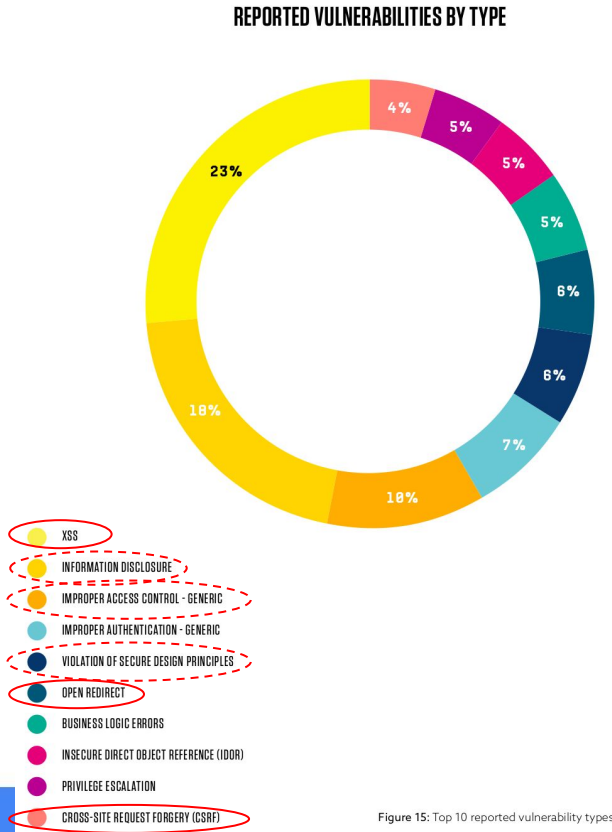


Figure 15: Top 10 reported vulnerability types.

Global vulnerability trends 2021

TOP 10 VULNERABILITIES

How has the Top 10 changed in the past 12 months?

2020		2021		YoY % increase in valid reports
1	Cross-site Scripting (XSS)	1	Cross-site Scripting (XSS)	7%
2	Improper Access Control	2	Information Disclosure	58%
3	Information Disclosure	3	Improper Access Control	26%
4	Server-Side Request Forgery (SSRF)	4	Insecure Direct Object Reference (IDOR)	9%
5	Insecure Direct Object Reference (IDOR)	5	Privilege Escalation	55%
6	Privilege Escalation	6	Improper Authentication	18%
7	SQL Injection	7	Code Injection	12%
8	Improper Authentication	8	SQL Injection	-7%
9	Code Injection	9	Server-Side Request Forgery (SSRF)	-17%
10	Cross-Site Request Forgery (CSRF)	10	Business Logic Errors	67%

hackerone

Remember LocoMocoSec 2019?

Let's revisit the web security landscape at Google back then.

Web vulnerabilities at Google ~5 years ago

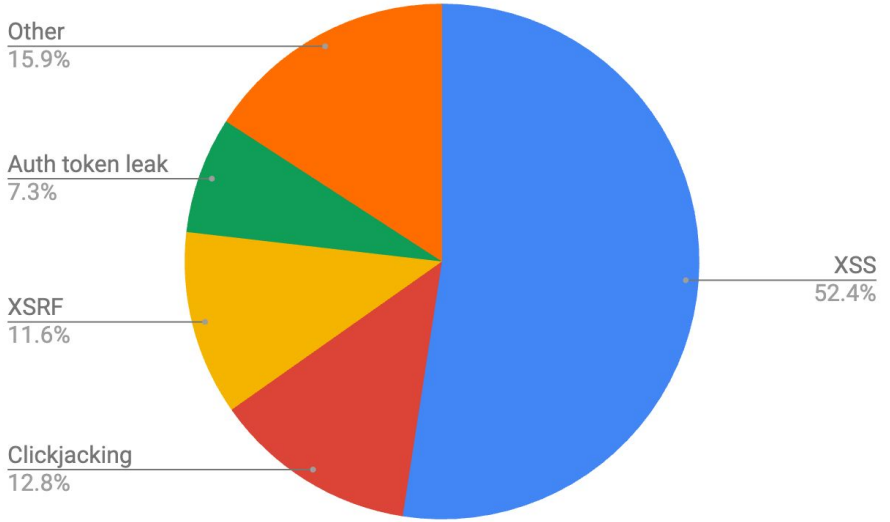
Addressing common and new web vulnerabilities (in particular XSS) was a top priority!

Common vulnerabilities in Google Web applications*

- Most common: **XSS** (100+ every year)
- Common: XSRF, Clickjacking, etc.

New vulnerability classes on the Web

- Microarchitectural issues like Spectre
- Advanced web APIs used by attackers
- Improved exploitation techniques



* Google Bug Hunters data from 2020

Fast forward

Where are we today?

Fast Forward

Where are we today after applying our web security scaling recipe?

Almost no XSS

XSS isn't our top concern anymore.
We had **almost no exploitable XSS** across hundreds of web applications that are **built on our recommended and hardened web frameworks** and have strongly reduced the number of XSS in legacy frameworks.

Safe by Default Launches

Newly launching web applications like Gemini are safe by default and don't require launch reviews. Important security features like CSP and Trusted Types are built into the framework and can't regress.

Security is Transparent for Developers

Web security is baked in and guaranteed by frameworks and the ecosystem allowing developers to fully focus on developing great features instead of having to stay on top of web security. Important security features are backported at scale by the security team.

Fast Forward

Where are we today after applying our web security scaling recipe?

Almost no XSS

XSS isn't our top co
We had **almost no**
hundreds of web ap
on our recommen
frameworks and h
number of XSS in le

rent for Developers

in and guaranteed by
osystem allowing
us on developing great
ving to stay on top of web
urity features are
the security team.

Main Dish

Loco Moco

Today's Main Dish

Scaling Web Security Recipe

The three ingredients for obliterating entire (web) vulnerability classes

01

Address Root Causes

02

Safe Coding Approach

03

Fix Legacy Code

The three ingredients for obliterating entire (web) vulnerability classes

01

Address Root Causes

02

Safe Coding Approach

03

Fix Legacy Code

Addressing Root Causes

Addressing vulnerability classes closer to their root cause exponentially increases scalability of the fix

Scalability	*Patching a vulnerability in the…*	*Difficulty*
	Application code	easy

Example: Patching a DOM XSS vulnerability directly in the application

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

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

Addressing Root Causes

Addressing vulnerability classes closer to their root cause exponentially increases scalability of the fix

Scalability	*Patching a vulnerability in the…*	*Difficulty*
	Application code	easy
	Server- or client-side framework	hard

Example:

- Web framework enforces Trusted Types and a strict CSP by default.
  ⟶ New applications are written in a way that avoid dangerous APIs like innerHTML
  ⟶ Many common XSS sinks disabled in the browser (javascript: URIs, innerHTML, etc.)

Addressing Root Causes

Addressing vulnerability classes closer to their root cause exponentially increases scalability of the fix

Scalability	*Patching a vulnerability in the...*	*Difficulty*
	Application code	easy
	Server- or client-side framework	hard
	Web platform (browser)	very hard

The web platform isn't safe by default

But today we have powerful opt-in web platform security features and many sharp edges have been fixed in the recent years.

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

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

Addressing Root Causes

Addressing vulnerability classes closer to their root cause exponentially increases scalability of the fix

Scalability	*Patching a vulnerability in the…*	*Difficulty*
	Application code	easy
	Server- or client-side framework	hard
	Web platform (browser)	very hard

Examples:

- Make `document.domain` immutable **in the web platform**
- Default to `https` **in the web platform**
- Web platform security features like CSP, Trusted Types, etc. (opt-in)

But how to fix the web platform without breaking the Internet?

The Web is designed to be backward compatible

How to overcome backward compatibility issues?

Ideal case: New default / Fixed for everyone!

