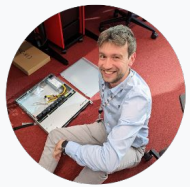
Security Signals

Making Web Security Posture Measurable At Scale

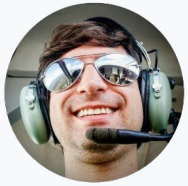
2025 Workshop on Measurements, Attacks, and Defenses for the Web


Michele Spagnuolo


David Dworken


Artur Janc


Sal Díaz


Lukas Weichselbaum

Google's Approach to Web Security

Web Security is hard, especially at Google

Possibly the **largest** deployment of web applications in the world:

- > 8,000 web services
- across ~1,000 registrable domains
- processing trillions of HTTP requests from billions of daily users

... serving web pages generated by a **heterogeneous ecosystem** with:

- many programing languages, e.g. Java, C++, Python, Go
- many distinct web frameworks and templating systems
- billions of line of code, thousands of third-party libraries

... changing all the time.

Secure-by-Design or Fail to Scale

With a large-scale, rapidly evolving codebase, fixing vulnerabilities one-by-one is neither efficient nor scalable.

To make security a property of the ecosystem/developer infrastructure, we need:

- **secure tools, libraries,** and **frameworks** (aka "**well-lit path**")
- **guidelines** and **recommendations** for developers to keep them on the well-lit path
- **security review required for opt-outs**
- **regression monitoring** and continuous remediation

