Security Signals: Making Web Security Posture Measurable At Scale

Michele Spagnuolo
Google, Inc.

David Dworken
Google, Inc.

Artur Janc
Google, Inc.

Santiago Díaz
Google, Inc.

Lukas Weichselbaum
Google, Inc.

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.**

I. INTRODUCTION

In recent years, governments, standards organizations and software makers have universally recognized the need to make broad, systemic security improvements to the software ecosystem. Initiatives such as secure-by-design(1), attempts to secure the software supply chain(2), and move to memory-safe languages(3) all aim to reduce the likelihood of writing vulnerable code that can be exploited by malicious actors to undermine the security guarantees of systems depending on that code.

Across all these initiatives, a common challenge is insufficient measurability: developers and security teams do not have comprehensive information that helps them understand the security posture of their services and systems. As a result, they often lack the ability to systematically implement security-relevant improvements. This has attracted increased attention of policymakers, with both the US Office of Science and Technology Policy(4) and Office of the National Cyber Director(5) issuing recommendations urging software makers and academic researchers to identify scalable strategies to collect *cybersecurity quality metrics*.

There are a number of known shortcomings of existing security measurability approaches: they frequently focus on assessing the posture of individual applications or application components, providing limited utility in complex ecosystems; they are often technology-specific (e.g. static security analysis for a given programming language), and they base assessments on specific security flaws (such as vulnerabilities discovered in the past in a given component) rather than on adherence to security best practices across the evaluated codebase.

These limitations pose significant obstacles to applying security measurability to enable meaningful defensive improvements to an organization's services. To make well-informed prioritization decisions about code hardening and related security investments, security teams need to build a thorough understanding of their organization's attack surface. This requires building a comprehensive inventory of systems, their security properties, the sensitivity of processed data and capabilities they provide.

In that context, a particularly underserved area of security measurability is the analysis of web applications. Organizations typically rely on large numbers of web services accessed through a web browser by employees and end users, either developed internally by the organization's developers, or built by external software makers and hosted as on-premise deployments. Because of the heterogeneity of web application development stacks (web applications can be built in a large number of programming languages and for each language there may exist dozens of commonly used web application development frameworks with varying security properties), it is difficult to find general, broadly applicable indicators of web application security.

This creates a significant gap in the ecosystem's security measurability efforts and requires developing new, scalable approaches for measuring the security of web services.

A. Contribution

In this paper, we outline the design of Security Signals: scalable infrastructure to collect an array of runtime security quality metrics—including a number of custom application security properties exposed through active instrumentation of web services—currently deployed in a large-scale, heterogeneous web application ecosystem, comprising more than 8000 web services built using a wide variety of programming languages and frameworks. These services are hosted across almost 1000 registrable domains, including some of the world's most frequently visited websites, processing trillions of requests from billions of web users daily. To our knowledge, Security Signals is currently the largest implementation of security measurability on the web.

Our paper focuses on the following novel contributions to the area of security measurability:

- **System architecture:** We present a generic, extensible approach for collecting web application security metrics based on real user traffic collected at the reverse proxy layer, accompanied by a number of lessons learned from implementing and deploying this system in production.
- **Data collection:** We provide a list of security metrics relevant for determining the security posture of a web service, and the data sources which make it possible to collect this data at scale. We introduce the concept of *synthetic signals*, which allow surfacing custom security-relevant information at the HTTP protocol level.
- **Security applications**: We share several case studies of successful initiatives that led to substantial web security improvements across our application ecosystem. These efforts significantly reduced the risks associated with common web security vulnerabilities in our services and provide a framework for uplifting security at scale.
- **Visualizing and extending collected data**: A common concern with collecting security metrics is the ability to make them actionable and spur concrete, impactful security improvements. We demonstrate how collected data can be presented to both expert and non-expert users to allow quick assessments of application security, as well as automated analysis of security posture for both individual services, and across entire ecosystems.

Our aim is to provide academics and security practitioners with a blueprint for practical, comprehensive measurement of security quality in web services, and initiate a discussion about extensions and additional use cases for this infrastructure among the security community.

II. BACKGROUND

A. Web service measurability challenges

The typical architecture of web services—defined here as HTTP-based services with which the user generally interacts using their web browser—poses unique challenges which make it difficult to comprehensively assess their security posture using established approaches such as static analysis(6), dynamic analysis(7), and formal methods(8). Web applications typically interact with a large variety of backend components, for example by transmitting data and invoking capabilities with Remote Procedure Calls(9), calling software libraries written in different languages, and using databases or other data storage systems for persistence; they are also often controlled through a set of values known only at code execution time (e.g. command-line parameters, runtime experiments). The highly distributed and dynamic nature of such services makes it difficult to collect information about source code powering these services that could facilitate whole program analyses of internal program states.

This is exacerbated by the lack of standardized approaches for receiving external input—web services typically read and make decisions (security-related or otherwise) based on a variety of user-controlled data at any point during the processing of a given HTTP request. They can collect this information from various sources: the HTTP request method, path or URL parameters, as well as in-memory session data based on the user's cookie or databases which store information about the user. This absence of clear interfaces for exchanging information between the user and the web application makes reasoning about the internal state, and thus about the security properties, of a web application difficult in practice.

Web services also tend to evolve rapidly, frequently implementing many unrelated changes per day(10). Effective measurement approaches must thus be fully automated—without requiring human-in-the-loop assistance—and responsive to any changes in the underlying codebases.

B. Vulnerabilities in web services

A nearly universal aspect of web service behavior is that they accept requests from users in the form of HTTP requests and return HTTP responses with relevant data, possibly after initiating logic that triggers additional server-side behaviors or modifies stored data. In practice, the attack surface of a web service corresponds to the set of actions that can be invoked either through HTTP requests processed by the target service or by browser-mediated interactions processed by client-side code. Vulnerabilities can generally be triggered either through sending requests on behalf of a victim user, authenticated with their cookies (for example if the user visits an attacker's web page), or directly sent by the attacker to the target web server.

A key insight of our measurement approach is that information provided at the HTTP request/response level is often sufficient to gain an understanding of both potential attacks that might affect a given web application endpoint and the defenses or mitigations applied by the application that prevent it from being vulnerable.

For example, a web application's attack surface for common classes of vulnerabilities exploitable against logged-in users can often be closely approximated by looking at request/response pairs: exploitable cross-site scripting and clickjacking vulnerabilities will likely be limited to renderable MIME types (HTML or XML `Content-Type` headers); cross-site request forgery is mostly confined to endpoints that process HTTP POST requests; cross-site script inclusion(11) can only affect responses that are returned with a JavaScript MIME type such as `text/javascript`, etc.

Likewise, defensive mechanisms that protect an application from common web flaws are usually delivered as HTTP response headers as a signal to the user's browser to enforce a given security restriction. The presence of an `X-Frame-Options` response header ensures that a resource will be safe from clickjacking attacks; a `Content-Security-Policy` with a strong policy will ensure any injection flaws are unlikely to result in cross-site scripting; a restrictive `Cross-Origin-Resource-Policy` value will protect a resource from cross-site script inclusion(11) and many cross-site leaks.

Thus, by collecting data from HTTP requests and responses to a production instance of a web application, it's possible to understand the susceptibility of the application to common classes of web vulnerabilities and enabled protections. Importantly, this approach can be extended to other classes of vulnerabilities and defenses.

C. *Enabling effective measurability*

We have found that to be successful in practice a measurability approach needs to be:

1. **Technology-agnostic**: Data collection needs to be easy to enable for a wide range of applications built using diverse programming languages and frameworks, ideally without requiring either developers or system administrator teams to make service-specific changes. It should be possible to enable it even in the absence of code or direct access to the measured systems.
2. **Comprehensive**: It's necessary to collect information about all the web-exposed endpoints of an application to understand the individual security properties of each endpoint. More so, the gathered data should ideally provide complete information about the security posture of the application, covering the common classes of flaws to which the application might be susceptible.

We designed our approach to focus on these two goals.