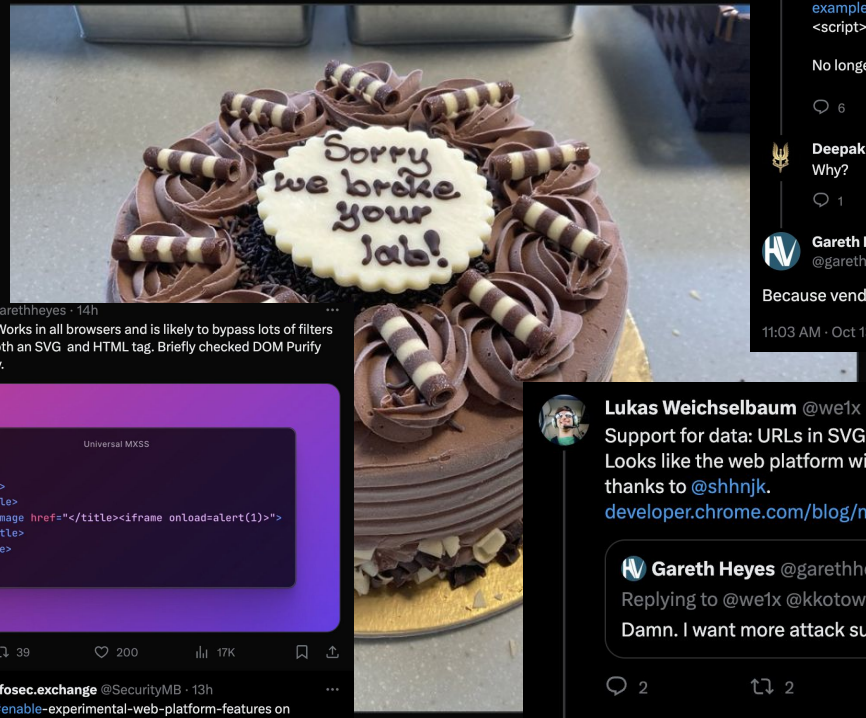
- Immediately removes a vulnerability class for existing/future applications
- Involves measuring, outreach and support for affected sites to **break as little as possible**

Example: Make `document.domain` immutable

The Web is

Gareth Heyes @garethheyes · Nov 9

Google has apologised in style for breaking our dangling markup lab with new mitigations in Chrome. Thanks from @WebSecAcademy and @PortSwiggerRes!



Lukas Weichselbaum @we1x · May 30

'document.domain' will be immutable in Chrome 115.
Thank you and congratulations to @mikewest and his amazing team for making the web safer by deprecating legacy badness. It's great to see Chrome leading by example here. Hopefully others will follow.

5 25 87

Gareth Heyes @garethheyes

Damn. I want more attack surface it's more fun.

Gareth Heyes @garethheyes · 14h

Universal MXSS. Works in all browsers and is likely to bypass lots of filters because title is both an SVG and HTML tag. Briefly checked DOM Purify and it looked okay.

6 39 200 17K

@securitymb@infosec.exchange @SecurityMB · 13h

Won't work with #enable-experimental-web-platform-features on Chrome 😉

1 3 5 1K

Gareth Heyes @garethheyes

The future of XSS is going to be so boring. Thankfully we'll have AI to keep us occupied.

Gareth Heyes @garethheyes · Oct 13

My current fav XSS has gone 😲

example.com/?%0aalert%281%...
<script>location.protocol='javascript'</script>

No longer works 👎

6 9 69 28K

Deepak bug_vs_me @bug_vs_me · Oct 13

Why?

1 1 1.5K

Gareth Heyes @garethheyes

Because vendors read my tweets

11:03 AM · Oct 13, 2023 · **1,833** Views

Lukas Weichselbaum @we1x · Jun 2

Support for data: URLs in SVG <use> element is going away too.
Looks like the web platform will be even less fun soon for @garethheyes thanks to @shhnjk.
developer.chrome.com/blog/migrate-w...

> **Gareth Heyes** @garethheyes · May 31
> Replying to @we1x @kkotowicz and @mikewest
> Damn. I want more attack surface it's more fun.

2 2 12 3.1K

Gareth Heyes @garethheyes

You are kidding me. That was my fav XSS vector!!!!!

The Web is designed to be backward compatible

How to overcome backward compatibility issues?

Plan B: Opt-in

- An **opt-in** mechanism to not affect existing applications
- **Requires additional work**:
  - **Enable feature by default** in serving frameworks for new launches [**Green field**]
  - **Backport** to existing applications [**Brown field**]

Example: Content Security Policy

The web platform is malleable!

Many important web platform features were added in the recent years that help to fix or turn off dangerous default behaviour. A few examples:

Content Security Policy Level 3 (strict-dynamic, script-src-attr/elem, etc.)	52	79	52	15.4
Trusted Types	83	83	x	x
Fetch Metadata	76	79	90	16.4
Cross Origin Opener Policy	83	83	79	15.2

Update standards position on Trusted Types - fixes mozilla#20

Browse files

new-tt-position (mozilla/standards-positions#936)

Frederik Braun committed on Dec 11, 2023 — 1 parent 742589f commit 784f70d

Showing 1 changed file with 2 additions and 2 deletions. Whitespace | Ignore whitespace | Split | Unified

4 activities.json

```
@@ -1573,8 +1573,8 @@
    "description": "An API that allows applications to lock down powerful APIs to only accept non-spoofable, typed values in place of strings to prevent vulnerabilities caused by using these APIs with attacker-controlled inputs.",
    "id": "trusted-types",
    "mozBugUrl": null,
-   "mozPosition": "neutral",
-   "mozPositionDetail": "The API could be used to harden sites against certain cross-site scripting issues, but it is sufficiently complex that we are concerned that it will not be suitable for many sites.",
+   "mozPosition": "positive",
+   "mozPositionDetail": "Mozilla believes that preventing DOM-based XSS is an important security goal. The track record of preventing DOM-based XSS is convincing. That being said, the Trusted Types specification is also providing API methods with little or unknown value and uptake, like getPropertyType, getAttributeType. Additionally, there are features in the Chrome implementations that are not yet standardized, like the beforepolicycreation event. These should ideally be properly standardized (based on a proven need) or deprecated and removed. Dealing with inscrutable third-party dependencies or external JavaScript has been a major concern of security and enforcing reasonable boundaries is a promising approach. We believe that runtime checking of untrusted HTML might pair well with a Sanitizer API that should be complementary.",
    "mozPositionIssue": 20,
```

…at help to fix or turn off

Content Security
(strict-dynamic, scr

15.4

Trusted Types

X

…ta

16.4

…p

15.2

Open **Trusted Types** #186

bartoszniemczura opened this issue on May 15, 2023 · 16 comments

annevk commented 2 weeks ago — Contributor

Colleagues and I discussed this once more and our suggestion is to mark this as 'position: support' one week from now with the ongoing caveat that standardization needs to be completed before we can feel fully confident about the implementation in WebKit.

(The CSP integration issue discussed above has now moved to #355 (comment).)

👍 2

Addressing Root Causes

Addressing vulnerability classes closer to their root cause exponentially increases scalability of mitigations

Scalability	*Patching a vulnerability in the...*	*Difficulty*
	Application code	easy
	Server- or client-side framework	hard
	Web platform (browser)	very hard

Key takeaways:

- The web platform is malleable!
- **Small businesses can contribute too!** The W3C and browser vendors take community feedback very seriously. It can make the difference between shipping or not shipping a new feature.

And sometimes it takes a tweet from Jim (and some help from Igalia)

Jim Manico from Manicode Security @manicode · Aug 25, 2021
I believe that Safari's lack of CSP3 support and similar W3C standards is a reason to boycott and stop using it.

I now see Safari as a browser that primarily impedes the secure web.

Edge, Chrome and FireFox all support CSP3 well.

Why not Safari? I'm open to counter opinions.

8 2 25

Mark Goodwin @mr_goodwin · Aug 25, 2021
I expect @johnwilander will know the answer to that?

FWIW, I share your frustration that Safari lags on key standards. It's still a fine browser and, while I prefer Firefox, I'd still sooner use it than Chrome.

1 1

Jim Manico from Manicode Security @manicode · Aug 26, 2021
I am trying not to harass John he is good people and I believe this is above his pay-grade.