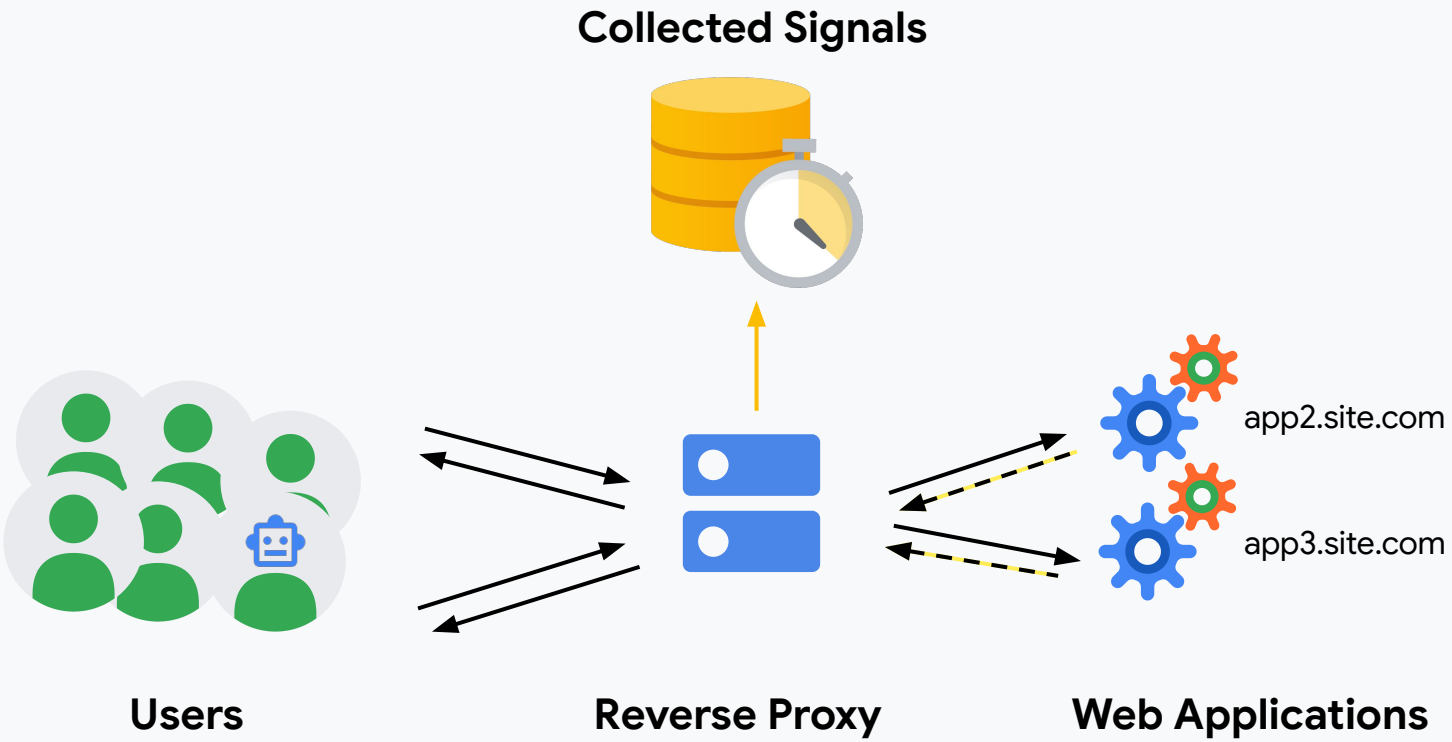
Security Signals

Security Signals is a framework to **collect security-relevant data** (aka *signals*) about a web ecosystem from **production HTTP traffic** to:

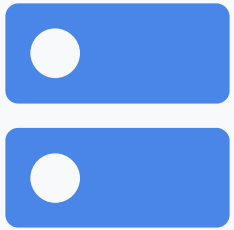
- provide **visibility** into security stance of the web infrastructure
- determine if certain applications are inherently "**secure-by-design**" from broad classes of vulnerabilities
- enable **security remediations** and continual upkeep of the well-lit path
- provide **continuous monitoring** of security controls and assurance of the alignment to the "secure-by-design" principles

Collecting Signals

Collecting Security Signals



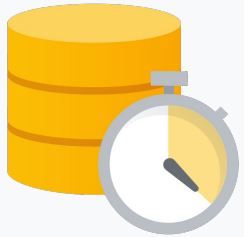
Reverse Proxies as Observability Hooks



Reverse proxies route every user connection to a requested web service running on a selected machine. Reverse proxies often have additional features:

- load balancing to optimize selection of a web service
- termination of HTTPS traffic and establishing
- protection against DoS attacks
- **centralized logging** ⭐
- etc.

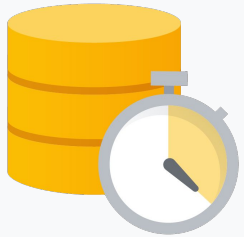
Collecting Data: Challenges



Google processes trillions of HTTP requests from billions of web users daily. To ensure the privacy of users, and the feasibility and quality of generated insights:

- **web traffic is sampled** with a rate of usually up to 1%, and 10% for internal traffic
- **sensitive data** and request/response bodies **are not collected**
- a very short **retention time**
- **audited log access**, and only **justified human access**

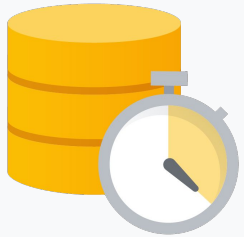
Collecting HTTP Request & Response Data



- HTTP method
- Destination host
- Redacted path and no query parameters!
- Status code
- Response MIME type
- `Referrer-Policy`
- `Cache-Control`
- User agent: browser name and major version
- Cookie metadata

The HTTP request/response bodies are not collected.

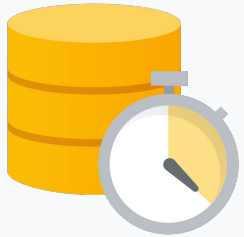
Collecting Security-Related HTTP Headers



HTTP request and response headers contain all kinds of security info:

- `Content-Security-Policy`
- `Cross-Origin-Embedder-Policy`
- `Cross-Origin-Opener-Policy`
- `Cross-Origin-Resource-Policy`
- `Sec-Fetch-*`
- `Strict-Transport-Security`
- `X-Content-Type-Options`
- `X-Frame-Options`
- …

Synthetic Security Signals

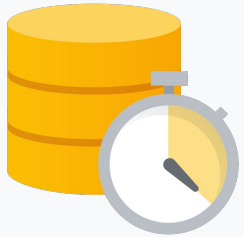


Synthetic signals are a core capability of the Security Signals approach. They contain additional metadata that is not normally included in the HTTP response.