D. *Collecting HTTP request/response data*

A key practical observation is that while the set of programming languages and application stacks used to build web applications is particularly diverse, there is a smaller number of commonly used intermediary systems such as HTTP servers and reverse proxies which forward traffic between the user and a web service. Popular tools in this category include nginx, HAProxy, Caddy, Traefik as well as Apache configured in reverse proxy mode.

Organizations typically use a reverse proxy system to route user connections to a large number of web services they provide with the goal of providing load balancing, terminating HTTPS traffic, as well as providing centralized logging and related capabilities.

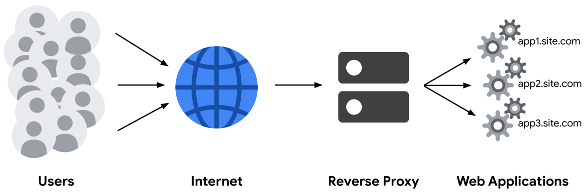


Fig. 1. Traffic to web services typically flows through a reverse proxy.

This design pattern creates an opportunity to implement data collection functionality in the reverse proxy and collect and store data for a number of web services whose traffic flows through the system. This has multiple scalability benefits, including automatically collecting data for any newly created services, and creating a single place where any custom logging capabilities can be added.

For organizations with a more varied network architecture, where web service traffic isn't routed through a few "choke points", an alternative design is to use the request logging capability, universally available in HTTP servers, to store relevant information from both requests and responses. Logs from individual systems can then be aggregated to provide a unified view of their security posture available for analysis.

E. *Exposing custom security properties with synthetic signals*

While HTTP response headers provide data about common web security defenses enabled by an application, they necessarily cover only a small subset of the information necessary to evaluate the security posture of a web service. There are a number of important security properties of a web application that by default aren't exposed in HTTP headers. For example, a security engineer might be interested in whether an access control check was made during the handling of a given request, or whether an HTML response was constructed using a safe, autoescaping HTML templating system.

A critical aspect of our security measurability approach is allowing applications, frameworks and middleware components to expose these security properties at runtime, through *custom HTTP headers* carrying information about the presence or absence of a given security property. These values can be stored by the reverse proxy system and removed before forwarding the response to prevent internal security-relevant information from being exposed to end users. A detailed review of custom security properties relevant for web applications is present in the *Synthetic Signals* section.

III. ARCHITECTURE

The real-world production implementation of our web service security measurability approach is known within Google as **Security Signals**. At its core, the system is a Flume(12) distributed map-reduce data processing pipeline that collates various sources of information to produce insights into the security properties of web traffic flowing to and from our organization's web services. Because of the massive scale and sensitivity of collected data, the pipeline focuses on reducing the cardinality of input data and removing privacy-sensitive information, producing a high-quality output that enables engineers to execute fast queries on the collected data.

A. *Input Data Sources*

External HTTP Traffic Logs

Our infrastructure employs a standardized reverse proxy system(13) to handle nearly all incoming HTTP traffic. Due to large traffic volume, the reverse proxy capability is distributed, performed across many physical machines in multiple geographically dispersed data centers; however, because these systems are powered by the same code, they can be considered as a single system. These reverse proxies produce and manage traffic logs: structured data sources that capture information

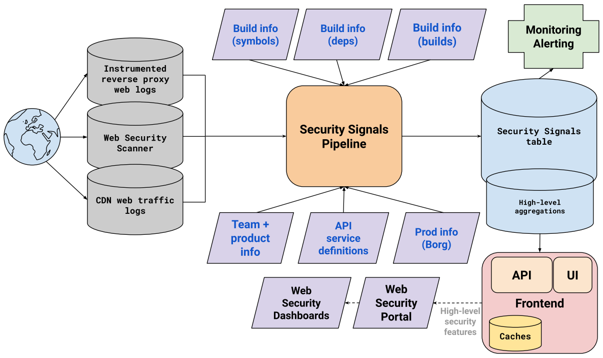


Fig. 2. Input and output data sources of Security Signals.

about HTTP headers for requests and responses that flow through them.

Due to the high volume of requests processed by these servers, logs can capture only a small random sample of traffic—in most cases, 1% of a service's requests. Security Signals is aware of sampling rates and can dynamically increase them on a per-service basis to ensure that low-traffic services are represented in its output. This is important in order to provide visibility across all externally facing web applications while collecting accurate information about traffic volume.

Employee Traffic Logs

Many organizations have large internal user populations which interact with their systems, both those visible to external users and internal-facing. Storing information about these users' interactions with the organization's web services has several benefits: it provides information about applications that aren't exposed externally to end users, including internal administrative systems and not-yet-launched services, and it allows for setting a higher sampling rate (in our implementation we sample 10% of traffic from employees).

Security Scanner Logs

The Security Signals system also consumes traffic logs generated by Google, a custom security scanner focusing on automatically probing for common web vulnerabilities. Ingesting these logs alongside with the real-world traffic logs allows us to gain coverage on yet-to-be launched products, allowing the detection of security issues before they affect a production system.

Furthermore, since scanner logs are guaranteed to not contain any user data, we can rely on them to inspect more precise metadata about requests/responses (e.g. inspecting the full HTTP response body), thereby enabling a wider range of security investigations and remediations.

A combination of these data sources allows for generating a comprehensive inventory of web applications and their individual endpoints, which serves as the foundation for all other Security Signals capabilities.

B. Collected Information

In addition to having a comprehensive inventory of an organization's web services, to provide robust security measurability capabilities the system must collect security-relevant data that allows building a thorough understanding of the overall security posture of each application, and the ecosystem as a whole. To achieve this, we have designed Security Signals to collect data from a variety of sources.

1) Basic HTTP Request & Response Data: The raw information available by default in HTTP traffic provides a large amount of security-relevant data. Security Signals store information about both the request and response, including the HTTP method, destination host, redacted path, status code, returned MIME type, and related values.

Even information seemingly not directly related to security can become important from a security perspective. Because of this we collect a variety of other information, including the values of `Referer`, `Cache-Control` and other headers, as outlined in *Appendix A*. Additionally, we store metadata about the request, including the timestamp and sampling frequency.

2) HTTP security headers: Web platform security mechanisms are generally configured through HTTP response headers; similarly, clients often provide security-related information in request headers. Security Signals aims to collect all available platform security headers from the request and response, corresponding to the security features listed below.

TABLE I
HTTP SECURITY HEADERS AND THEIR ASSOCIATED SECURITY CONTROLS

HTTP Security Header	**Security Control**
Content-Security-Policy	Strict Content Security Policy(14): The presence of a strict CSP policy to mitigate XSS vulnerabilities.
Content-Security-Policy	Allowlist-based Content Security Policy: Restricting script loading to trusted locations to prevent loading of third-party scripts and mitigate supply chain attacks.
Content-Security-Policy	Trusted Types(15): The adoption of Trusted Types for DOM-based XSS protection.
Cross-Origin-Opener-Policy	Preventing cross-window cross-site leaks and related attacks.
X-Frame-Options	Restricting framing to protect against clickjacking attacks.
Strict-Transport-Security	Enforcing the use of HTTPS for an origin or domain.
Sec-Fetch-Dest, Sec-Fetch-Mode, Sec-Fetch-Site	Fetch Metadata headers(16) for assessing if resource and framing isolation policies were applied to prevent cross-site leaks.
Cross-Origin-Resource-Policy	Protecting against certain requests from other origins (such as those issued with elements like `<script>` and `<img>`), to mitigate speculative side-channel attacks, like Spectre(17), as well as Cross-Site Script Inclusion(11) attacks.
Cross-Origin-Embedder-Policy	Restricting embedding cross-origin resources into the document, ensuring that all resources loaded by a given document have explicitly opted into being embedded.
X-Content-Type-Options	Prevents MIME sniffing attacks.

Similarly, we collect a number of auxiliary sources of security-relevant information, including the `Origin` and various headers related to Cross-Origin Resource Sharing and cookie security attributes.