1 5

John Wilander @johnwilander · Aug 26, 2021
Don't worry. I want to see CSP3 in WebKit asap and always forward good faith feedback to the team. If you want to talk directly to the person in charge, he's @othermaciej.

2 10

Lukas Weichselbaum
@we1x
Would you accept pull requests? 🙂

9:42 AM · Aug 26, 2021

1 year later

John Wilander
@johnwilander

"WebKit in Safari 15.4 improves support for Content Security Policy Level 3 (...) New support for 'strict-dynamic', 'unsafe-hashes', and 'report-sample' source expressions give developers more flexibility." webkit.org/blog/12445/new... Happy to deliver you this message, my friend!

The Web is designed to be backward compatible

How to overcome backward compatibility issues?

Plan B: Opt-in

- An **opt-in** mechanism to not affect existing applications
- **Requires additional work**:
  - **Enable feature by default** in serving frameworks for new launches [**Green field**]
  - **Backport** to existing applications [**Brown field**]

Example: Content Security Policy

The three ingredients for obliterating entire (web) vulnerability classes

01

Address Root Causes

02

Safe Coding Approach

03

Fix Legacy Code

Securing Newly Written Code with Safe Coding

Shifting **responsibility** for web security **away from developers** to security engineers and ultimately **to frameworks** enables to eliminate entire vuln classes in new code

Scalability	*Securing Newly Written Code (Green Field)*	*Who's Responsible*
	No process / developers are on their own	Developers

Securing Newly Written Code with Safe Coding

Shifting **responsibility** for web security **away from developers** to security engineers and ultimately **to frameworks** enables to eliminate entire vuln classes in new code

Scalability	*Securing Newly Written Code (Green Field)*	*Who's Responsible*
	No process / developers are on their own	Developers
	Security reviews, documentation, dev training	Security Engineers

Securing Newly Written Code with Safe Coding

Shifting **responsibility** for web security **away from developers** to security engineers and ultimately **to frameworks** enables to eliminate entire vuln classes in new code

Scalability	*Securing Newly Written Code (Green Field)*	*Who's Responsible*
	No process / developers are on their own	Developers
	Security reviews, documentation, dev training	Security Engineers
	Safe-by-default APIs	Security Engineers

Safe Coding - Developing software that is secure by design

The two main ingredients of **Safe Coding** are:

Safe-by-default APIs

These APIs are inherently safe (they're a way of expressing a security invariant)

- ➜ Code using these APIs can under no (reasonable) circumstances suffer from the vulnerability in question
- ➜ These APIs **replace** unsafe/dangerous APIs (e.g. setting script.src, innerHTML, SQL string concat)

Enforcement of security invariants like safe-by-default APIs, languages, frameworks, etc.

Ensures that inherently safe APIs are used correctly and comprehensively throughout a code base

- ➜ Enforcement needs to be built into the development ecosystem
- ➜ A "mature ecosystem" enforces security invariants at multiple levels
  - compiler, pre-submit checks, web platform, etc.

Safe Coding Benefits

Scalable

Eliminates entire classes of security vulnerabilities, such as cross-site scripting (XSS)

Proactive

Prevents vulnerabilities from ever being introduced during development (compiler & presubmit checks)

Confidence

That a particular class of vulnerabilities does not exist in an application and will not be introduced as the code base evolves.

Low Friction

Reduces friction for developers by providing feedback as early as possible (compile-time vs. runtime in prod). Also doesn't require developers to become security experts.

Securing Newly Written Code with Safe Coding

Shifting **responsibility** for web security **away from developers** to security engineers and ultimately **to frameworks** enables to eliminate entire vuln classes in new code

Scalability	*Securing Newly Written Code (Green Field)*	*Who's Responsible*
	No process / developers are on their own	Developers
	Security reviews, documentation, c	ng
	Safe-by-default APIs	ng

Google enforces **safevalues** at:
- runtime via Trusted Types (TT)
- compile time (tsec)
- presubmit check

Fully compatible with strict CSP & TT via: CSP auto-noncing and TT policy creation

Example:

```
url = '//example.com/ab.js';
s = document.body.createElement('script');
s.src = url;
```

Google **safevalues** github.com/google/safevalues

```
import {trustedResourceUrl} from 'safevalues';
import {safeScriptEl} from 'safevalues.dom';
url = trustedResourceUrl`//example.com/ab.js`;
s = document.body.createElement('script');
safeScriptEl.setSrc(s, url);
```

Securing Newly Written Code with Safe Coding

Shifting **responsibility** for web security **away from developers** to security engineers and ultimately **to frameworks** enables to eliminate entire vuln classes in new code

Scalability	*Securing Newly Written Code (Green Field)*	*Who's Responsible*
	No process / developers are on their own	Developers
	Security reviews, documentation, dev training	Security Engineers
	Safe-by-default APIs	Security Engineers
	Safe-by-default frameworks / guardrails	Framework

- **Framework enforcement security invariants** like safe-by-default APIs and web platform security features
- Framework configuration itself is safe-by-default
- Provide the (non security) features developers need/want

Securing Newly Written Code with Safe Coding

Shifting **responsibility** for web security **away from developers** to security engineers and ultimately **to frameworks** enables to eliminate entire vuln classes in new code

Scalability	*Securing Newly Written Code (Green Field)*	*Who's Responsible*
	No process / developers are on their own	Developers
	Security reviews, documentation, dev training	Security Engineers
	Safe-by-default APIs	Security Engineers
	Safe-by-default frameworks / guardrails	Framework

- **Framework enforcement security invariants** security features
- Framework configuration itself is safe-by-default
- Provide the (non security) features developers need/want

Security becomes an ambient property of the system.
Developers don't have to think about it!
No need for web security launch reviews 🎉

Safe defaults in our hardened frameworks

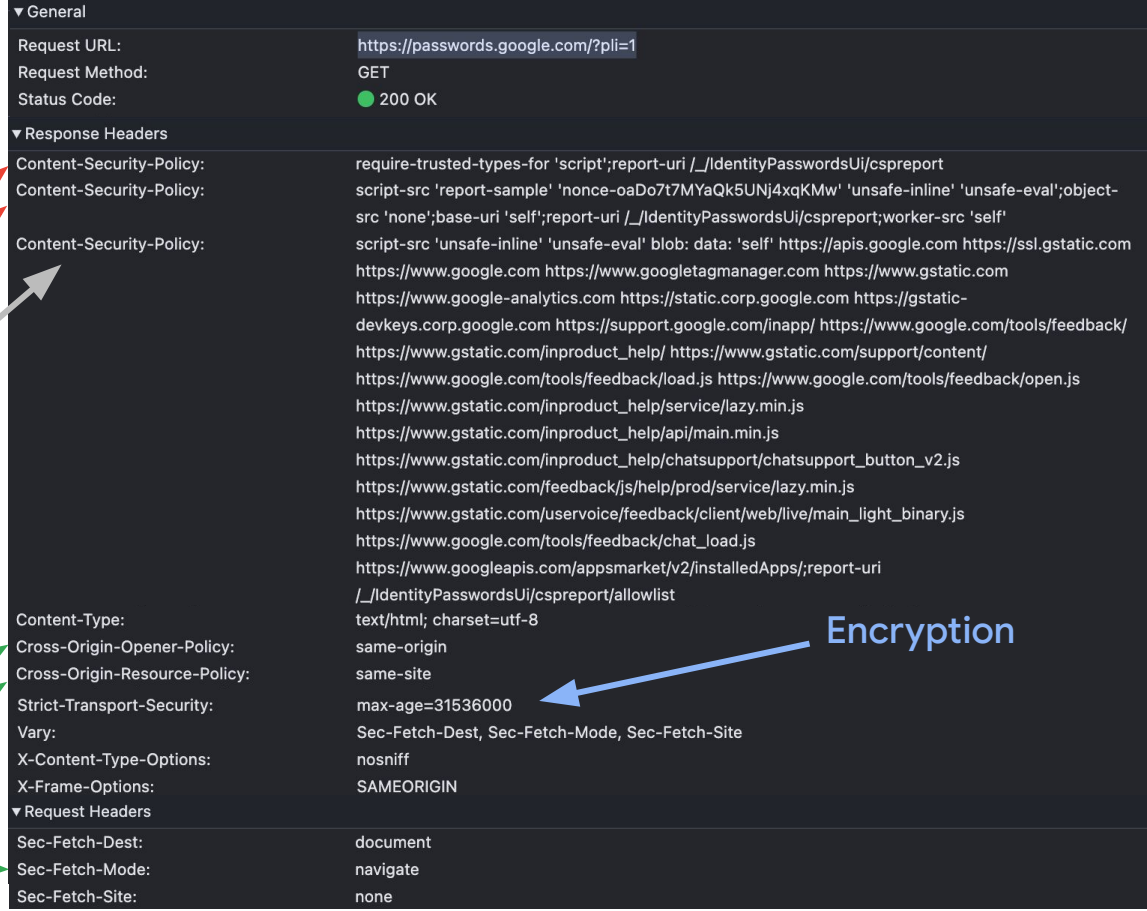
These frameworks power over 300 external facing web applications like Google Cloud Console, Photos, Gemini, Passwords, ...

