If It's Not Secure, It Should Not Compile: Preventing DOM-Based XSS in Large-Scale Web Development with API Hardening

Pei Wang, Julian Bangert, Christoph Kern
Security Engineering Research
Google
{pwng, bangert, xtof}@google.com

Abstract—**With tons of efforts spent on its mitigation, Cross-site scripting (XSS) remains one of the most prevalent security threats on the internet. Decades of exploitation and remediation demonstrated that code inspection and testing alone does not eliminate XSS vulnerabilities in complex web applications with a high degree of confidence.**

This paper introduces Google's secure-by-design engineering paradigm that effectively prevents DOM-based XSS vulnerabilities in large-scale web development. Our approach, named *API hardening*, enforces a series of company-wide secure coding practices. We provide a set of secure APIs to replace native DOM APIs that are prone to XSS vulnerabilities. Through a combination of type contracts and appropriate validation and escaping, the secure APIs ensure that applications based thereon are free of XSS vulnerabilities. We deploy a simple yet capable compile-time checker to guarantee that developers exclusively use our hardened APIs to interact with the DOM. We make various of efforts to scale this approach to tens of thousands of engineers without significant productivity impact. By offering rigorous tooling and consultant support, we help developers adopt the secure coding practices as seamlessly as possible. We present empirical results showing how API hardening has helped reduce the occurrences of XSS vulnerabilities in Google's enormous code base over the course of two-year deployment.

Index Terms—**Web security, cross-site scripting, language-based security, empirical software engineering**

I. Introduction

The history of cross-site scripting (XSS) can be dated back to the last few years of the 90s. Ever since, it continues to be one of the major sources of web security threats, identified as a top-10 vulnerability by the Open Web Application Security Project (OWASP) [1]. Fresh industry reports disclose that, to date, XSS is still among the most prevalent and exploitable security vulnerabilities in web applications [2], [3]. According to HackerOne, one of the most popular security bounty award platforms, XSS tops the chart of reported vulnerabilities by volume through 2018, with the average severity being "critical" [4]. The vulnerability reward program hosted by Google has awarded external experts millions of dollars for reporting XSS bugs in Google's products [5].

In general, XSS vulnerabilities stem from the lack of or inappropriate validation and sanitization of inputs to web applications from untrusted *sources* which later flow into a *sink* where the inputs are interpreted as executable code. One of the most commonly exploited sinks is the Document Object Model (DOM) APIs in client-side JavaScript code. With the boom of single-page applications (SPA), more and more of the dynamics in web applications are now being handled by browsers, making the threat of DOM-based XSS more ominous than ever before.

XSS vulnerabilities are often severe because they can bypass the same-origin policy, one of the foundations of the web application security model. The reason why XSS has been an undying security nightmare runs deep into the complexity and subtlety of the web technology stack. Mitigating measures such as the content security policy (CSP) [6], [7] have been developed, which make vulnerabilities harder to exploit but cannot eliminate them. Some mitigations, such as XSS filters [8] and web application firewalls (WAF), only passively block known categories of attacks and have subtle security issues themselves [9]. On the other hand, active mitigations like HTML sanitizing and URL escaping impose considerable cognitive load on developers, requiring them to develop a deep understanding about how to properly handle untrustworthy inputs in different scenarios. Some organizations can install security training programs in an attempt to help engineers grasp that knowledge, but oftentimes this is insufficient to prevent them from wirting XSS-prone code even for moderately complex applications [10].

In this paper, we introduce *API hardening*, a novel software engineering paradigm that proactively rids large web applications of DOM-based XSS vulnerabilities. Instead of trying to mitigate or remedy security threats after the code is shipped, our method focuses on security assurance in the early stages of software development. By deploying a series of technical enforcement controls, we firmly guide developers towards writing code that can be shown to be free of XSS by a simple and fast analyzer blended into the compiler.

The technical aspects of API hardening are two fold. First, we extend the JavaScript and TypeScript compilers to forbid the use of DOM APIs that are susceptible to code injection. These checks utilize type information but are otherwise flow insensitive, therefore can be made extremely scalable. Second, we offer a set of safe APIs to allow developers to utilize XSS-prone APIs in a secure way. We introduce special types to designate values that are safe to use in the context of injection-prone DOM sinks. In this way, all potentially dangerous DOM manipulation and code execution are highly regulated.