They are:

- generated by **instrumented web frameworks**
- using an internal-only **`X-Google-Security-Signals`** HTTP response header
- collected when passing the reverse proxy...
- ... and dropped before sending to the world.

Collected Synthetic Security Signals



For example:

- `FRAMEWORK`: The serving web tech stack.
- `ACTION`: A pointer to the method/function generating the web response.
- `TEMPLATE`: The server-side templating system that generates HTML output.
- `BUILD`: Information about the application's build environment.
- `CSRF`: The presence of Cross-Site Request Forgery protections to verify if an CSRF check was carried out by the backend on state changing requests.
- `SEC_FETCH`: The presence of server-side isolation policies to assess if Fetch Metadata isolation policies were applied to prevent cross-site attacks.

Auxiliary Data

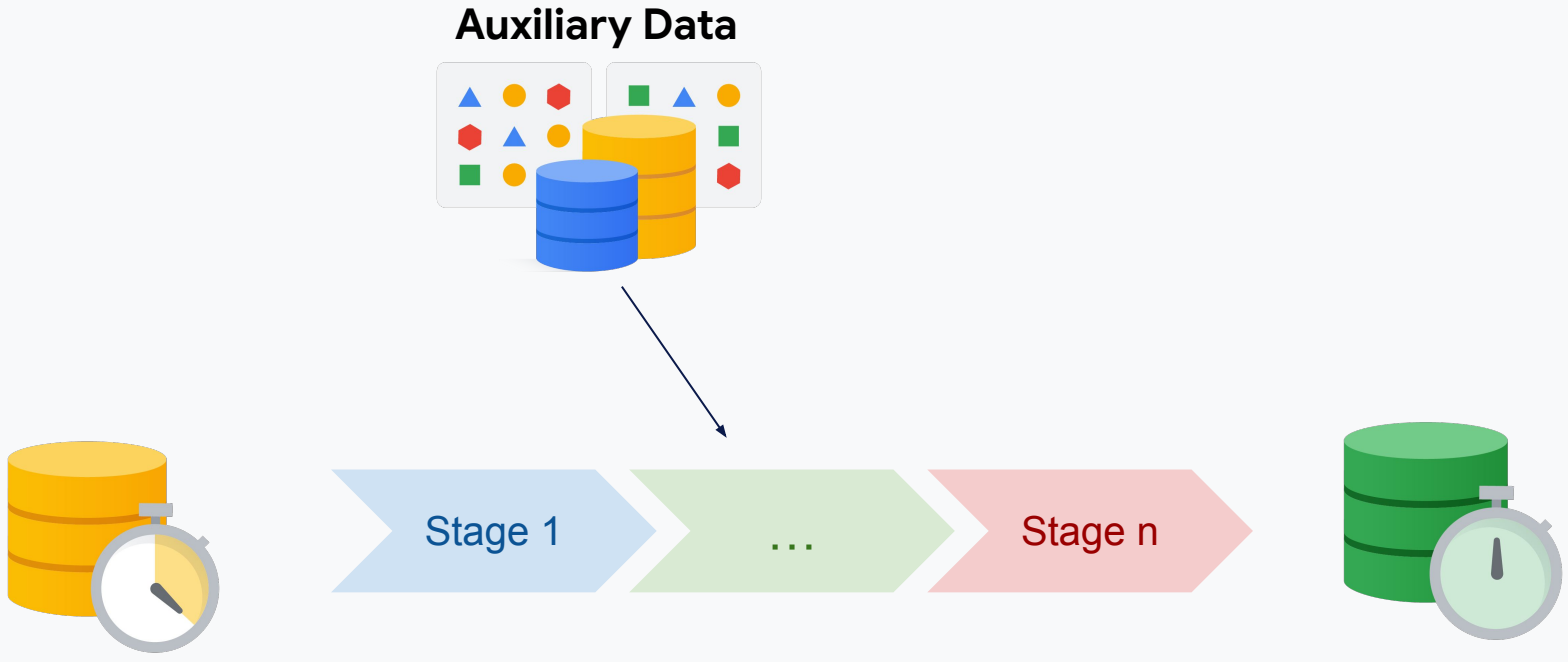
Auxiliary data, collected from internal databases, enriches security signals with information about:

- the **production** environment,
- product and **ownership** information,
- risk/**exposure**: sensitivity of the hosting domain (Domain Tiers),
- **source code** information, etc.