A typical response* from our hardened web frameworks

XSS (strict CSP + TT)

Block 3rd party scripts (allowlist CSP)
Mitigates supply chain risk and is not intended to mitigate XSS

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



* some request/response headers are omitted to fit the slide

A typical response* from our hardened web frameworks

XSS (strict CSP + TT)

Block 3rd party scripts (allowlist CSP)
Mitigates supply chain risk and is not intended to mitigate XSS

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

The Hacker News

Subscribe – Get Latest News

Home Cyber Attacks Vulnerabilities Store Contact

Polyfill[.]io Attack Impacts Over 380,000 Hosts, Including Major Companies

Jul 05, 2024 Newsroom

The supply chain attack targeting the widely-used Polyfill[.]io JavaScript library is broader in scope than previously thought, with new findings from Censys showing that over 380,000 hosts are embedding a polyfill script linking to the malicious domain as of July 2, 2024.

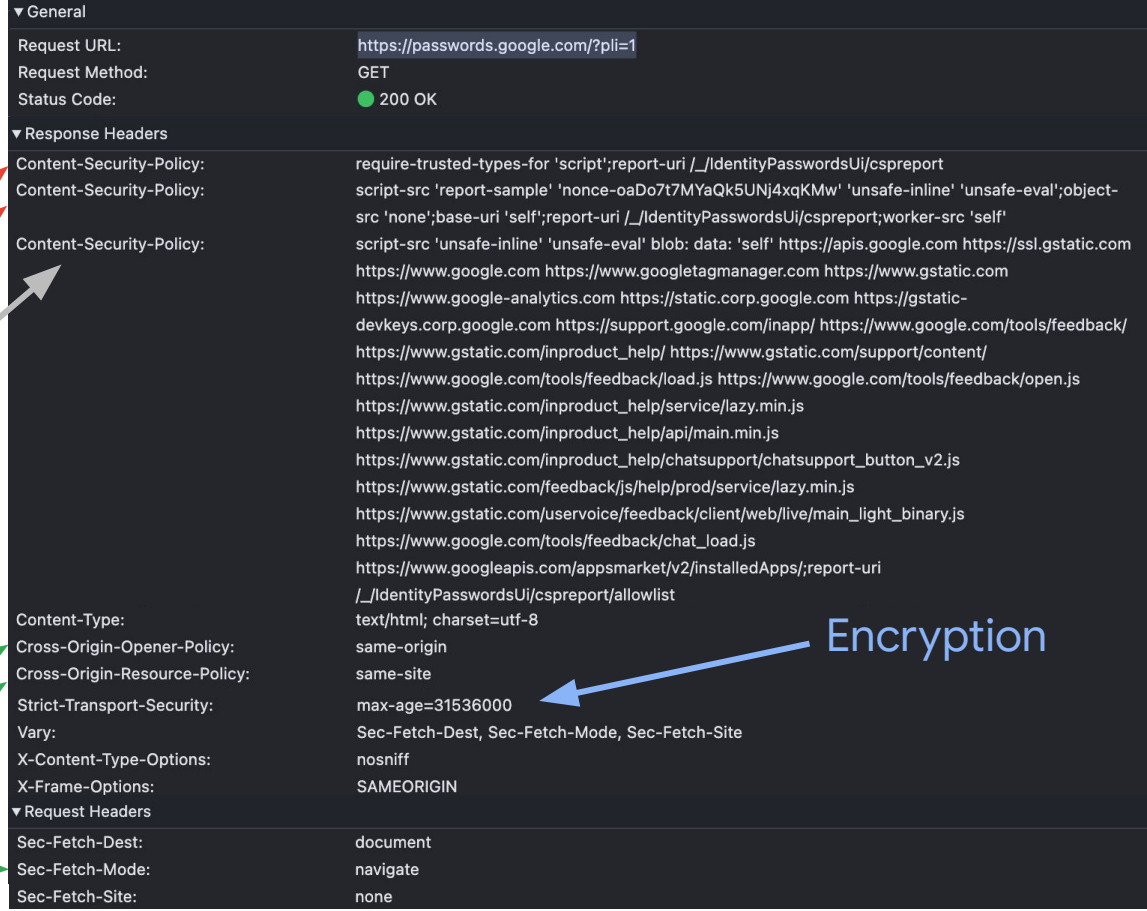
A typical response* from our hardened web frameworks

XSS (strict CSP + TT)

Block 3rd party scripts (allowlist CSP)

Mitigates supply chain risk and is not intended to mitigate XSS

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



* some request/response headers are omitted to fit the slide

The path to zero XSS

with web platform security features and safe coding

Runtime protections (web platform)

- Trusted Types
  `DOM XSS`
- Strict Content Security Policy (CSP)
  Stored/reflected XSS | `DOM XSS`

Compile-time protections (safe coding)

- Safevalues ("compile-time Trusted Types")
  `DOM XSS`
- Strict Contextual Escaping Templates
  Stored/reflected XSS
- Safe Response Types (server-side)
  Stored/reflected XSS
- Prototype Pollution protections
  `Prototype Pollution`

Deployment guidance at
w3c.github.io/webappsec/mitigation-guidance

The path to zero XSS

with web platform securit

Runtime protections are highly effective to overcome limitations of compile-time checks (e.g. dynamic code loading)

Runtime protections (web platform)

- **Trusted Types**
  `DOM XSS`
  web.dev/articles/trusted-types
- **Strict Content Security Policy (CSP)**
  Stored/reflected XSS | DOM XSS
  web.dev/articles/strict-csp

Trusted Types and strict CSP complement each other very well and cover a very large part of the XSS attack surface

Compile-time protections (safe coding)

- **Safevalues** ("compile-time Trusted Types")
  `DOM XSS`
  github.com/google/safevalues
- **Strict Contextual Escaping Templates**
  Stored/reflected XSS
- **Safe Response Types** (server-side)
  Stored/reflected XSS
- **Prototype Pollution protections**
  `Prototype Pollution`

github.com/tc39/proposal-symbol-proto

Deployment guidance at
w3c.github.io/webappsec/mitigation-guidance

The path to zero XSS

with web platform security features and safe coding

Runtime protections (web platform)

Trusted Types

a. Deploy in report-only mode
   + Get exhaustive list of XSS-prone DOM operations
   + Easy to deploy (ideally on a fraction of traffic)
   + No backward compatibility risk
   - No runtime protection

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

b. Deploy in enforcement mode with default policy
   + Runtime protection / dangerous DOM sinks disabled
   + Works also if not all (3p) code can be refactored
   - Default policies can be difficult to get right

```
Content-Security-Policy:
    require-trusted-types-for 'script';
    trusted-types default * 'allow-duplicates';
```

c. Deploy in enforcement mode
   + Runtime protection / dangerous DOM sinks disabled
   + Ensures that **all** dangerous DOM sinks are refactored
   - Difficult to deploy