3) Synthetic Signals: While traffic logs do provide significant utility alone, a core capability of the Security Signals approach is the collection of *synthetic signals* that contain additional metadata that is not normally included in traffic logs. At an architectural level, this is done by instrumenting web frameworks to emit this metadata in an internal-only `X-Google-Security-Signals` HTTP response header. This header is then collected and included in traffic logs to be consumed by the Security Signals pipeline, while removing it before the response is served to external users.

Any information that is known to the server at the moment of processing a given HTTP request can be exposed as a synthetic signal. Some synthetic signals are *request-scoped*, allowing the surfacing of custom security-relevant information, such as whether a check of a CSRF token was performed during the handling of the request, or whether an HTML response was constructed using a safe HTML templating system. Other signals may represent *service-level* security properties such as the build version of the web service binary, the programming language and application framework powering the service (which can be a reliable indicator of the overall security posture of a web application), or information about the specific server-side code responsible for creating the response. See tables II and III.

Our web services are built on a variety of different server-side web frameworks that we have individually instrumented to emit these synthetic signals to provide a more complete view of their security properties.

TABLE II
SYNTHETIC SECURITY SIGNALS COVERING CUSTOM SERVER AND CLIENT SIDE SECURITY CONTROLS

Synthetic Signal	**Description**
RESPONSE_TYPE	Exposes the use of type-safe responses and autoescaping HTML templating systems for XSS prevention.
TEMPLATE	The server-side templating system that generates HTMLoutput .
SEC_FETCH	The presence of server-side isolation policies(16) to assess if isolation policies were applied to prevent cross-site attacks.
CSRF	The presence of Cross-Site Request Forgery(18)(19) protections to verify if an CSRF check was carried out by the backend on state changing requests.
PROTOTYPE_POLLUTION	The presence of prototype pollution protections to determine if front-end code makes JavaScript prototypes immutable.

4) Auxiliary Data: Certain kinds of security-relevant information might not be exposed directly in HTTP request/response pairs, or easy to provide as a synthetic signal. For this reason, the Security Signals system also queries several internal databases such as build systems and corporate IT systems, collecting information about organizational structure. This allows joining information about the production environment, ownership information (team, project name, and the owner's contact information), source-code information (allowing to identify the specific function that serves a given endpoint), providing additional information about a given service.

TABLE III
SYNTHETIC SECURITY SIGNALS PROVIDING ADDITIONAL CONTEXT ON THE SERVING ENVIRONMENT

Synthetic Signal	**Description**
FRAMEWORK	The serving web framework. This allows easy differentiation between hardened and safe-by-default frameworks vs. the use of legacy frameworks.
ACTION	Method-level pointer to the code generating the web response, together with some framework-specific metadata such as experiment configuration.
BUILD	Information about the application's build environment.

This context is also crucial for streamlining remediation efforts, as it allows for automatic identification of relevant code and ownership information, allowing automated bug filing.

5) Risk Signals: Not all services are equally sensitive from a security perspective. Security Signals incorporates various factors that enable assessing and prioritizing risk:

- **Sensitivity of the hosting domain**: Leveraging a categorization based on Domain Tiers(20), Security Signals determines the inherent risk associated with each web origin based on factors such as the sensitivity of data it processes and potential impact of compromise. This enables focusing security efforts on securing the most critical applications.
- **Traffic volume**: While not always a direct indicator of risk, high traffic volume can be used to prioritize within risk tiers and identify popular applications that require extra security attention.
- **External exposure**: Traffic exposed to external users naturally carries a larger risk of being exposed to attacks. Security Signals identifies such traffic to prioritize mitigation of external threats while also considering critical internal systems for insider risk management.

C. Cardinality Reduction

An important idea behind Security Signals is that it is possible to take a high-cardinality input that is impractical to query (e.g. traffic logs with hundreds of billions of distinct entries), and transform it into a lower-cardinality output designed to be easily queryable. To reduce the cardinality of the input, it is necessary to purposefully drop information from the input traffic logs, while still maintaining sufficient granularity to make the data useful. This process also helps ensure that the output data is fully anonymous by removing any personalized data from the input. This strategy is applied to all data in Security Signals, including for:

Path Redaction: Individual instances of URLs often contain superfluous information that negatively affects the cardinality of input data and which could contain personal identifiers. Examples include authentication tokens, timestamps or parameters containing user input. Because the security properties of server-side code serving such URLs are invariant, Security Signals employs a number of techniques that reduces input

URL cardinality with no loss of generality. Since URL query parameters are always ignored by our system, we call this process *path redaction*. The end result converts individual URL paths into *path patterns*.

At a high level, the path redaction algorithm is as follows: if available, leverage path routing information provided by either frameworks or reverse proxies. This data may be present in the `ACTION` synthetic signal or per-service infrastructure configurations. This allows us to match and replace variable parts, for instance `/v1/search/query+string` with `/v1/search/$query`. Since this technique uses source-of-truth inputs, it successfully redacts over 90% of all paths. If this information is not available, we apply filtering rules based on a manually curated set of well-known high-entropy paths.

Finally and as a fallback, we execute a stateless random forest machine learning model applied on individual path tokens, trained on real-world data. This model uses entropy- and dictionary-based techniques to infer redactions from a corpus of real-world traffic, and consists of 11 decision trees with a maximum depth of 5.

User-Agent Parsing: We parse user agent information and keep only coarse-grained information, such as the browser name and major version, obtained from User-Agent Client Hints(21), where available, and by parsing the `User-Agent` request header otherwise. Storing only the browser major version, together with its name, ensures that the cardinality of the output table remains limited while still preserving the utility of being able to query based on browser version.

D. Output Database

After cardinality reduction and the inclusion of synthetic signals and auxiliary data, the resulting set of outputs is materialized to a dated database table that can be queried in SQL for a specific period of time. This process occurs on a daily basis, as most of the capabilities and use cases described in this paper have no need for real-time information. Security Signals output data is retained for a period of 30 days, which enables time series analysis, regression detection, and other monitoring tasks.

Finally, adjacent jobs store aggregated views on top of Security Signals that provide high-level statistics. For example, coverage information of important web security features or the total number of hostnames and services. This information is kept in secondary tables for long-term retention and visualization in internal dashboards.

These condensed views aggregate data in a way that provides actionable insights to users with more or less expertise with web security. Section V describes what types of aggregations exist and example use cases for different types of users.

IV. APPLICATIONS

Data collection alone is not sufficient for implementing a robust security measurability program. A crucial question is whether the information can be effectively used in practice to improve security outcomes in the measured ecosystem. In this section, we outline how Security Signals supports Google's strategy for securing web services at scale(22), and the capabilities it offers to security engineering teams and decision-makers.

A. Adoption of Web Security Features

Organizations are frequently faced with a large amount of legacy code and systems built using approaches without modern security safeguards. This creates an ongoing need to improve the security state of existing web services to bring them in line with security best practices, such as using safer application components or adopting web-platform level defensive mechanisms.

Making these kinds of large-scale security improvements can be daunting, since it involves:

- Identifying services or specific endpoints where a protection is missing. This has traditionally been challenging in large heterogeneous environments.
- Initiating service-specific work to enable a new security feature. This work can involve making far-reaching changes to a service, which introduces the risk of breaking existing functionality.
- Tracking deployment progress across hundreds or thousands of services. This requires measuring the status of a complex rollout, and the ability to prioritize deployments for critical services to maximize impact.

Measurement of security deployments

From a project management perspective, being able to accurately and continuously measure the progress of large-scale security deployments is critical. Security Signals makes it easy to precisely measure deployment progress based on flexible criteria; for example, measuring what percentage of services enable Content Security Policy, broken down by framework and application sensitivity.

Prioritization of security rollouts

It is critical to be able to prioritize security rollouts to maximize risk reduction. Since Security Signals incorporates information about the sensitivity of web origins through the Domain Tiers(20) classification, it becomes easy to assess if a given service's sensitivity makes it a good candidate for the adoption of a given security feature. Similarly, it is possible to join Security Signals data with information from a bug bounty program to identify whether a given service has been historically prone to vulnerabilities and would thus benefit from enabling additional defenses.