This context is crucial for streamlining **remediation efforts** and **automated bug filing.**

Processing Signals

Security Signals Pipeline



Security Signals Pipeline

Security Signals Pipeline is a Flume distributed map-reduce data processing pipeline, which:

- **reads** billions of signals from collected request/response pairs
- **reduces** their number by deduplication and initial evaluation
- **joins** them with auxiliary data (ownership, production info, external debug info)
- **persists** them in Security Signals **tables**

Cardinality Reduction

Traffic logs have billions of entries with high-cardinality dimensions, which makes them impractical to query. The pipeline reduces cardinality by aggregating records, while maintaining data usefulness.

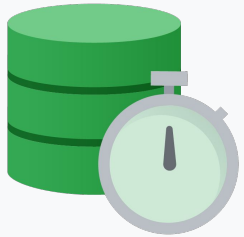
All **URL paths** are **redacted** into *path patterns* by:

1. leveraging path routing information to match and replace variable parts, e.g. from synthetic signals or per-service infrastructure configurations (API definition)

2. on remaining paths, using filtering rules based on a manually curated set of well-known high-entropy paths

3. on the remaining paths, executing an ML model

Security Signals Tables



The **output** of the Pipeline is:

- persisting only **aggregated** and **de-identified** data
- **accessed** by approved engineers and services
- **monitored** to detect any anomalies in quality of data
- **retained** for 30 days

Targeted Security Research & Remediations

Example: Cross-Site Request Forgery

- Web applications write code to interact with their own endpoints in reasonable ways

```
<form action="/transfer">
  <input name="destination" value="my-friend" />
  <input name="amount" value="10 $" />
```

- The attacker triggers these interactions from untrusted third-party sites

evil.com

```
<form action="//victim-bank.com/transfer">
  <input name="target" value="evil-ddworken" />
  <input name="amount" value="100000 $" />
```

Example: Cross-Site Request Forgery Remediation

CSRF token: a new piece of information that is both **unguessable** and **client-correlated** and sent with each request.

```
Csrf-token=YL9yaTsbfn
```

Example: Cross-Site Request Forgery Remediation

CSRF token: a new piece of information that is both **unguessable** and **client-correlated** and sent with each request.

```
Csrf-token=YL9yaTsbfn
```

The **remediation**:

1. introduce a new **synthetic signal**: `CSRF`
2. **refactor** web frameworks/libraries to populate `CSRF` signal whenever CSRF checks are done
3. **identify** endpoints with state-changing functionality that don't set the CSRF synthetic signal
    - If this is a vulnerability, fix it
    - If this is a missing synthetic signal, goto #2

Example: Cross-Origin Resource Sharing Remediation

Example **CORS** validation problems:

```
origin.endsWith("google.com") → evil-google.com
origin.startsWith("youtube.") → youtube.in.my.domain.com
```

Example: Cross-Origin Resource Sharing Remediation

Example **CORS** validation problems:

```
origin.endsWith("google.com") → evil-google.com
origin.startsWith("youtube.") → youtube.in.my.domain.com
```

The **remediation**:

1. **identify** CORS-enabled endpoints, including third party or untrusted origins
2. **test** CORS endpoints to identify vulnerabilities
3. **fix** discovered vulnerabilities
4. provide a centrally supported **secure-by-design** CORS implementation that mitigates these misconfigurations

Adoption of Web Security Features

Deploying Trusted Types across Alphabet

Goal: Add the following header to each HTTP response to enable Trusted Types and update relevant JavaScript code:

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

Deploying Trusted Types across Alphabet

Goal: Add the following header to each HTTP response to enable Trusted Types and update relevant JavaScript code:

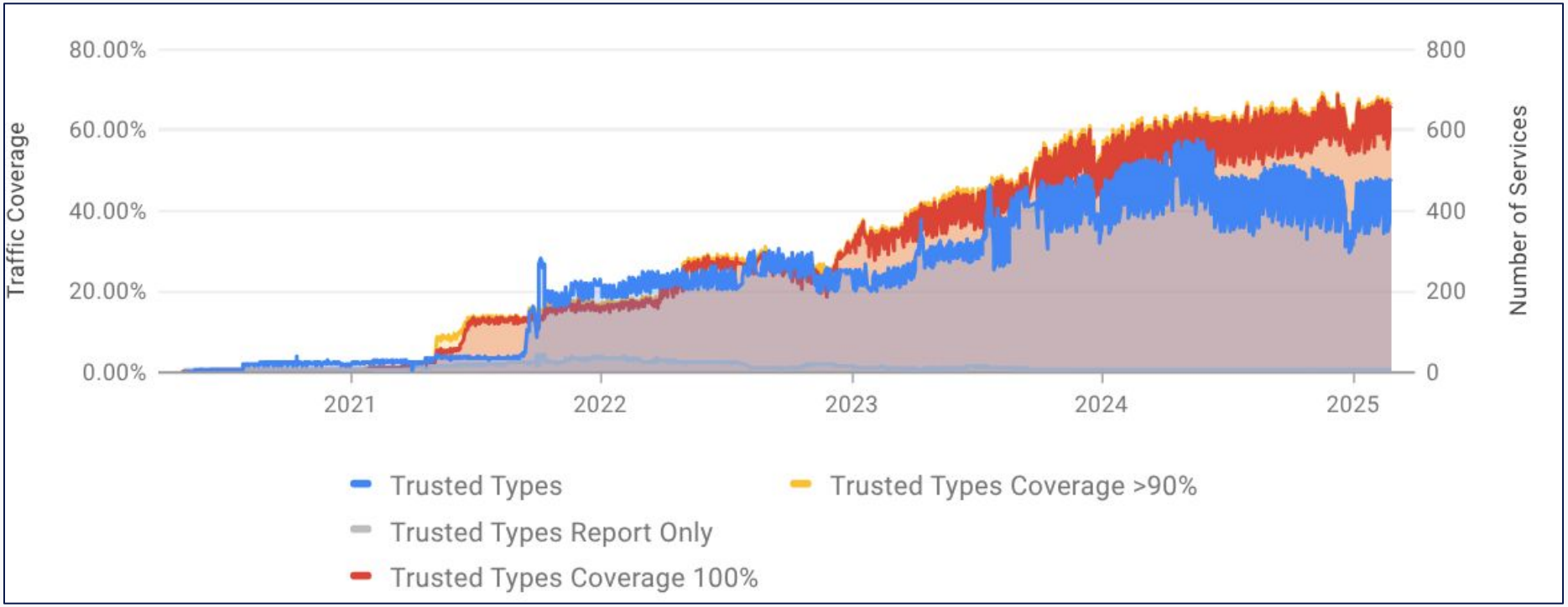
```
Content-Security-Policy: require-trusted-types-for 'script';
```

The **rollout**:

1. refactor code to adhere to Trusted Types
2. prioritize domains based on domain sensitivity classification (Domain Tiers)
3. rollout in *report-only mode*
4. large scale cross-infrastructure rollouts in batches for groups of similar services

Security Signals enabled accurate and easy measurement of our rollout progress.
⇒ In the past 2 years, **we've deployed Trusted Types to over 600 distinct services**.

Deploying Trusted Types across Alphabet



Security Signals enabled accurate and easy measurement of our rollout progress.

⇒ In the past 2 years, **we've deployed Trusted Types to over 600 distinct services.**

Measuring Runtime Dependencies & Trust Relationships

Surfacing Security-Relevant Trust Relationships

Highlighting cross-service trust relationships: Security Signals allows to identify critical services that establish trust relationships with lower-sensitivity services.