Indeed, with API hardening, developers are restricted to a strict subset of the DOM APIs. In some cases, this subset may not be sufficiently expressive. To accommodate such use cases, we establish a communication channel through which developers can request exemptions from some extended compiler checks for certain pieces of code. An easy-to-follow protocol is designed to ensure those requests are appropriately reviewed by security experts. Approved exemptions are added into a centrally managed list that is automatically respected by compilers. The practicality of our approach depends on an API design that is versatile enough to cover the vast majority of use cases encountered in practice, so that the overhead of this exemption process is rarely encountered. Our study shows that the need for exemptions can be made very rare, relative to the size of a humongous code base and a notably large developer community.

The adoption of API hardening requires a software development paradigm shift, which can be extremely challenging in a large organization. To render the adoption of API hardening as seamless as possible, we carefully design multiple supportive mechanisms to help developers adapt themselves to the new APIs and coding practices.

We have applied API hardening to real software production at Google, covering tens of thousands of engineers who develop some of the most complex web applications in the world. The adoption of API hardening is at the scale of the entire company's JavaScript and TypeScript code base for over two years. Empirical data show that our approach has a significant contribution to reducing the occurrences of XSS vulnerabilities in Google's humongous code base. Analyses on communication logs between web developers and security experts demonstrate that the exemption review protocol associated with the API hardening adoption process is effective and scalable.

Our contributions can be summarized as the following,

- We propose API hardening, a new software development paradigm that employs compile-time security checks and safe API designs to effectively prevent XSS vulnerabilities in monolithic code repositories owned by large software development teams.
- We extend JavaScript and TypeScript compilers to capture known dangerous DOM operations and offer developers a specially designed DOM-manipulating library. The major part of these extensions has been open sourced. See Appendix A for availability details.
- We developed a set of software engineering protocols to help tens of thousands of engineers in the same organization adopt the hardened APIs without significantly degrading their productivity.
- We collected and analyzed a rich amount of data during over two years of API hardening effort in real-world software development at Google. Empirical results demonstrate that our approach is effective and scales to tens of millions of lines of code.

```
1  <html>
2   <title>Search Results</title>
3   <body>
4    <p>You are searching: <span id="kw"></span></p>
5    <script>
6     const untrustedInput = new URLSearchParams(
7       location.search).get('kw');
8     const keyword = document.getElementById('kw');
9     keyword.innerHtml = untrustedInput; // XSS!
10   </script>
11  </body>
12 </html>
```

Fig. 1: Example Web Application with a DOM-Based XSS Vulnerability

II. Cross-Site Scripting

This section explains, with a tiny yet representative example, the concept of cross-site scripting and what consequences a XSS vulnerability can cause when exploited by malicious parties.

Consider the code in Fig. 1 which illustrates a web search application. It contains a client-side XSS vulnerability at line 9, where the JavaScript code consumes untrusted user input and render it as HTML markup through the `innerHTML` property of a DOM element. In this example, there is no dynamic server-side page rendering. It is a typical client-side XSS vulnerability, aka. DOM-based XSS.

Suppose this vulnerable application is served on `app.com`. A malicious website `evil.com` can exploit the vulnerability by posting a URL pointing to `app.com` on its own page. It then tricks the users of `app.com` to click on it. When a victim follows this URL and gets navigated to `app.com`, the search keyword encoded in the URL is injected into the search page as HTML markup. Since `app.com` does not sanitize the query parameter, `evil.com` can insert any content, including malicious JavaScript code that can be executed in the context of `app.com`.[1] When that happens, the capability of the malicious code is no longer restricted by the same-origin security policy and can therefore abuse users' trust on `app.com`. What happens next is entirely at the discrepancy of the attacker. They can read information private to `app.com` like cookies and send the information back to `evil.com`. They can also choose to exploit a vulnerability in the browser to further launch more advanced attacks [11]–[15].

For readability, the example in Fig. 1 is artificially simple, which may convey a false impression that XSS vulnerabilities are not difficult to detect or mitigate. In reality, the information flow from XSS sources to sinks can be extremely complex and often goes through servers and databases [10]. Traditional static analyzers have very limited capabilities to reason about this kind of software behavior.

[1]An example malicious URL is http://app.com?kw=⟨img%20src="x"%20onerror="alert(%27XSS%27)"⟩. In this case, the attacker utilizes the "onerror" event handler of an `<img>` tag to execute JavaScript code.