1) Example: Deploying Trusted Types: To demonstrate how measurement and prioritization can enable successful security rollouts, we provide a summary of how Security Signals supported the deployment of Trusted Types(15) across our ecosystem. Trusted Types is an important client-side security feature that aims to comprehensively prevent DOM XSS vulnerabilities(23) by relying on type information to ensure only safely constructed values can reach dangerous DOM APIs. It is enabled by setting a `Content-Security-Policy` HTTP header with a `require-trusted-types-for 'script'` directive. Our efforts to roll out Trusted Types consisted of:

1. **Targeted service-specific rollouts**: Combined with the domain sensitivity classification, Security Signals made it possible to scalably prioritize work on highly sensitive services. For example, we know that `Google` hosts our organization's login form and sets authentication cookies. Under the same-origin policy(24)(25), every service on that domain is sensitive, so we prioritized rollouts for that origin.
2. **Large scale cross-ecosystem rollouts:** Security Signals also enabled large-scale changes to centrally deploy Trusted Types across our ecosystem of existing services(22). We used Security Signals to approach the rollout in batched rollouts for groups of similar services.

Throughout this process, we were able to measure our rollout progress. In the past 2 years, we have deployed Trusted Types to over 600 distinct services; Security Signals made it possible to accurately track the status of this multi-year project and monitor the resulting security improvements in security critical applications (see Figure 3).

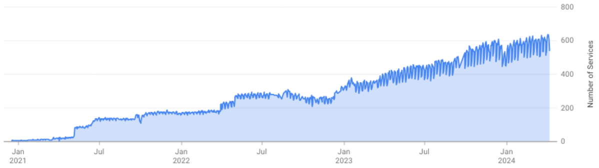


Fig. 3. Web services protected by Trusted Types over time.

We have completed similar rollouts for numerous web-platform security features(22) and to remediate a variety of unsafe patterns across our ecosystem.

B. Monitoring and Regression Detection

In environments where web services evolve quickly, there exists a risk of modifying existing functionality or implementing new features in a way that undermines the security posture of the service. This is particularly common in developer ecosystems that don't follow secure-by-design principles(1) and thus don't robustly prevent developers from writing unsafe code. While some organizations rely on code reviews or periodic penetration tests performed by security experts to identify any newly introduced unsafe patterns or vulnerabilities, these approaches are often costly and do not scale to large application codebases.

Security Signals monitoring is based on three components:

Security Invariant Monitoring: Security Signals continuously queries its own database, searching for violations of predefined security invariants representing expected security behaviors and configurations. For example, the security team may require that all HTML endpoints in a service include the `X-Frame-Options` header to prevent clickjacking vulnerabilities, or that all HTML responses are generated with the use of a safe autoescaping HTML templating system. Security Signals can automatically detect instances where the desired property isn't satisfied and trigger remediation actions.

Alerting: When anomalies or regressions are detected the system can trigger alerts to a security engineering team and, in some cases, directly to the responsible product teams, enabling swift investigation and remediation.

Automated Bug Filing: Leveraging ownership information within Security Signals, bug reports are automatically routed and assigned to appropriate service owners, streamlining the resolution process.

We have enabled this monitoring for a number of security properties, including the absence of defensive features, misconfigurations of security policy headers (e.g. invalid or unsafe Content Security Policy values), and the absence of a variety of application-specific security checks exposed via synthetic signals. This approach enables identifying and addressing potential issues quickly, contributing to a more resilient ecosystem security posture, while requiring minimal involvement from security engineering teams.

C. Targeted Security Research & Remediations

Security remediations are engineering efforts aimed at mitigating systemic sources of vulnerabilities. Remediations start with an observation about *potential* security risk, including traditional classes of security issues and those that emerge from the use of unsafe patterns specific to the programming language or framework used by the application, or as a result of using application-specific dangerous constructs.

Often, remediations are spurred by security research that determines whether a class of security issues exists, what its practical impact is, and how widespread it is likely to be. Once the risk is established, security engineers design mitigations that can be enforced at scale in ways that do not negatively affect service availability. Security Signals capabilities enable a centralized, relatively small team of engineers to execute targeted research and remediations without in-depth knowledge of the internals of specific services.

Importantly, Security Signals provides visibility into *actual* runtime behaviors of applications. In contrast to techniques such as static analysis or code reviews, which often give insights into *potential* behaviors, this approach surfaces only instances that have been demonstrated to exhibit a given behavior, reducing the number of false positive findings. This makes it possible not only to reliably identify and prioritize web endpoints that are likely to be vulnerable, but also to determine how new mitigations may affect them and what actions are needed to add protections safely.

The following sections describe real-world examples of security research and remediations enabled by Security Signals:

CSRF Remediation: Cross-Site Request Forgery is a class of web vulnerabilities that allows attackers to force an authenticated user's browser to invoke a state-changing action on behalf of the user(18)(19). To prevent this issue, web services often use an approach based on requiring the presence of a signed per-user token to verify the request originated from within the application, thereby removing reliance on ambient authority. However, even in services which use CSRF tokens, a failure to verify that a valid token is present when processing a state-changing request will result in a CSRF vulnerability.

Security Signals allowed us to scalably identify CSRF vulnerabilities by finding endpoints without sanctioned implementations of this defense and then determining which endpoints implement state-changing functionality.

Frameworks were instrumented with a synthetic signal indicating which requests were protected by the standard CSRF protection. This helps identify not only endpoints that are not protected against CSRF, but also those that implement custom CSRF protection logic, which is more likely to be vulnerable. Finally, we approximated which endpoints have state-changing features by using a mix of heuristics, including HTTP methods, changes in response sizes, content types, and other request and response properties.

Iterating over this process made it possible to identify endpoints vulnerable to this class of issues across the entire ecosystem of web applications, including many disparate frameworks and programming languages. This concrete list of potentially vulnerable services was then an input into a targeted remediation to adopt standard CSRF protections in applications that hadn't yet done so, reducing the risk of CSRF issues arising in them in the future.

CORS Remediation: Cross-Origin Resource Sharing (CORS)(26) is a mechanism for sharing information across origins, augmenting the same-origin policy. Because CORS allows sharing response data with arbitrary origins, web endpoints with misconfigured CORS headers may unintentionally expose sensitive information. Such misconfigurations are common due to the way CORS forces developers to check whether an origin is trusted; for instance, developers may allowlist all requesting domains ending with `example.com` (note the missing leading dot), thereby allowing requests from `evil-example.com`.

Security Signals made it possible to identify CORS-enabled endpoints, including those that accept requests from third party or untrusted origins. By combining this information with service metadata, we were able to test various CORS implementations to identify vulnerable code locations; in addition to fixing discovered vulnerabilities we developed a centrally supported secure-by-design CORS implementation that mitigates these misconfigurations.

Cached authenticated content remediation: Common misconfigurations of HTTP caching headers may result in sensitive content being unintentionally cached by proxies or edge servers responding to user requests. For example, a web service could accidentally enable the caching of one user's authenticated content and serve it to another user. Using Security Signals' capability to observe fine-grained authentication behavior, we were able to identify endpoints that are eligible for caching *and* that branch on authenticated information. This represents the set of endpoints that may be vulnerable to cache-based information leaks.

Following our secure-by-design approach, we implemented custom caching logic to prevent such cases from happening in the long term, without disrupting use cases that rely on this behavior and that are known to be safe.

D. Measuring Runtime Dependencies & Trust Relationships

Typical modern web applications orchestrate multiple backend services and infrastructure to process HTTP requests, fetch and collate the data necessary to implement complex user-facing functionality. This architecture favors flexibility, but poses a major challenge in measuring web security effectively, since runtime relationships between services and supporting infrastructure are seldom available to static analysis and are often the root cause of high-impact security issues.