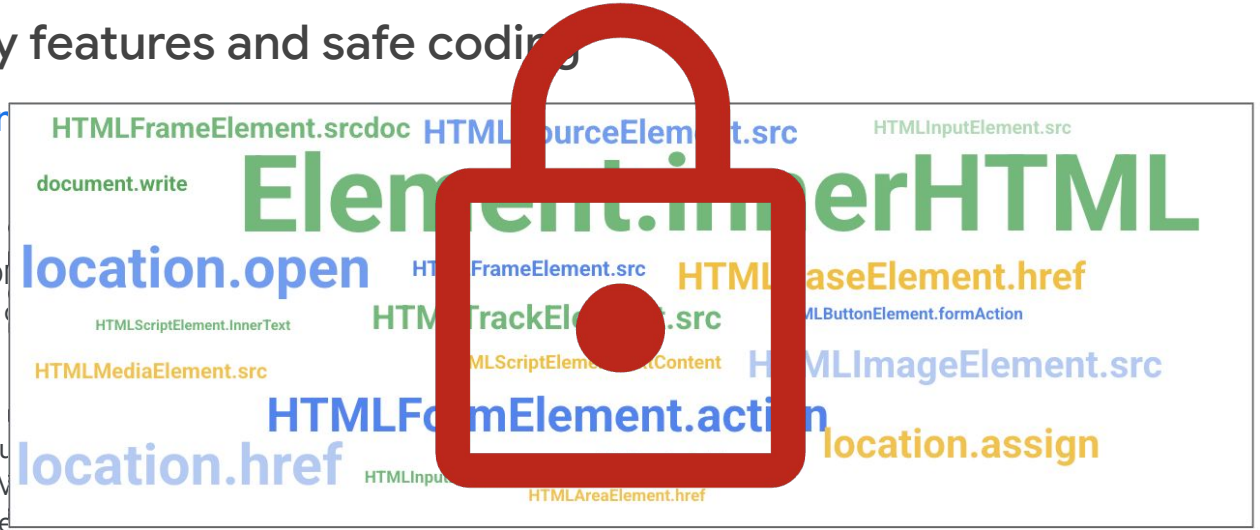
```
Content-Security-Policy:
    require-trusted-types-for 'script';
```

Deployment tips:
- web.dev/articles/trusted-types
- w3c.github.io/webappsec/mitigation-guidance/TrustedTypes/
- bughunters.google.com/blog/6037890662793216/enabling-trusted-types-in-a-complex-web-application-a-case-study-of-appsheet

The path to zero XSS

with web platform security features and safe codi

Runtime protections (web platfor



Trusted Types

a. Deploy in report-only mode
   + Get exhaustive list of XSS-prone DOI
   + Easy to deploy (ideally on a fraction
   + No backward compatibility risk
   - No runtime protection

b. Deploy in enforcement mode with defau
   + Runtime protection / dangerous DOM
   + Works also if not all (3p) code can be
   - Default policies can be difficult to get right

c. Deploy in enforcement mode
   + Runtime protection / dangerous DOM sinks disabled
   + Ensures that **all** dangerous DOM sinks are refactored
   - Difficult to deploy

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

Deployment tips:
- web.dev/articles/trusted-types
- w3c.github.io/webappsec/mitigation-guidance/TrustedTypes/
- bughunters.google.com/blog/6037890662793216/enabling-trusted-types-in-a-complex-web-application-a-case-study-of-appsheet

The path to zero XSS

with web platform security features and safe coding

Runtime protections (web platform)

Strict Content Security Policy – web.dev/articles/strict-csp

a. Deploy nonce or hash based CSP with *'strict-dynamic'*
   + Opt-in to disable many common XSS sinks in the browser (innerHTML, javascript: URIs, etc.)
   + Works also if not all (3p) code cannot be refactored
   + Good trade off between refactoring work and security benefits
   + Very effective at mitigating XSS at Google
   - 'strict-dynamic' trust propagation leaves some XSS

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

b. Deploy nonce or hash based CSP
   + Strong stored, reflected, and DOM XSS protection
   + Auto-noncing via github.com/google/safevalues
   - Doesn't work if not all (3p) code can be refactored
   - Difficult to deploy

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

c. Deploy "auto-hashing" CSP **[experimental]**
   + Works out of the box / no refactoring required
   + Mitigates common SPA XSS (javascript:, dangerouslySetInnerHTML)
   - Does not protect against classical stored/reflected XSS (server-side rendering)
   - Prototype (more soon): github.com/google/strict-csp/tree/main/strict-csp-html-webpack-plugin

```
Content-Security-Policy:
    script-src 'sha256-...' 'strict-dynamic'
               'unsafe-hashes';
    base-uri 'none'
```

CSP Evaluator

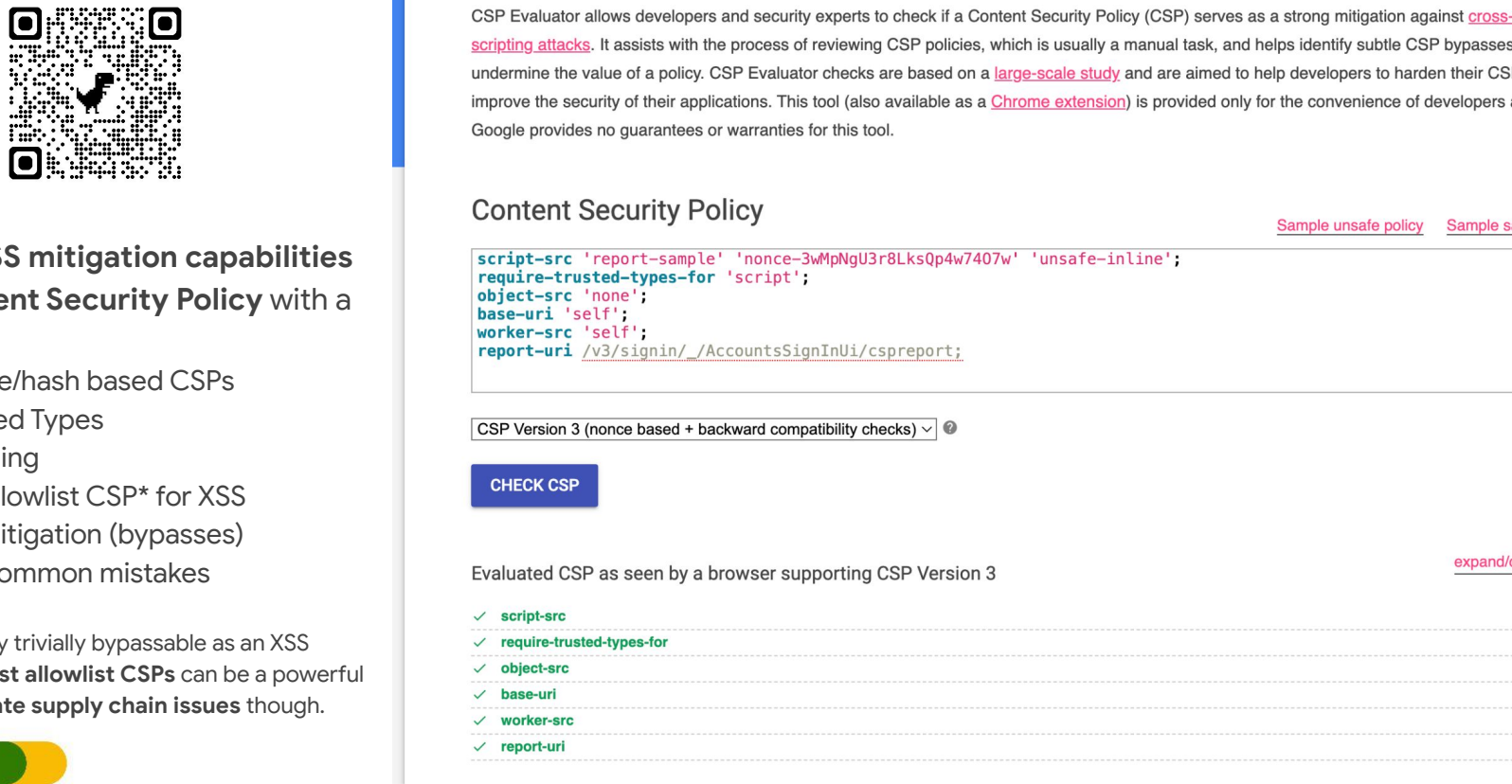
csp-evaluator.withgoogle.com

goo.gle/csp-evaluator-extension

Assess **XSS mitigation capabilities** of a **Content Security Policy** with a focus on:

- Nonce/hash based CSPs
- Trusted Types
- Avoiding
  - Allowlist CSP* for XSS mitigation (bypasses)
  - Common mistakes

* While usually trivially bypassable as an XSS mitigation **host allowlist CSPs** can be a powerful tool to **mitigate supply chain issues** though.