III. Design

Most previous work on thwarting the threat of XSS tackles the problem in the later phases of the software life cycle, e.g., whole-application static analysis [16]–[18], sophisticated automatic testing [19], [20], and run-time protections in production [7], [8]. As the functionality and complexity of software systems grow, the loss caused by critical security flaws and the cost of remedying them after they manifest has skyrocketed. In response to this trend, our approach confronts the challenge in the very early stage of web application development and nips XSS vulnerabilities in the bud.

A. Overview

API hardening operates by combining two technical components, i.e., 1) a fairly simple and fast static security analyzer embedded in the compiler, forbidding the use of the XSS-prone APIs and 2) a collection of specially designed types and APIs that reduce the task of demonstrating the absence of XSS vulnerabilities to a type checking problem.

Most XSS-prone APIs relate to certain combinations of DOM elements and properties. For example, in Fig. 1, the `innerHTML` property of HTML elements is an XSS sink. To prevent these APIs from being used, our compile-time security analyzer walks the abstract syntax tree of a program and emits errors on all occurrences of such APIs. In that sense, the analysis is flow-insensitive, since it does not consider any contextual information about these API uses and plainly asks the compiler to reject all of them. However, we do make use of type information to improve the precision of the analyzer. For example, the `src` property of an HTML element is by default an XSS sink. However, `src` on `<img>` is an exception since no modern browsers will invoke the JavaScript execution engine when loading `<img>` tags, even if the URL provided by the `src` property has the `javascript:` scheme.[2] With type information, the analyzer can distinguish different kinds of HTML elements and reduce false positives.

Indeed, JavaScript is not a typed language. However, many JavaScript compilers can infer or allow developers to annotate their code with type information to capture programming errors and seize more optimization opportunities during pre-deployment transformations [21], [22]. Annotating JavaScript with types has become a common practice in web development. There are now language variants of JavaScript that have native type systems, e.g., CoffeeScript and TypeScript. Therefore, requiring type information for our API hardening analyzer does not incur additional cost in practice.

At Google, we support API hardening in both JavaScript and TypeScript. Almost all Google's JavaScript code is annotated with static type information and developers are required to run a compiler on their code for optimization before deploying the code in production. Our JavaScript analyzer implementation is based on the Closure compiler [21], an open source JavaScript compiler capable of advanced type inference. Our TypeScript analyzer is implemented by extending the official TypeScript compiler.[3]

[2]Internet Explorer 6 is the last widely used browser known to be vulnerable to XSS through the `src` attribute on `<img>` tags.

B. Benefits

1) Effectiveness: Both the compiler and library-augmented APIs start to take effects from the very beginning of software development. In contrast, heavy-weight program inspection tools require developers to allocate dedicated time to run the analysis over a mostly completed project and the feedback will likely not be available in real time. Various research [23]–[25] suggests that code smells reported by an automatic analyzer after code has already been submitted are less likely to be addressed, mostly because the developer has already moved on to another task. Our approach relies on a combination of standard type checks and custom security checks during regular compilation, thus providing almost instant feedback to developers.

2) Availability: Being an interpreted language, JavaScript does not need to be compiled to run in browsers. However, it is now a common practice for web application vendors and publishers to perform a series of analyses and source-to-source transformations using a compiler before deploying the applications, such as obfuscation and minification [26]. Google has a monolithic code repository [27] and our engineers work within a unified compilation environment. Code with compilation errors is not allowed to be committed into the repository. By embedding the analyzer into compilers, it is automatically available to all developers and we can save massive delivery and education cost.

3) Efficiency: Google engineers maintain over two billion lines of code, a significant portion of which is JavaScript. The JavaScript compiler is invoked an exceedingly large number of times on a daily basis for development and continuous integration. Engineers build their code using centrally-maintained, distributed build infrastructure. Even tiny performance degradation of the compiler can cost a huge amount of extra CPU hours. JavaScript is known to be unfriendly to static analysis [28]. Employing advanced static XSS detection algorithms in a code base of the size we consider is unlikely to be cost-effective.

IV. XSS Sinks

A surprisingly large number of DOM APIs can cause XSS if they are used with partly user controlled inputs [29], [30]. The first step towards API hardening is to identify the originally dangerous DOM APIs, aka. XSS sinks.

A. Enumerate XSS Sinks

Essentially, a DOM API is prone to XSS if and only if the JavaScript engine will be invoked to interpret the input of the API as executable code. We collect such APIs by examining the HTML and DOM specifications, which, formalized as Web IDL [31], document all canonical APIs that can trigger

[3]For the sake of convenience, in the rest of the paper, TypeScript is regarded as a dialect of JavaScript. All discussions related to JavaScript apply to TypeScript unless otherwise noted.