One of Security Signals' core capabilities is the ability to highlight these relationships in a framework-agnostic way, effectively surfacing dependencies that would otherwise be impractical to understand at the service or framework levels. This capability is built by cross-referencing synthetic signals with request and response metadata.

Not only does this allow for a deeper understanding of how backend services and infrastructure interact with each other, but it can also highlight web traffic where infrastructure plays an important role for security, for example in marshaling or normalizing requests. The following list gives examples of capabilities enabled by these insights:

Highlighting cross-framework and cross-tier trust relationships. The ability to identify critical services that establish trust relationships with lower-sensitivity services or with a weaker security posture unlocks the ability to pinpoint web endpoints and workflows that are more likely to be vulnerable. These services are ideal candidates for security hardening efforts.

Understanding dependencies between business logic and infrastructure. Some synthetic signals may be emitted when requests are transformed or processed by infrastructure elements outside a service's business logic. This includes selecting backend services to route requests to, marshaling requests across several protocols, or making modifications to HTTP requests. Identifying where such transformations are applied enables security research that can discover vulnerabilities that surface from infrastructure nodes, including caching servers, load balancers, fetch systems, reverse proxies and others.

E. Additional Capabilities

The area of web application security is quickly evolving: new attacks and defenses are introduced on a regular basis, requiring security engineering teams to continuously evaluate their ecosystems' security posture and respond to new threats. A core goal of the Security Signals approach is flexibility: it can be adjusted to collect new types of data and integrate with additional sources of security information, thus acquiring new useful capabilities.

1) Enhancing JavaScript Security: We have extended Security Signals to the realm of JavaScript security through a dedicated *JavaScript Signal* pipeline. This pipeline maps JavaScript resources loaded by our web services to their corresponding source files. By integrating with our organization's standardized build tooling (Bazel(27) and Closure(28)), JavaScript Signal ensures that we can assess that all executing scripts adhere to strict security standards and are free of vulnerable patterns. It also allows us to assess code provenance properties to further mitigate supply chain risks; for instance, we can readily identify whether a vulnerable version of a third-party library is present within a production web application and pinpoint its exact code location, to ensure the library is promptly updated.

2) Improving Security Scanning Coverage: Our organization employs a custom web security scanner to automatically detect web application vulnerabilities. While effective at discovering web security issues, its impact can be reduced by limited coverage: scanners typically initiate crawls based on a few seed URLs and often cannot discover all parts of the scanned web service.

Security Signals addresses this limitation by providing the scanner with a targeted list of URLs derived from real-world traffic patterns. By leveraging Security Signals' inventory, ability to map URLs to backend actions and understand query parameter semantics, we generate a precise and deduplicated set of scan targets. This approach significantly enhanced our scan coverage, especially for internal-only services (resulting in a threefold scan coverage increase), ensuring that critical web applications and endpoints within applications are not missed during security scanning.

3) Non-security Use Cases: Although originally designed for security purposes, the Security Signals system is currently used by over 50 teams across Google for many purposes that are not directly security-related. This demonstrates that while our focus has been on making web security measurable, the resulting measurability has benefits that extend also to non-security domains.

a) Product-level usage questions: Data collected by Security Signals makes it possible to answer various kinds of product-level questions without having to implement custom application-specific data collection. For example, product teams can query for specific patterns, such as the use of Service Workers, or the use of resources based on their MIME type. This can allow implementing various optimizations; for example, in the case of commonly loaded images, a developer might decide to reduce their size or improve their caching properties to save on bandwidth costs and improve performance for users.

b) Measuring compatibility with third-party cookie restrictions: As browsers introduce restrictions on third-party cookies, existing services that rely on them may be affected. This makes it important to find ways to scalably identify application patterns incompatible with third-party cookie deprecation. Since Security Signals contains data about the source and destination of a given request (by collecting the `Referer`, `Origin`, and `Host` HTTP request headers), and whether a request is authenticated (based on the `Set-Cookie` header and synthetic signals), it was possible to leverage Security Signals to identify services that required third-party cookies to function. We were able to use this to identify internal services that require third-party cookies and deploy a set of well-scoped exceptions to mitigate these breakages for corporate users; without Security Signals, this would have required a time-consuming manual effort.

c) Surfacing AI/ML Properties: By joining the set of web-level information with a separate graph of Remote Procedure Calls, Security Signals can identify web endpoints that depend on generative AI models. This capability enables use cases that aim to understand which models have access to user data and where they are exposed to users. These analyses holistically assess the sensitivity of AI-enabled applications, going beyond the traditional model based on web origins.

V. MEASURABLE WEB SECURITY

We found that a key practical aspect of addressing the challenge of insufficient measurability is to transform collected data into meaningful security quality metrics. To achieve this, the data must be aggregated, enriched with expert insights, and tailored to the needs of different audiences.

A. Making Security Risk Measurable and Actionable

As part of Security Signals we built tailored interfaces catering to different users, each presenting security data at varying levels of granularity depending on the users' needs:

- **Security engineers**: Access raw data and visualizations to proactively detect vulnerabilities, conduct in-depth security investigations and run large scale remediations.
- **Product teams**: Utilize aggregated data and actionable insights to assess their product's security posture, self-evaluate against best practices, and adopt secure-by-design technologies.
- **Leadership**: Review risk metrics and strategic recommendations to inform decision-making, prioritize security initiatives, and steer their organizations towards the adoption of secure-by-design architectures for new projects.

1) Visualizations for Security Engineers: We provided security engineers with a powerful visualization tool to explore and analyze web application security posture. Application endpoints are presented as interactive "bubbles" organized by code package and color-coded to reflect their security status (see Figure 4 and 5). This provides security engineers with a range of capabilities useful in their work:

- **Identifying security gaps**: Visualize the security posture of each application endpoint, including details about enabled security features and potential vulnerabilities.
- **Initiating remediations**: Use advanced filtering capabilities to isolate endpoints with specific security weaknesses, enabling targeted remediation efforts.
- **Filing bugs for product teams**: Directly access relevant code locations and file pre-populated bug reports automatically assigned to the appropriate product teams, accelerating issue resolution.

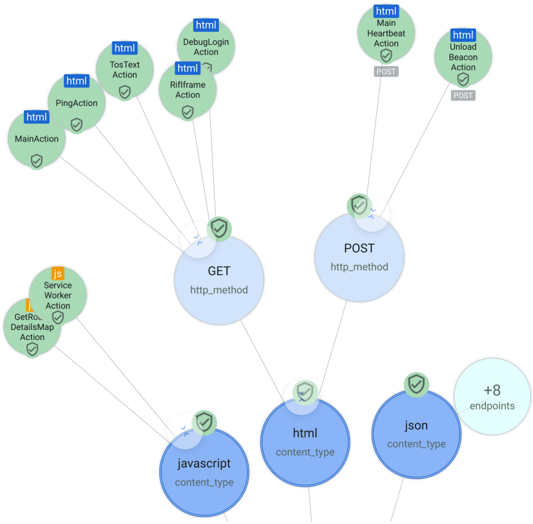


Fig. 4. Visualizing web endpoints by content type and method.