The three ingredients for obliterating entire (web) vulnerability classes

01

Address Root Causes

02

Safe Coding Approach

03

Fix Legacy Code

Fixing Legacy Code at Scale

Addressing legacy issues across hundreds of web applications at scale can be achieved by having security engineers conduct large scale changes with the support of tooling.

Scalability	*Securing Existing Code (Brown Field)*	*Who's Responsible*
	Fix vulnerabilities reactively when reported	Developers

Fixing Legacy Code at Scale

Addressing legacy issues across hundreds of web applications at scale can be achieved by having security engineers conduct large scale changes with the support of tooling.

Scalability	*Securing Existing Code (Brown Field)*	*Who's Responsible*
	Fix vulnerabilities reactively when reported	Developers
	File lots of bugs and ask the developer to deploy security features	Developers

Issues are identified proactively by the security team

Fixing Legacy Code at Scale

Addressing legacy issues across hundreds of web applications at scale can be achieved by having security engineers conduct large scale changes with the support of tooling.

Scalability	*Securing Existing Code (Brown Field)*	*Who's Responsible*
	Fix vulnerabilities reactively when reported	Developers
	File lots of bugs and ask the developer to deploy security features	Developers
	File lots of bugs, but also provide easy to use and preconfigured safe APIs and good documentation	Developers

Example:

- Filing 100s of bugs about deploying a CSP to mitigate XSS will likely yield bypassable CSPs
  → Provide good documentation about what a strict CSP is: web.dev/articles/strict-csp
  → Provide a CSP module that sets a pre-configured strict CSP and adds nonces to scripts

Fixing Legacy Code at Scale

Addressing legacy issues across hundreds of web applications at scale can be achieved by having security engineers conduct large scale changes with the support of tooling.

Scalability	*Securing Existing Code (Brown Field)*	*Who's Responsible*
	Fix vulnerabilities reactively when reported	Developers
	File lots of bugs and ask the developer to deploy security features	opers
	File lots of bugs, but also provide easy to use and preconfigured safe APIs and good documentation	opers
	Backport features across entire frameworks at scale	Security Engineers

Needs good tooling:
- inventory
- prioritization
- automation

However, fixing legacy code at scale can work!

Our approach allowed us to deploy a large set of security features at scale.

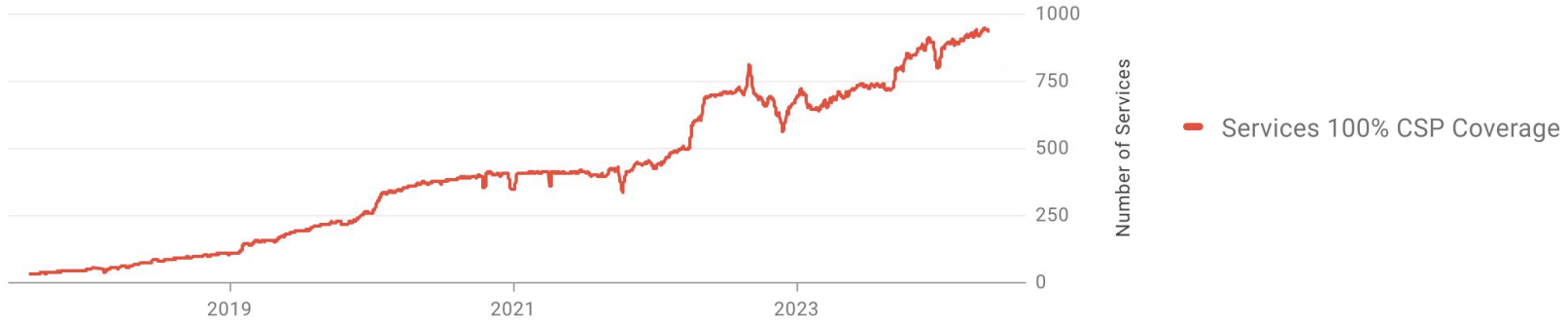
Just in 2023 we got:

- **166** services are newly enforcing Strict CSP
- **176** services are newly enforcing Trusted Types
- **347** services are newly enforcing Resource Isolation Policy
- **1079** JavaScript builds have new static guarantees that all transitive dependencies satisfy our safe coding conformance checks
- **438** legacy DOM XSS sink assignments were removed
- **160** services are newly enforcing SameSite cookies

https://bughunters.google.com/blog/5896512897417216/a-recipe-for-scaling-security

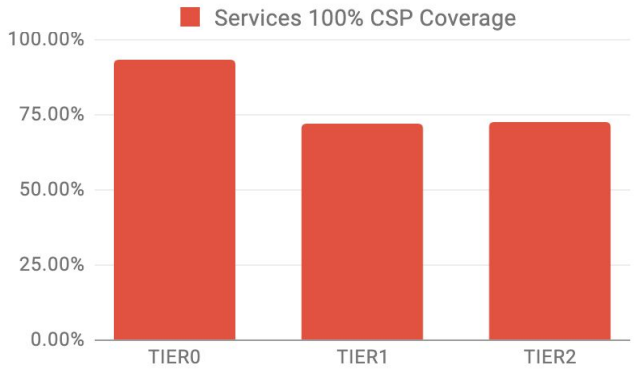
However, fixing legacy code at scale can work!

Strict CSP (nonce/hash based) enforced on **>900 web applications** (includes internal apps)



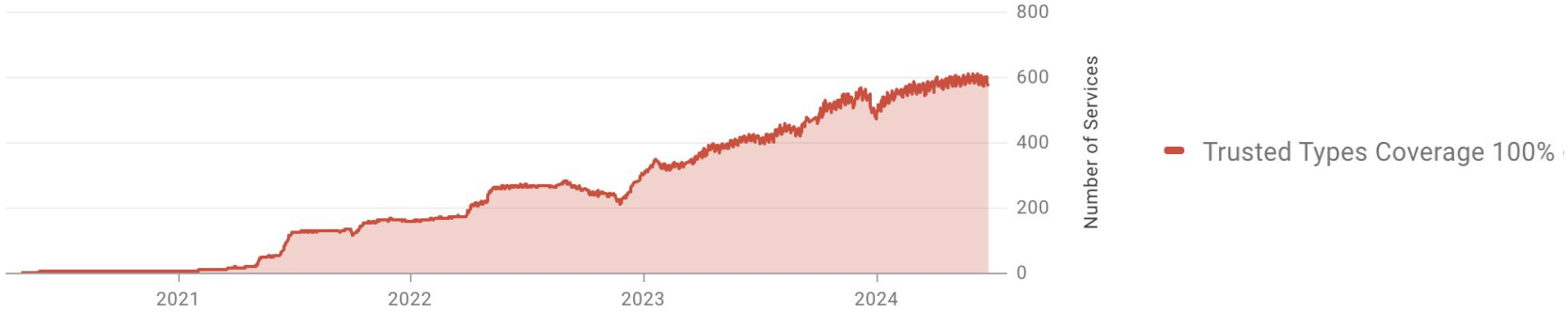
Highest coverage on most sensitive web applications

- >90% on **TIER0**
- Gmail, Account Login, Cloud Console, Docs/Drive, Meet, Photos, etc.
- Many XSS vulnerabilities **mitigated** by CSP



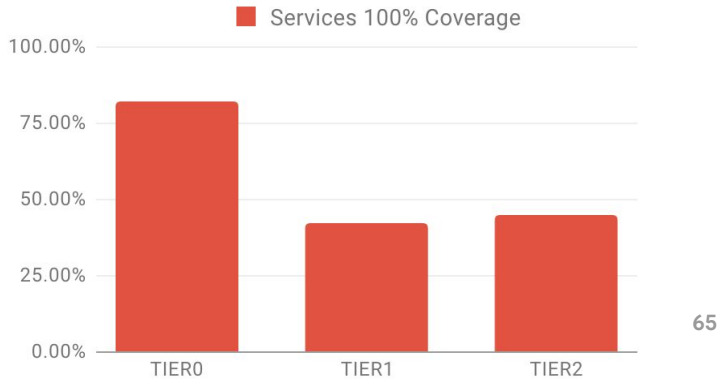
However, fixing legacy code at scale can work!