JavaScript code execution. Previous work relies on the same source to identify possibly misused web APIs [32]. Our analysis on the specifications are purely manual, which is manageable since collecting XSS sinks is mostly one-time effort. Appendix B lists the XSS sinks we inferred from the HTML5 and DOM specifications.

The specifications can cover most of the XSS sinks found in practice. However, browser vendors may not fully follow specifications and can implement non-standard features that are prone to XSS. An example of such features is content sniffing, with which browsers may ignore the metadata of a data blob and interpret the type of the blob by heuristically analyzing its content. Skilled attackers can exploit this feature to manipulate browsers and carry out code executions not expected by web developers and users [33].

It is unrealistic to cover all browser-specific XSS sinks because many browsers have undocumented behaviors. As our best effort, we work with the developers of some mainstream browsers to stay informed about new browser features that may have security implications. Most of the time, we do not consider ancient browsers which tend to carry more XSS attack vectors. This is not a problem since our web applications do not support those browsers, either.

We would like to mention that many other XSS countermeasures, such as data-flow analyses, also need to identify a reasonably comprehensive set of XSS sinks to be effective. Therefore, enumerating the sinks are somewhat orthogonal to API hardening.

B. XSS Sink Classification

In total, we have identified five kinds of XSS sinks. The classification is based on the mechanism bound to the sink that directs the JavaScript engine to execute untrusted code. The rest of the section elaborates on the nature of each of these categories.

Code Execution Sinks: JavaScript and DOM have several APIs that accept string values and evaluate them as JavaScript code. The interpreted code will run in the same context as these sinks. If outsiders can control the values fed to these APIs, it will lead to the most straightforward cross-site scripting exploits. Typical sinks of this kind include `eval` and the `innerHTML` property of `<script>`.

URL Navigation Sinks: Many DOM APIs interpret strings as interactive URLs that navigate users to other web resources. In modern web applications, URLs can have schemes with rich semantics attached and can direct browsers to perform complicated actions, including executing arbitrary code. For example, following URLs of the "`javascript:`" scheme causes immediate code execution. Therefore, DOM APIs accepting navigational URLs are in general prone to XSS vulnerabilities.

Loadable URL Sinks: In some cases, DOM URLs are not for users to interact with. Instead, they are used to instruct browsers to request and load additional resources needed to render web pages, including executable JavaScript code. Typical examples include the `src` property of `<script>` and `<link>` elements. Attackers can perform cross-site scripting by injecting URLs pointing to contents they control. The Content-Security Policy (CSP) is a countermeasure against XSS attacks through loadable URL injection, but there are many cases where CSP is ineffective or can be sophisticatedly bypassed [34].

HTML Sinks: Some DOM sinks interpret string values as arbitrary HTML markup. The most straightforward way to exploit those sinks is to inject JavaScript code marked by the `<script>`. In some complicated attacks, HTML sinks can be used to spawn other kinds of sinks.

CSS Sinks: There are DOM APIs in JavaScript for developers to dynamically control how browsers render HTML elements by changing their associated Cascading Style Sheets (CSS). In ancient browsers like IE 6 and IE 7, JavaScript code can be embedded into CSS and will be executed when the style sheets are loaded. Although this is not longer the case in modern browsers with enhanced security, academic research has revealed that CSS sinks can cause more subtle attacks. For example, one of the attack methods is to inject CSS configurations that make browsers perform time-consuming UI rendering [35]. By monitoring how much time is spent in redrawing the UI, attackers can infer the content of a web page at the character granularity. Strictly speaking, this kind of attacks are not cross-site scripting since they do not involve JavaScript code execution. Nevertheless, they are similar enough for API hardening to keep CSS sinks on the list of banned DOM APIs with little additional cost.

In addition to the standard JavaScript and DOM APIs, we also consider XSS-prone sinks in third-party libraries that are commonly used inside our company. For example, the `createDom` function in the Closure JavaScript library [36] can create DOM elements with some user-provided values as attributes. This is one of the most dangerous sinks that can lead to XSS and therefore is included into our list of prohibited APIs. Expanding the compiler checks to library functions fits well into our approach.

V. Safe APIs

With the XSS-prone APIs identified and forbidden in development, we need to provide developers with a set of "hardened" APIs as alternatives. These safe APIs put limits on the manners in which developers can interact with the DOM and JavaScript engine, making it almost impossible for them to accidentally write vulnerable code, without requiring every developer to have a deep understanding about the security nature of every component in the web development stack.