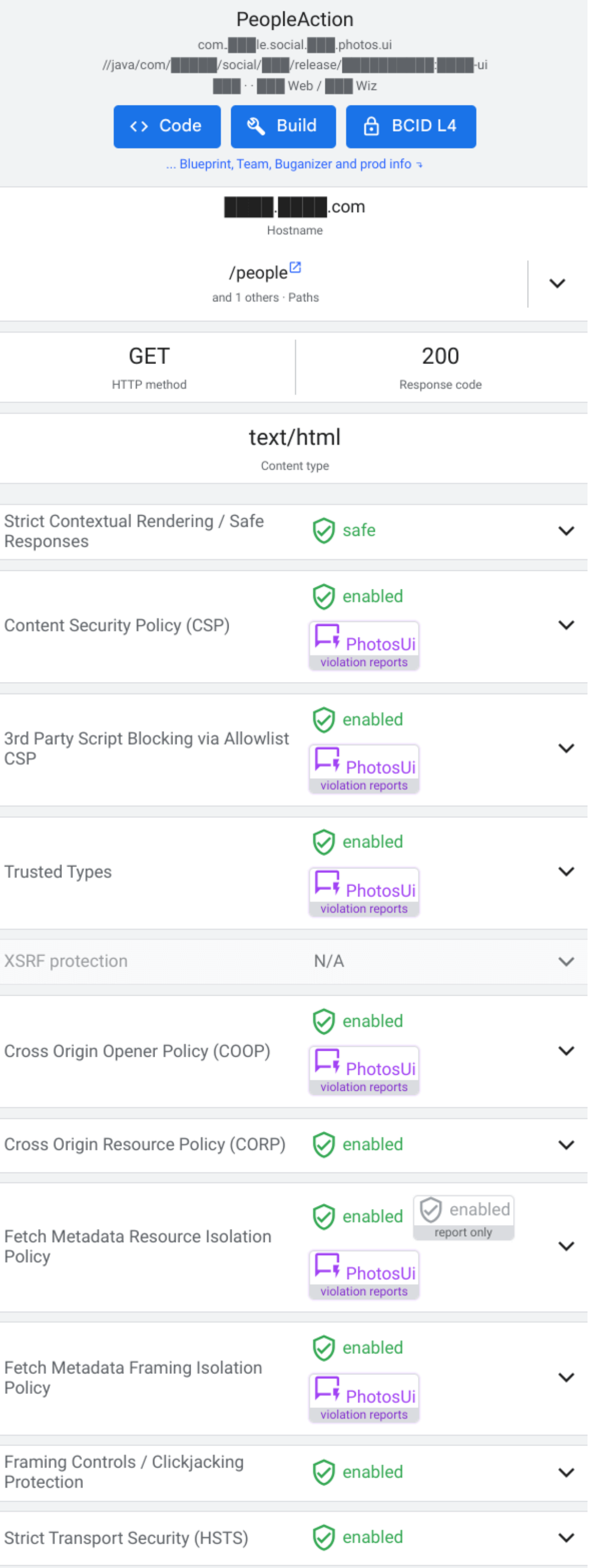
2) *Security Scorecards for Product Teams:* An important goal of Google's web security team is to scale security through the use of secure defaults and large-scale improvements. To this end, we developed an application to empower product teams to actively participate in this process and gain a clear understanding of their own service's security posture and discover how adopting recommended frameworks can streamline their web security efforts.

This application provides insights tailored to each team's application framework. By highlighting areas for improvement and offering framework-specific recommendations, it makes it easier for product teams to implement security best practices and protect their users.

Developers without security expertise are provided with information in an easily digestible format, categorized by project, hostname, team, or product area. Teams can readily identify areas needing attention, track their remediation progress, and monitor for regressions, ensuring continuous improvement in their web application security (see Figure 6).

3) *Dashboards for executives:* The data collected through Security Signals provides high-level visibility and strategic insights to executives to allow risk-based prioritization and resource allocation decisions. To cater to this use case, we developed dashboards that allow organization leaders to:

- **Assess overall web security posture**: Gain a comprehensive understanding of the organization's web application security posture through surfacing aggregated metrics, historical trends, and risk scores. This allows making informed, data-driven decisions about resource allocation and policy development.
- **Identify areas of focus**: Pinpoint areas of higher risk or requiring immediate attention, such as the absence of important security controls.
- **Track progress and quantify impact**: Monitor the effectiveness of security initiatives and remediation efforts over time, demonstrating how they reduce security risks.


PeopleAction
com.[redacted]le.social.[redacted].photos.ui
//java/com/[redacted]/social/[redacted]/release/[redacted]:[redacted]-ui
[redacted] · · [redacted] Web / [redacted] Wiz

[redacted].[redacted].com
Hostname

/people
and 1 others · Paths

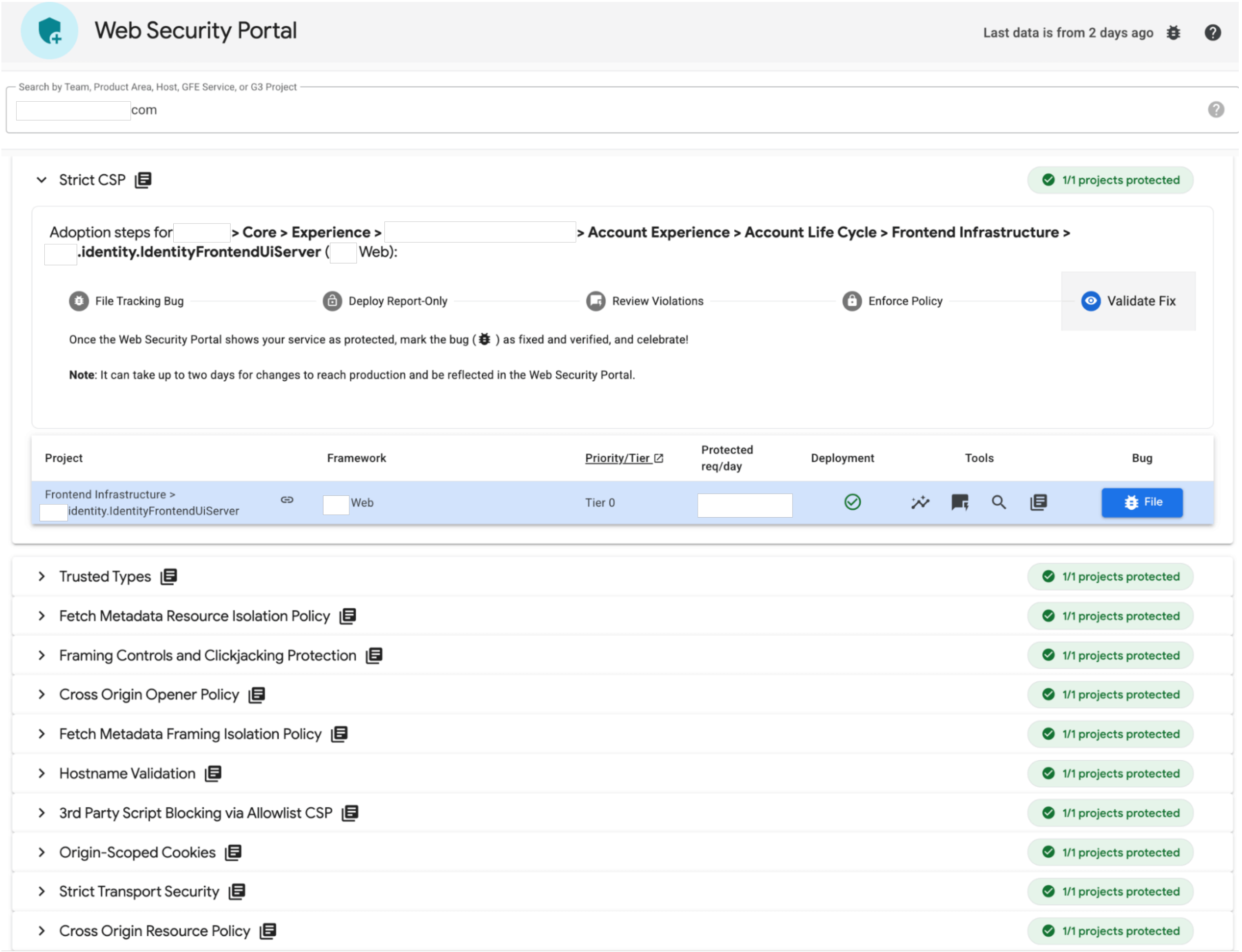
GET HTTP method	200 Response code

text/html
Content type

Strict Contextual Rendering / Safe Responses	safe
Content Security Policy (CSP)	enabled PhotosUi violation reports
3rd Party Script Blocking via Allowlist CSP	enabled PhotosUi violation reports
Trusted Types	enabled PhotosUi violation reports
XSRF protection	N/A
Cross Origin Opener Policy (COOP)	enabled PhotosUi violation reports
Cross Origin Resource Policy (CORP)	enabled
Fetch Metadata Resource Isolation Policy	enabled enabled report only PhotosUi violation reports
Fetch Metadata Framing Isolation Policy	enabled PhotosUi violation reports
Framing Controls / Clickjacking Protection	enabled
Strict Transport Security (HSTS)	enabled

Fig. 5. Visualizing web security features and their status.