Examples:

- allowing other origins to perform CORS requests
- sending/accepting postMessage messages
- embedding scripts from other domains

Understanding transitive risks enables **comprehensive security hardening**

Surfacing AI/ML Properties

By joining the HTTP-level data with a separate DB of Remote Procedure Calls, Security Signals can identify web endpoints that depend on generative AI models.

⇒ Enables identifying where and how **models** are used and exposed

⇒ Enables **holistic assessment** of AI-enabled applications and novel AI security risks

Monitoring and Regression Detection

From Reactive Security Reviews to Scaling Security

1. **Security Invariant Monitoring**: Define expected security behaviors and configurations: CSRF, CORS, Trusted Types, ...
2. **Automated Alerting**: Detect anomalies or regressions ⇒ alert the security team ⇒ swift investigation and remediation.
3. **Automated bug filing**: Leveraging service ownership info within Security Signals, automatically file bug reports with service owners, streamlining the resolution process.

Security Signals Frontends

Target Audiences

Executives:
- Executive Dashboards

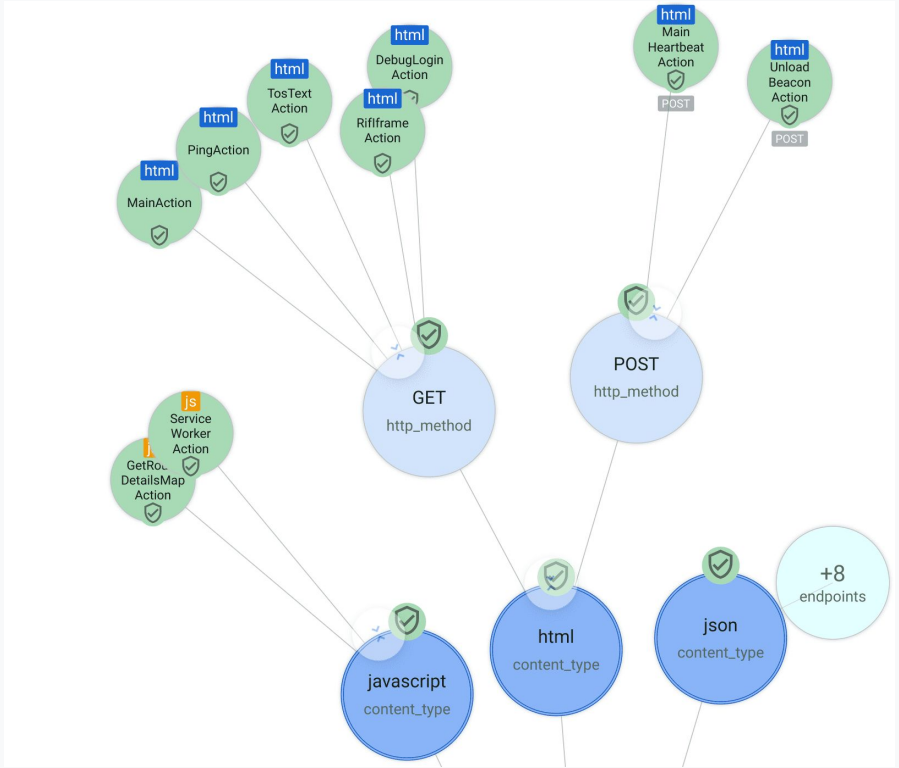
Product/Software Engineers:
- UI for Developers

Security Engineers:
- UI for Security Engineers
- Monitoring
- Raw Table Access

UI for Security Engineers

Application endpoints are presented as interactive “bubbles” organized by code package and color-coded to reflect their security status to:

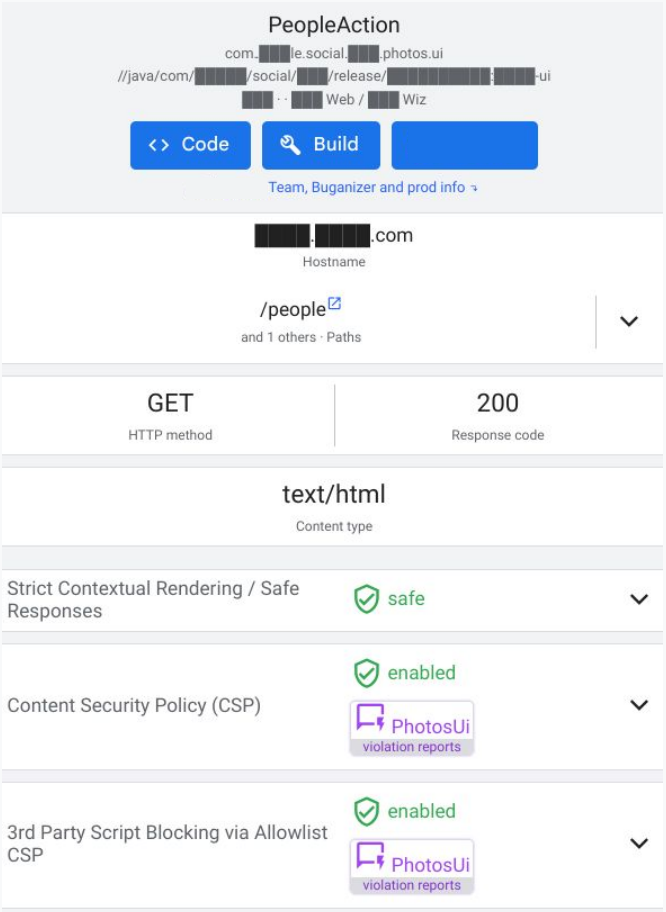
- identify security gaps
- initiate targeted remediations
- file pre-populated bugs



UI for Security Engineers

Application endpoints are presented as interactive “bubbles” organized by code package and color-coded to reflect their security status to:

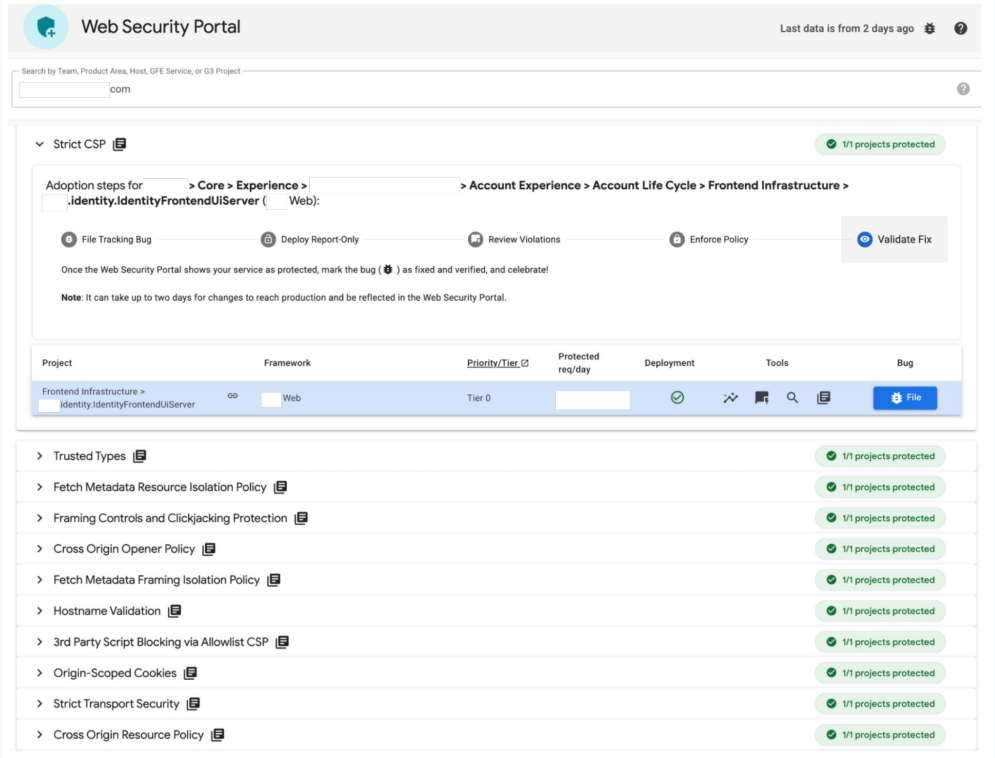
- identify security gaps
- initiate targeted remediations
- file pre-populated bugs