Trusted Types enforced on **~600 web applications** (includes internal apps)



Highest coverage on most sensitive web applications

- >75% on **TIER0**
- Gmail, Account Login, Cloud Console, Docs/Drive, Meet, Photos, etc.



Howe

Trusted Ty

k!

Coverage 100%

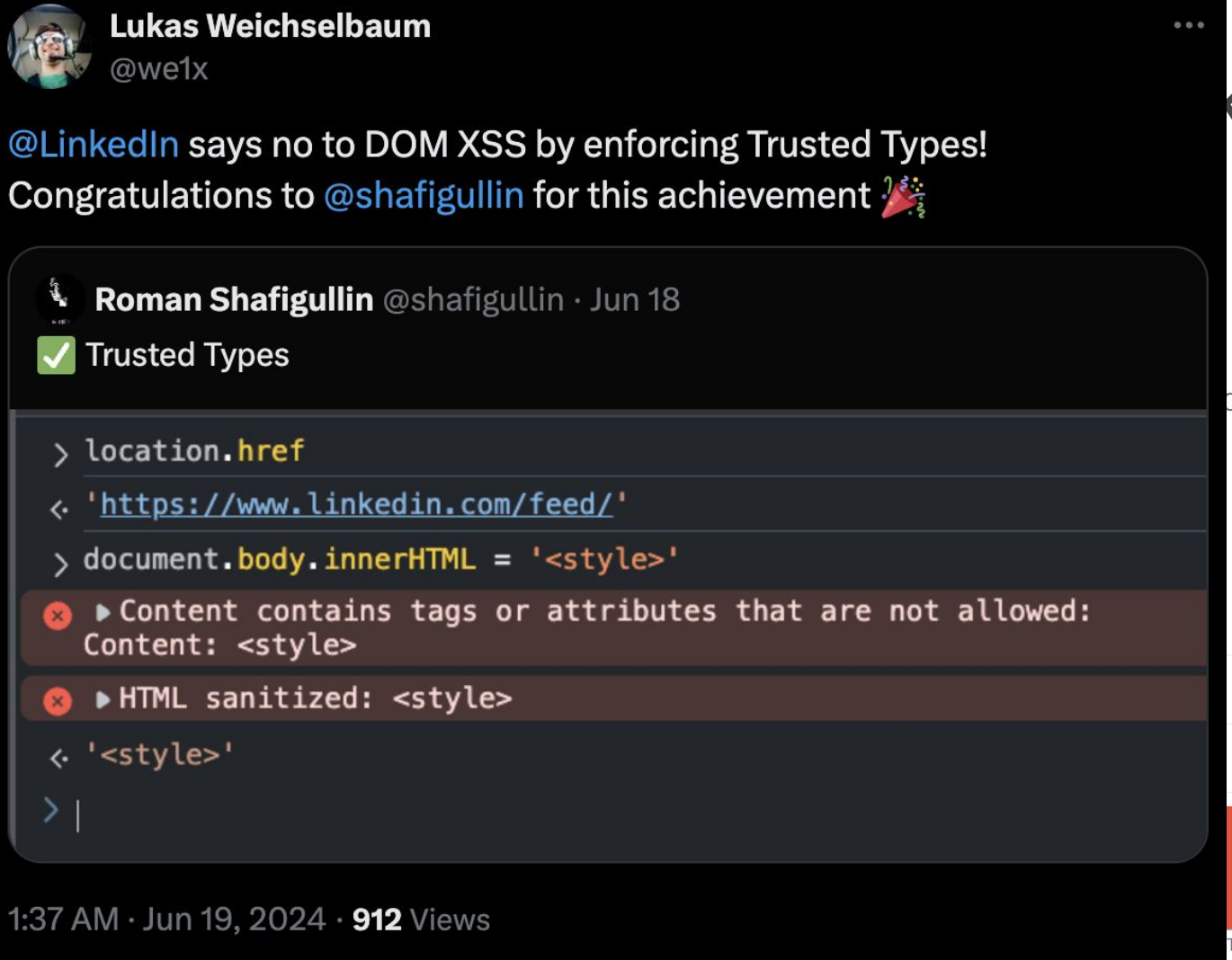
Highest co

- >75%
- Gmai

Docs

66

TIER2



Fixing Legacy Code at Scale

01 **Inventory of web applications and sensitive domains (=trust boundary) to:**
- Identify rollout targets
- Prioritize highly sensitive web applications
- Monitor progress

02 **Assess compatibility of security features and identify blockers at scale**
- **Runtime**: Monitor native & synthetic violation reports (e.g. CSP reports via Reporting API)
- **Compile-time**: Identify breaking changes at compile time.
  → Identify and fix most common blockers

03 **Identify and fix common blockers & opt-in applications at scale**
- Patch large numbers of web applications simultaneously

04 **Experiment system**
- Gradual ramp up of a change at runtime across a large set of applications
- Allowing fast ramp down if needed

05 **Regression prevention**
- **Runtime**: Monitor security feature coverage based on traffic
- **Compile-time**: Prevent regressions via compiler checks or pre-submit checks

Fixing Legacy Code at Scale – Public resources

01 **Inventory** of web applications and sensitive domains (=trust boundary) to:

- **Security Signals** – Framework to collect and process aggregated and anonymized traffic logs across Google
  goo.gle/security-signals [NEW – research paper published last week]
- **Domain Tiers** – Concept to assign a sensitivity factor/score to each domain that is hosting a web application
  goo.gle/domain-tiers

02 **Assess compatibility** of security features and identify blockers at scale

- Reporting API
  bit.ly/reporting-api
- **tsec** – Perform a basket of security checks to assert Trusted Types compatibility and find possible XSS issues
  github.com/google/tsec

03 Identify and fix **common blockers** & opt-in applications at scale

- Rosie – Send out hundreds of changes for review, run all automated tests
  goo.gle/rosie

04 **Experiment system**

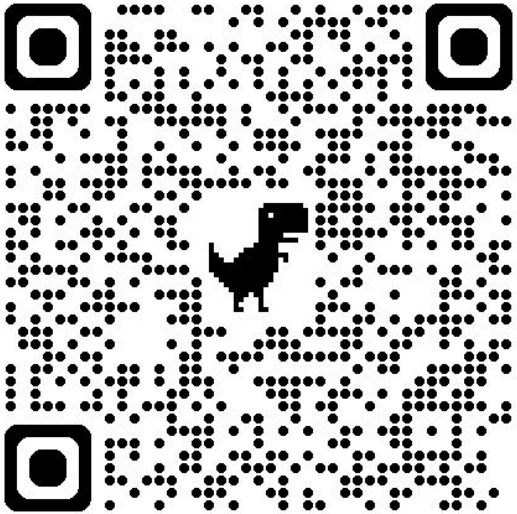
- Google SRE Book
  goo.gle/sre-book

05 **Regression prevention**

- **Security Signals:** See above
- **tsec (and others):** See above

Security Signals Research Paper

Published last week



Google Research | Who we are | Research areas | Our work | Programs & events | Careers | Blog

Home > Publications >

Security Signals: Making Web Security Posture Measurable At Scale

Michele Spagnuolo · David Dworken · Artur Janc · Santiago (Sal) Díaz · Lukas Weichselbaum ·
(2024) (to appear)