There are three components in the design of Safe APIs,

- Safe types. These types designate values that are safe to flow to security sensitive sinks.
- Safe builders. These are the factory APIs that can produce values of safe types. Safe type values can only be constructed through these APIs.
- Safe sinks. These are the safe alternatives of the XSS-prone JavaScript and DOM sinks. Unlike the original

DOM sinks that accept strings, safe sinks only accept values of safe types.

Among these three components, safe types are the connections between safe builders and safe sinks. Essentially, the type checking process performed by the compiler proves that any values flowing into XSS-prone sinks are produced by an appropriate kind of safe builders.

A. Safe Types

We designed six safe types to cover the major attack vectors in the JavaScript language and DOM. Each safe type corresponds to a type of sinks as described in Section IV. CSS sinks are covered by two safe types. The correspondence is displayed in Table I.

Values of safe types (denoted by "safe values") are immutable once created. Beneath the surface, safe types are wrappers of primitive strings. The internals of safe types are hidden from application developers through the implementation language's visibility and type encapsulation features. In TypeScript, we mark the internals as private properties of the type. In JavaScript, we implement additional compile-time checks that forbid any code from accessing the internal properties. These private properties also prevent safe types from being confused with other user-created types, which is otherwise a problem due to JavaScript and TypeScript being structurally typed. In structural typing, the partial order of types is determined by the actual structures of the types instead of type names or places of declaration, in contrast to nominal typing. However, safe types are essentially markers stating values are constructed by following certain contracts and therefore have to be nominal. Defining private properties is a common way to simulate nominal types in JavaScript and TypeScript.

Our type encapsulation is not expected to be fully resistant to code that intentionally attempts to circumvent the intent of our hardened APIs (see Section VIII-A); rather, we designed the types and corresponding static checks such that code trying to access the internals of safe types appears clearly non-idiomatic and stands out in code reviews.

B. Safe Builders

Builders of safe values are the crux of the security of API hardening. In general, safe values are secure for sensitive DOM sinks because our static checks and APIs ensure that they can only be constructed in manners known to be secure. There are four ways of constructing safe values.

Build from literal values: Recall that the nature of XSS vulnerabilities is that defective code allows attackers to control what values can flow to security-sensitive JavaScript and DOM sinks. Literal values, as they are hard-coded by application developers, cannot be manipulated by attackers. Therefore, they can be safely converted to safe types. It should be noted literal values are not inherently secure. For example, an engineer could hard-code a loadable URL that points to a third-party domain from which arbitrary, untrustworthy content could be loaded. We treat this concern as out of the scope of API hardening, and to which necessary additional mitigations apply. Since the values are part of the program text as literals, they are clearly apparent to code reviewers and amenable to analysis by linters and presubmit checks.

Build by sanitizing: We have built sanitizers to automatically transform untrustworthy strings into safe-to-render HTML markup or safe-to-follow URLs by removing parts that may lead to XSS. For instance, the HTML sanitizer prunes sensitive tags and possibly vulnerable combinations of tags and attributes. The URL sanitizer prunes potentially insecure URL schemes and parameters. Safe value builders based on sanitizing and escaping are convenient to use, but they cannot fulfill the needs of many development tasks because sanitizers need to be conservative. For example, sanitizers strip all inline JavaScript code in HTML tags since they lack the context to decide whether the code is secure.

Build from template: Template systems are commonly used in web development, mostly for rendering large volumes of structural contents [37]. Building safe values from templates can be regarded as a combination of building from constants and building by sanitizing. The bulk of a template is static while some small parts of it can be configured by run-time values. When rendering contents from a template, the rendering framework ensures that the dynamically constructed contents are properly validated and escaped with respect to their context [38]. Furthermore, to qualify as a safe builder in API hardening, the template has to be *strict* in its application of contextual escaping [10], i.e., it should not allow developers to suppress or alter the inferred context-specific escaping in any way. To facilitate composition, the template system accepts values of our safe types for substitution, in which case context-inferred validation and escaping can be temporarily suppressed in a type-safe manner.

Build from manually reviewed sources: Sometimes, the previously mentioned builders are not expressive enough to support the needs of complicated applications. We introduce a special type of builder to resolve this problem. They are basically "backdoors" in the safe API design in the sense that they can cast any value to a safe value. Uses of these builders are subject to mandatory code review by a security expert. We enforce the review requirement by classifying these builders themselves as prohibited XSS sinks. Security reviewers allow uses of these builders by adding the code into the exemption list of the static security analyzer.