Web Security Portal for Product Engineers

Web Security Portal provides insights tailored to each team's application and:

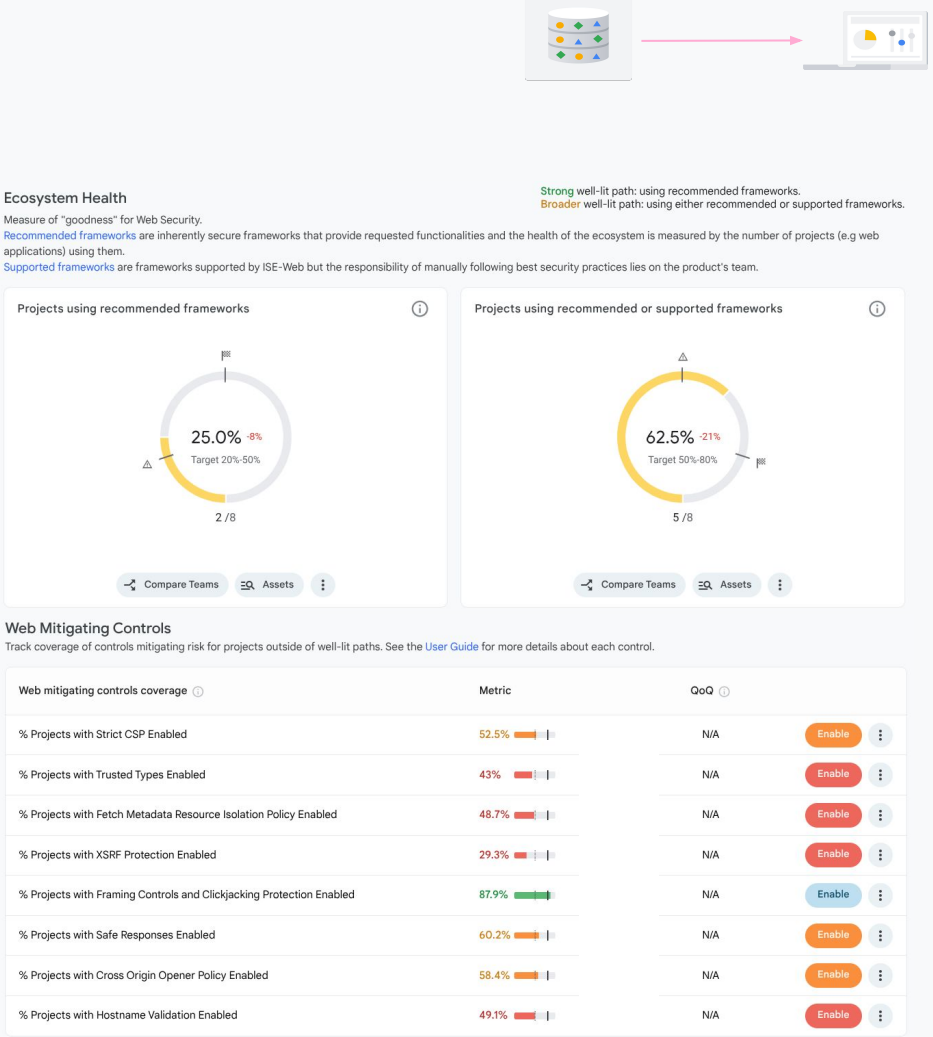
- is accessible to developers without security expertise
- shows web security posture of a product
- highlights areas for improvement
- offers service-specific recommendations



Dashboards for Executives

High-level visibility and strategic insights for executives:

- assessing overall web security posture
- identifying areas of focus
- tracking progress and quantifying impact
- risk-based prioritization
- optimizing resource allocation decisions



Thank you!
Q&A


Michele Spagnuolo


David Dworken


Artur Janc


Sal Díaz


Lukas Weichselbaum

mikispag
ddworken
aaj
saldiaz
lwe

@google.com