These dashboards distill complex security data into simple visualizations, enabling leaders to quickly grasp the key chal-



Search by Team, Product Area, Host, GFE Service, or G3 Project

[redacted]com

Adoption steps for [redacted] **> Core > Experience >** [redacted] **> Account Experience > Account Life Cycle > Frontend Infrastructure >** [redacted]**.identity.IdentityFrontendUiServer (**[redacted] Web):

Once the Web Security Portal shows your service as protected, mark the bug ( ) as fixed and verified, and celebrate!

Note: It can take up to two days for changes to reach production and be reflected in the Web Security Portal.

Project	Framework	Priority/Tier	Protected req/day	Deployment	Tools	Bug
Frontend Infrastructure > [redacted]identity.IdentityFrontendUiServer	[redacted]Web	Tier 0	[redacted]	✓		File

Trusted Types	1/1 projects protected
Fetch Metadata Resource Isolation Policy	1/1 projects protected
Framing Controls and Clickjacking Protection	1/1 projects protected
Cross Origin Opener Policy	1/1 projects protected
Fetch Metadata Framing Isolation Policy	1/1 projects protected
Hostname Validation	1/1 projects protected
3rd Party Script Blocking via Allowlist CSP	1/1 projects protected
Origin-Scoped Cookies	1/1 projects protected
Strict Transport Security	1/1 projects protected
Cross Origin Resource Policy	1/1 projects protected

Fig. 6. High level overview of a web security scorecard for product teams.

lenges and opportunities related to web application security (see Figure 7). This empowers them to champion security initiatives and support adoption of secure-by-design principles.

B. Assuring Security-by-Design

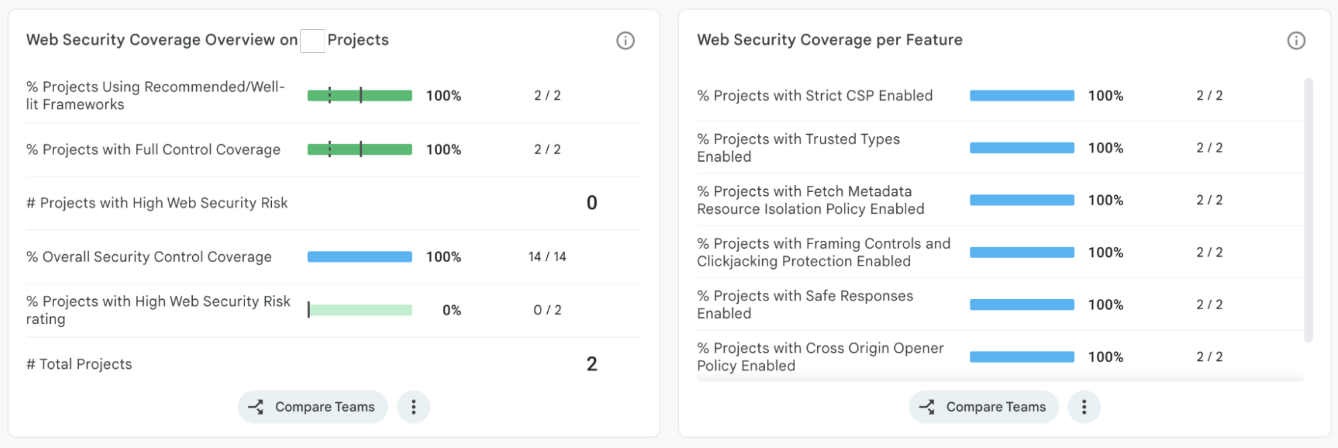
Importantly, Security Signals can also provide higher-level interpretations of the data to demonstrate that certain systems or applications are inherently "safe-by-design"(1) from broad classes of vulnerabilities. This means achieving a level of assurance about the absence of specific security issues that wouldn't be possible through traditional point-in-time audits or by measuring the coverage of individual security controls.

This has significant benefits: if it's possible to gain a high degree of confidence about the absence of a given class of flaws, security engineers and decision makers can make better risk-based prioritization decisions and avoid investing time in efforts less likely to lead to practical security improvements.

1) Example: Preventing Cross-Site Scripting: Cross-site scripting (XSS) has historically been the most common high-risk vulnerability affecting web applications(14). Holistically addressing XSS requires strong separation between code and data throughout the application. The only robust defense is to implement a number of security controls that prevent the use of unsafe server-side and client-side templating systems and code that fails to guarantee this separation. Additionally, these unsafe APIs can be restricted directly in the web browser via opt-in security mechanisms such as strict Content Security Policy(14) or Trusted Types(15).

By measuring the presence of security controls in an application, we can infer that it is safe-by-design against XSS vulnerabilities. This provides a high degree of confidence in the application's resilience to this class of threats. Past and future defect data can then be used to validate the effectiveness of the threat model and ensure the continued adequacy of the implemented security controls. This approach, driven by comprehensive measurement and the enforcement of safe-by-design principles, has allowed us to successfully eliminate XSS vulnerabilities across Google, a popular serving stack used by hundreds of applications.

The use of Security Signals has allowed security teams to



Web Security Coverage Overview on Projects

% Projects Using Recommended/Well-lit Frameworks	100%	2 / 2
% Projects with Full Control Coverage	100%	2 / 2
# Projects with High Web Security Risk		0
% Overall Security Control Coverage	100%	14 / 14
% Projects with High Web Security Risk rating	0%	0 / 2
# Total Projects		2

Web Security Coverage per Feature

% Projects with Strict CSP Enabled	100%	2 / 2
% Projects with Trusted Types Enabled	100%	2 / 2
% Projects with Fetch Metadata Resource Isolation Policy Enabled	100%	2 / 2
% Projects with Framing Controls and Clickjacking Protection Enabled	100%	2 / 2
% Projects with Safe Responses Enabled	100%	2 / 2
% Projects with Cross Origin Opener Policy Enabled	100%	2 / 2

Fig. 7. Aggregated view of web security features for executives.

ensure that all the necessary anti-XSS defenses are comprehensively enabled in Google applications, reducing the need for conducting manual security reviews focusing on identifying this class of flaws and freeing up substantial security engineering resources.

VI. CONCLUSION

In this paper, we presented our experience from the design and real-world use of *Security Signals*—a far-reaching effort to implement security measurability for web services focusing on enabling practical security improvements in a large-scale web services ecosystem.

Collecting security-relevant information from HTTP traffic at the reverse proxy layer, we developed a capable security system which addresses several shortcomings of prior web measurability proposals and introduces several novel ideas. The concept of *synthetic signals*, exposing custom application security properties as HTTP response headers, makes it possible to collect arbitrary security-related information, complementing the limited set of data present by default in HTTP headers. Integrating additional risk information, such as the relative sensitivity of a given web origin or the amount of traffic a service receives, aids security teams in assessing risk and making prioritization decisions. Optimizations such as path redaction to reduce cardinality can ensure that the output database remains limited in size for efficient querying, while also preventing any sensitive information from being unintentionally persisted. Integrations with various organization-specific systems (e.g. bug tracking tools and additional sources of security data) make it possible to extend the capabilities of the system beyond originally envisioned uses.

Security Signals has aided security teams at Google in uplifting the security of a complex ecosystem with over 8000 web services. It facilitates automatic monitoring of security invariants, preventing regressions and providing notifications to product teams. It supports deployments of security improvements, including native web mechanisms and framework-specific enhancements. By uplifting the ecosystem's security posture, it supports the implementation of secure-by-design frameworks and technologies. Security Signals has also been instrumental in security research, enabling teams to flag potentially unsafe patterns for investigation and remediation.