C. Safe Sinks

Safe sinks are hardened versions of XSS-prone JavaScript and DOM APIs. Typically, safe sinks are type-safe wrappers of the original APIs. In other words, we built a natural transformation from the original sinks accepting plain string values to hardened sinks accepting safe values. Figure 2 shows some examples of these trivially transformed sinks in TypeScript.

However, not all safe sinks are trivial transformations of the original unsafe versions. Some of APIs are made non-trivial purely for convenience, e.g., they also support security efforts

TABLE I: Safe Types in API Hardening

Safe Type	Corresponding XSS Sink	Security Invariant
`SafeScript`	Code Execution Sinks	JavaScript code that is safe for browsers to execute.
`SafeUrl`	URL Navigation Sinks	URLs that are safe for browsers to follow.
`TrustedResourceUrl`	Loadable URL sinks	URLs pointing to resources that contain trusted JavaScript or CSS code.
`SafeHtml`	HTML Sinks	HTML that is safe to render in a user's browser.
`SafeStyle`	CSS Sinks	CSS declarations that can be safely used as in-line style values of HTML elements.
`SafeStyleSheet`	CSS Sinks	CSS declarations with path selectors that are safe to evaluate as a style sheet in browsers.

```
// Navigate window to a new URL.
window.location.href = 'https://example.com';
// Append new HTML markup to the current document.
document.write(html);
```

(a) Original APIs

```
// Signature of a safe setter for Location#href.
// Values assigned to the sink must have the
// `SafeUrl` type.
declare function safeSetLocationHref(
  loc: Location, url: SafeUrl): void;

// Signature of a safe version of Document#write.
// Values appended must have the `SafeHtml` type.
declare function safeDocumentWrite(
  doc: Document, html: SafeHtml): void;

// Navigate window to a safe URL built from literal
safeSetLocationHref(
  window.location,
  SafeUrl.fromLiteral('https://example.com'),
);
// Append sanitized HTML markup to the current
// document.
safeDocumentWrite(document, SafeHtml.sanitize(html));
```

(b) Hardened APIs

Fig. 2: Examples of Safe Sinks in TypeScript

complementary to API hardening. Others exist because certain HTML tags have more complicated security contracts. Below are some examples of each case.

- The hardened APIs for setting the `textContent` and `src` attributes of `<script>` tags additionally propagate the CSP nonces [34] so that the trusted JavaScript code is not blocked by browsers when dynamically inserted.
- Because the type of `href` is dependent on the value of `rel`, the `rel` and `href` properties of `<link>` tags are merged into a single safe sink in hardened APIs. For example, if `rel` is `"icon"`, `href` should accept `SafeUrl` values; if `rel` is `"stylesheet"`, `href` should accept `TrustedResourceUrl` values. We need a dependently typed API to cover this case, and it would be difficult to allow the two properties to be separately assigned.
- When setting the `src` property of `<iframe>` tags, we make sure the `sandbox` property is also set to minimize the risks of loading untrusted pages inside our own applications.

It is worth mentioning that the safe sinks is a more dynamic piece in API hardening compared with other components. The design of the hardened APIs should be constantly revisited as the engineering environment evolves. We continuously collect developer feedback to improve the ergonomics of the hardened APIs. Section VI details our effort from this aspect.

The run-time performance cost of using safe types and safe sinks is mostly negligible, since the security is enforced at compile time. Operations like HTML sanitization and CSP nonce propagation can indeed introduce some overhead, but they are not an essential part of API hardening. Even without employing safe types and hardened APIs, web applications will still need to perform these operations. In certain cases, API hardening even helps remove redundant dynamic security checks since safe types can statically propagate the security properties of the values.

VI. Adoption

The technical materialization of API hardening is only the first step towards its adoption in an organization with tens of thousands of web developers. We now introduce how we deploy the new static checks and safe APIs across Google.

A. *Deployment Process*

API hardening itself is not a static technique but is constantly evolving as new vulnerabilities and attacks emerge. In large organizations, even a minor change to coding practices can affect a tremendous volume of code. Asking engineers to refactor their legacy code with hardened APIs and pause developing new features would be prohibitively disruptive. Therefore, it is essential to roll out API hardening checks and new APIs in an incremental manner.