Google Scholar | Copy Bibtex

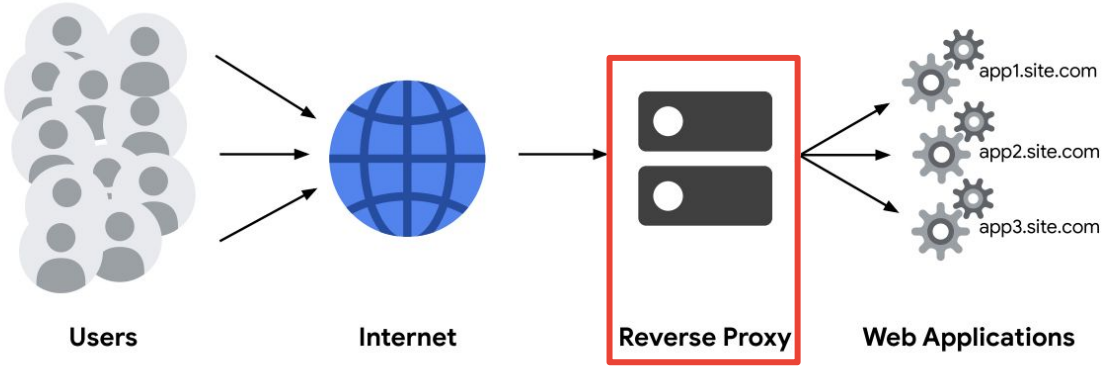
Abstract

The area of security measurability is gaining increased attention, with a wide range of organizations calling for the development of scalable approaches for assessing the security of software systems and infrastructure. In this paper, we present our experience developing Security Signals, a comprehensive system providing security measurability for web services, deployed in a complex application ecosystem of thousands of web services handling traffic from billions of users. The system collects security-relevant information from production HTTP traffic at the reverse proxy layer, utilizing novel concepts such as synthetic signals augmented with additional risk information to provide a holistic view of the security posture of individual services and the broader application ecosystem. This approach to measurability has enabled large-scale security improvements to our services, including allowing prioritized rollouts of security enhancements and the implementation of automated regression monitoring; it has proven valuable for security research and prioritization of defensive work. Security Signals addresses shortcomings of prior web measurability proposals by tracking a comprehensive set of security properties relevant to web applications, and by extracting insights from collected data for use by both security experts and non-experts. We believe the lessons learned from the implementation and use of Security Signals offer valuable insights for practitioners responsible for web service security, potentially inspiring new approaches to web security measurability.

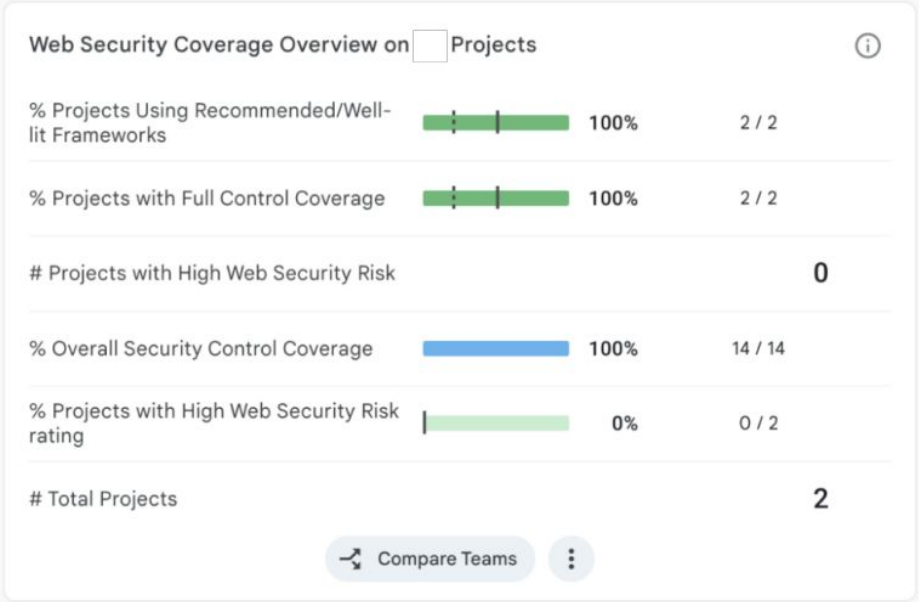
Security Signals

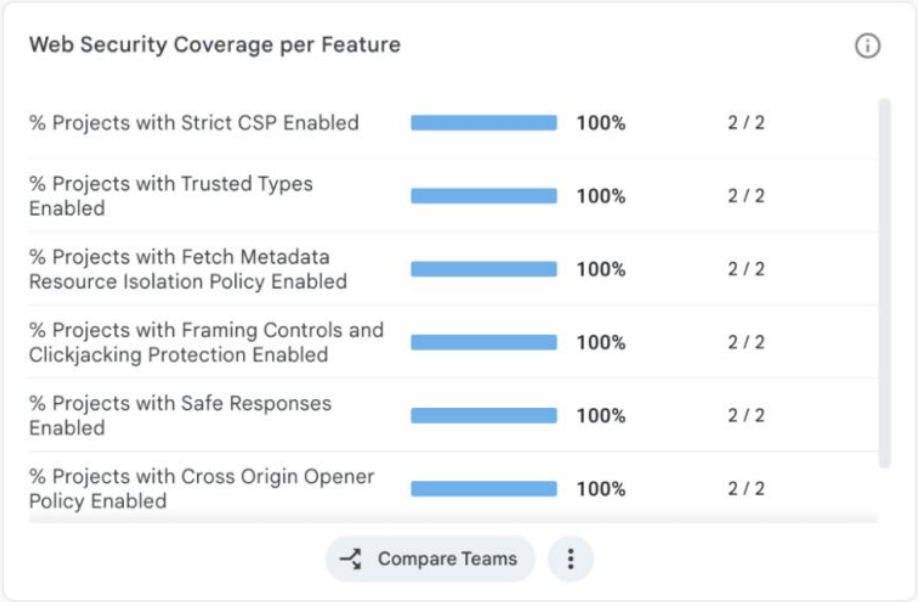
Framework to collect and process aggregated and anonymized traffic logs across Google

- security relevant HTTP headers
- custom application security properties exposed through active instrumentation
- enriched with auxiliary data like code location, code ownership, etc.



Security Signals





Dessert

Key Takeaways

Key Takeaways

01 Shift responsibility away from individual developers to security engineers and the development ecosystem

02 Address vulnerabilities as close to their root cause as possible.

03 Data and tooling are important to drive change at scale without causing reliability issues

Key Takeaways

01 **Shift responsibility away from individual developers to security engineers and the development ecosystem**

- Making every developer a security expert doesn't scale.
- Instead security should be built into the developer ecosystem (web platform, frameworks, developer environment)
- Creating these secure ecosystems requires Security Engineers to become deep experts in software development

Key Takeaways

02 **Address vulnerabilities as close to their root cause as possible.**

- Many web application vulnerabilities are caused by lack of safety in the web platform.
- Luckily, the web platform is malleable and improving.
- Many powerful web platform security features are available to tackle pressing security issues like XSS. In particular a strict CSP and Trusted Types are very effective.
- Web platform discussions are open and feedback from developers is appreciated

Key Takeaways

03 **Data and tooling are important to drive change at scale without causing reliability issues**

- Harness the power of data and purpose-built tools to implement security measures at scale, without compromising system reliability or hindering development velocity
- Build a web application inventory to identify and track sensitive web applications
- Leverage compile-time and runtime data to assess compatibility of a new security control across multiple web applications simultaneously
- Prevent regressions either through new compile-time checks run before new code can be submitted or by monitoring for regressions in runtime traffic.

Together, let's strive to build a web that's **secure by default, not by chance.**

A web where web developers are free to innovate, and not having to defend.

Just as memory safety isn't a concern in modern languages,
let's make web vulnerabilities like XSS relics of the past.

Mahalo 🤙

Lukas Weichselbaum

Google Information Security Engineering

lwe@google.com

@we1x

lweichselbaum