Finally, the visibility into web service security properties provided by Security Signals has found a large number of practical uses among product teams, security engineering teams, and security executives. By exposing data at different levels of detail—from HTTP-level information about specific endpoints to aggregate "security scores" for web services, or groups of services—it has supported security decision-making and prioritization across our ecosystem.

We expect that lessons learned from the use of Security Signals are broadly applicable to practitioners responsible for the security posture of web services, and can spur the development of powerful approaches for web security measurability.

VII. APPENDICES

A. List of collected HTTP headers

Security Signals reads the following request and response HTTP headers:

- `Access-Control-Allow-Credentials`
- `Access-Control-Allow-Headers`
- `Access-Control-Allow-Methods`
- `Access-Control-Allow-Origin`
- `Access-Control-Expose-Headers`
- `Access-Control-Request-Headers`
- `Access-Control-Request-Method`
- `Authorization`
- `Cache-Control`
- `Content-Disposition`
- `Content-Length`
- `Content-Security-Policy`
- `Content-Security-Policy-Report-Only`
- `Content-Type`
- `Cross-Origin-Embedder-Policy`
- `Cross-Origin-Embedder-Policy-Report-Only`
- `Cross-Origin-Opener-Policy`

- `Cross-Origin-Opener-Policy-Report-Only`
- `Cross-Origin-Resource-Policy`
- `Location`
- `Origin`
- `Purpose`
- `Referer`
- `Referrer-Policy`
- `Report-To`
- `Sec-Ch-Ua`
- `Sec-Fetch-Dest`
- `Sec-Fetch-Mode`
- `Sec-Fetch-Site`
- `Sec-Fetch-User`
- `Server`
- `Service-Worker`
- `Set-Cookie`
- `Strict-Transport-Security`
- `User-Agent`
- `Vary`
- `X-Content-Type-Options`
- `X-Frame-Options`

REFERENCES

[1] C. Kern, "Developer ecosystems for software safety: Continuous assurance at scale." *Commun. ACM*, apr 2024, online First. [Online]. Available: https://doi.org/10.1145/3651621

[2] "Securing the software supply chain: Recommended practices for customers," 2022. [Online]. Available: https://www.cisa.gov/sites/default/files/2023-01/ESF_SECURING_THE_SOFTWARE_SUPPLY_CHAIN_CUSTOMER.PDF

[3] "Press release: Future software should be memory safe," 2024. [Online]. Available: https://www.whitehouse.gov/oncd/briefing-room/2024/02/26/press-release-technical-report/

[4] "Federal cybersecurity research and development strategic plan," 2023. [Online]. Available: https://www.whitehouse.gov/wp-content/uploads/2024/01/Federal-Cybersecurity-RD-Strategic-Plan-2023.pdf

[5] N. Whitehouse, L. Williams, M. Finifter, D. Geer, M. Finifter, P. Johnson, H. Lipson, G. McGraw, P. Schneck, A. Shostack *et al.*, "Back to the building blocks: A path toward secure and measurable software," 2011.

[6] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: using static analysis to find bugs in the real world," in *Communications of the ACM*, vol. 53, no. 2. ACM New York, NY, USA, 2010, pp. 66–75.

[7] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Cross site scripting prevention with dynamic data tainting and static analysis," in *NDSS*. The Internet Society, 2007. [Online]. Available: https://auto.tuwien.ac.at/~chris/research/doc/ndss07_xssprevent.pdf

[8] M. Bugliesi, S. Calzavara, and R. Focardi, "Formal methods for web security," *Journal of Logical and Algebraic Methods in Programming*, vol. 87, pp. 110–126, 2 2017. [Online]. Available: https://iris.unive.it/bitstream/10278/3685125/1/jlamp16.pdf

[9] A. D. Birrell and B. J. Nelson, "Implementing remote procedure calls," *ACM Transactions on Computer Systems (TOCS)*, vol. 2, no. 1, pp. 39–59, 2 1984. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/2080.357392

[10] T. W. Beyer, T. Manshreck, and H. Wright, *Software Engineering at Google: Lessons Learned from Programming Over Time*. O'Reilly Media, 2016.

[11] N. Jovanovic, C. Kruegel, and E. Kirda, "Cross-site script inclusion attacks and defenses," in *Proceedings of the 2006 ACM Symposium on Information, computer and communications security*, 2006, pp. 237–246.

[12] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. Henry, R. Bradshaw, and Nathan, "Flumejava: Easy, efficient data-parallel pipelines," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, B. G. Zorn and A. Aiken, Eds. 2 Penn Plaza, Suite 701 New York, NY 10121-0701: ACM, 2010, pp. 363–375. [Online]. Available: http://dl.acm.org/citation.cfm?id=1806638

[13] "Google front end service." [Online]. Available: https://cloud.google.com/docs/security/infrastructure/design#google-frontend-service

[14] L. Weichselbaum, M. Spagnuolo, S. Lekies, and A. Janc, "Csp is dead, long live csp! on the insecurity of whitelists and the future of content security policy," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16, E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, Eds. New York, NY, USA: Association for Computing Machinery, 10 2016, p. 1376–1387. [Online]. Available: https://doi.org/10.1145/2976749.2978363

[15] P. Wang, B. Gumundsson, and K. Kotowicz, "Adopting trusted types in productionweb frameworks to prevent dom-based cross-site scripting: A case study," in *2021 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, 9 2021, pp. 60–73. [Online]. Available: https://doi.org/10.1109/eurospw54576.2021.00013

[16] "Protect your resources from web attacks with fetch metadata." [Online]. Available: https://web.dev/articles/fetch-metadata

[17] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *arXiv preprint arXiv:1801.01203*, vol. abs/1801.01203, pp. 1–19, 1 2018. [Online]. Available: https://export.arxiv.org/pdf/1801.01203

[18] A. Barth, C. Jackson, and J. C. Mitchell, "Robust defenses for cross-site request forgery," in *Proceedings of the 15th ACM conference on Computer and*

communications security, P. Ning, P. F. Syverson, and S. Jha, Eds. ACM, 10 2008, pp. 75–88. [Online]. Available: https://www.adambarth.com/papers/2008/barth-jackson-mitchell-b.pdf

[19] J. Wang, X. Wang, X. Yang, Y. Wu, and P. Wang, "An efficient approach for detecting csrf vulnerabilities in web applications," in *2020 15th International Conference on Computer Science & Education (ICCSE)*. IEEE, 2020, pp. 587–592.

[20] "Externalizing the google domain tiers concept," 2024. [Online]. Available: https://bughunters.google.com/blog/4562175388155904/externalizing-the-google-domain-tiers-concept

[21] "Migrate to user-agent client hints." [Online]. Available: https://web.dev/articles/migrate-to-ua-ch

[22] "A recipe for scaling security," 2024. [Online]. Available: https://bughunters.google.com/blog/5896512897417216/a-recipe-for-scaling-security

[23] A. Klein, "Dom based cross site scripting or xss of the third kind," *Web Application Security Consortium Articles 4*, 2005.

[24] S. Maffeis and J. C. Mitchell, "Privileges in web applications," in *2006 IEEE Symposium on Security and Privacy (S&P'06)*. IEEE, 2006, pp. 11–pp.

[25] Y. Yuan, H. Chen, Z. Mao, T. Xie, W. Yu, and C. Zou, "Assessing the value of the same-origin policy for client-side web security," in *2009 30th IEEE Symposium on Security and Privacy*. IEEE, 2009, pp. 125–138.

[26] P. De Ryck, N. Nikiforakis, L. Desmet, and W. Joosen, "Cross-origin resource sharing (cors)," in *2013 IEEE 27th International Conference on Advanced Information Networking and Applications Workshops*. IEEE, 2013, pp. 211–218.

[27] "Bazel build system." [Online]. Available: https://bazel.build

[28] "Closure tools." [Online]. Available: https://developers.google.com/closure