Whenever we introduce new security enforcement, we automatically exempt legacy violations so that existing code still compiles. Technically, this can be achieved by customizing the compiler or the build system. Our team maintains the exemption list of legacy source files and approves changes made to the list through reviews.

There are two ways for an organization to eliminate legacy API violations. The first choice is to actively refactor the code if the XSS risks have been historically high. Otherwise, when the code base is being developed at a rapid pace and the legacy code is expected to have a short life span, the organization can choose to passively wait for legacy violations being cleaned up through product iterations. At Google, we apply both methods on a per-project basis. For legacy violations sharing a common pattern, we use large-scale code refactoring tools like RefasterJS [39] to generate patches at scale.

B. Technical and Operational Support

We have developed accompanying measures to help developers become familiar with the new coding practice. The key objective is to allow developers to quickly unblock themselves when their code contains references to XSS-prone APIs and gets flagged by compilers. We also want to deepen developer understanding about XSS and the mitigating strategy of API hardening after they encounter first few API hardening violations, which can help avoid similar obstacles in the future.

1) Automatic Fix: When the compiler captures an API hardening violation in the code, we try to automatically construct a fix and present it along with the compiler error message. Previous research shows that code smell checks tend to be more useful if they provide fixes as it makes developers less confused and reduces the need for cognitive context switches [24]. We generate fixes with a relatively simple strategy that does not consider too much about the context of the violation, though more sophisticated auto repair or refactoring techniques [40]–[43] could be applied. In general, a suggested fix tries to enlighten developers on two points: 1) how to build safe values from the original values fed into the unsafe sinks, and 2) which safe API is appropriate for substituting the prohibited sink. If the generated fix can lead to unexpected program behavior, e.g., values being truncated due to sanitization, we explicitly highlight these parts in fixes to prompt developers to carefully review the changes.

2) Documentation and Offline Consultation: It is critical to have well maintained static documentation on API hardening and make it easily accessible by developers. In addition, we encourage developers to ask questions about safe types and hardened APIs in the company's internal discussion platform by embedding directive instructions and relevant Q&A tags into the error messages produced by compilers. Our team schedules a support rotation to provide timely responses to developer questions.

3) Exemption Review: On rare occasions, the security constraints imposed by API hardening make it difficult or infeasible for developers to implement certain functionality using safe APIs only. The API hardening team offers security review services for such cases. Similar to the offline consultation service, our team has a support rotation to respond to exemption requests. The security experts communicate with the developers through the Google's standard code review platform. If the exemption request is approved, the source file containing references to unsafe APIs will be added to the exemption list managed by the API hardening team. Before approving, the reviewer ensures that the use case indeed cannot be expressed by any hardened API. The reviewer then checks if the code is structured such that its security can be established with a high degree of confidence. If necessary, the reviewer will recommend code changes in the code review process.

C. Developer Feedback

To understand if the hardened APIs are friendly to developers, we integrate the XSS-prone sink detection into Tricorder [24], a scalable program analysis platform that allows developers to report the usefulness of the analysis results. Our team collects and analyzes feedback reports quarterly. Based on the analysis results, we make directed improvements to our API design and documentation.

VII. Results

We have conducted an empirical, *in situ* study on the effectiveness and practicality of API hardening in Google's production setting. All data presented in this section were collected from real-world enterprise web development activities.

It should be emphasized that although we support API hardening for both JavaScript and TypeScript, we lack the data to showcase the TypeScript aspect, since it is a relatively new language in the company. Everything presented in this section reflects the manifest of API hardening in our JavaScript code base.

A. Adoption Progress

We started deploying mandatory compile-time checks for all JavaScript code across Google in the first quarter of 2018.[4] In the second quarter of 2018, we launched infrastructures and data pipelines to continuously monitor the company-wide adoption progress.

Fig. 3 demonstrates the deployment pace of API hardening by showing the number of checked XSS-prone sinks and the total count of source files exempted from at least one of the checks at the beginning of each quarter. Note that our developers maintain over a million JavaScript files, so less than 1% are exempted by the end of 2020. Despite that we deprecate more XSS-prone sinks over time, the number of exempted source files, most of which are legacy, is decreasing. This shows that our deployment strategy is effective at reducing the technical debt across the company.

In addition to the number of exempted legacy files, we also measure the total number of program locations that violate API hardening standards, by compiling all JavaScript targets in Google's code base every few days. Fig. 4 shows the total number of legacy JavaScript code locations with security violations.[5] It also shows the top 10 XSS-prone sinks ranked by the number of legacy violations. Based on our internal statistics, these 10 sinks are also the most commonly attacked surfaces in XSS exploits known to us. As can be seen, the total number of legacy violations is constantly decreasing over the past months. The same trend applies to all popular subcategories of legacy violations.

As mentioned in Section VI-B2, we encourage developers to ask questions in the company's internal Q&A platform,

[4]We had previously developed our safe types and many of the hardened builder APIs and applied compile-time checks for select development projects on an opt-in basis

[5]The temporary downticks in the figures are artifacts caused by transient failures in our compilation pipeline.

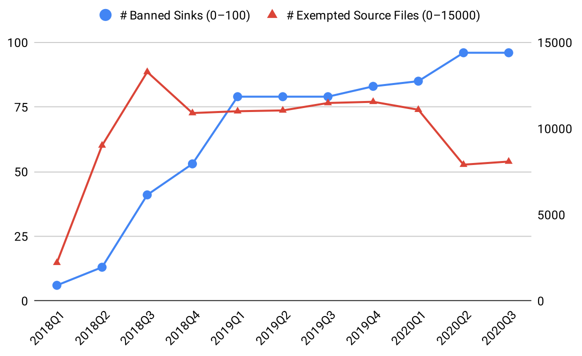
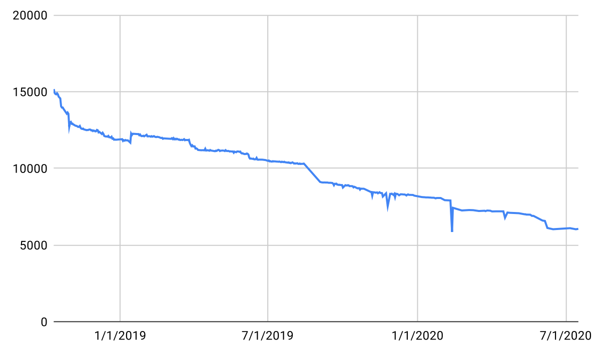
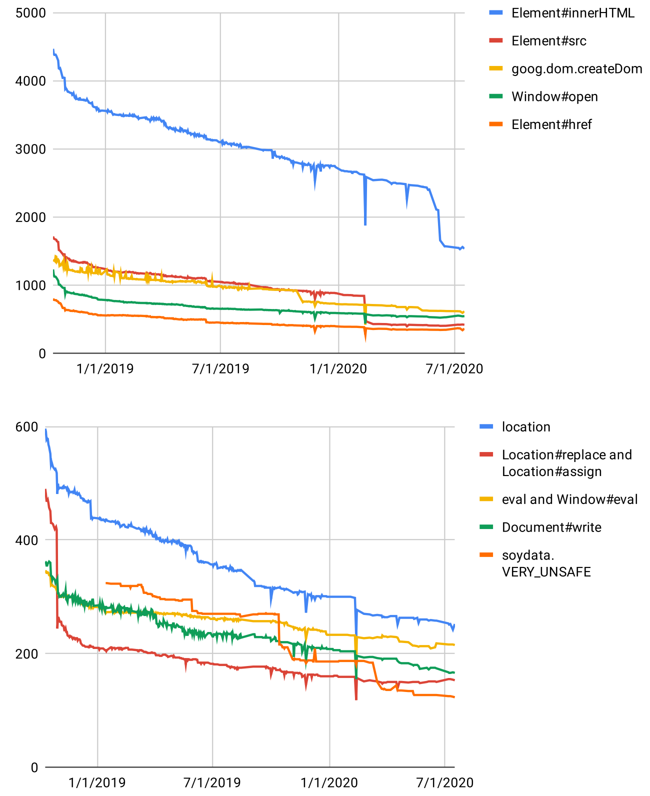


Fig. 3: The Numbers of Checked XSS-Prone Sinks and Exempted Source Files (Among the Total of Millions of Files)



(a) Total Number of Legacy Violations



(b) Top 10 Sinks Ranked By Number of Legacy Violations

Fig. 4: Daily Counts of Legacy Code Locations Violating API Hardening Checks

in case they encounter problems when adapting themselves to API hardening during development. To measure developer acceptability of API hardening, we count the number of questions raised in each week for the past two years, as displayed in Fig. 5. We received an elevated volume of questions during 2018-10-07 to 2018-10-27, during which we hosted a company-wide awareness campaign for the adoption of safe APIs. Engineers across the company participated in the event to fix legacy violations in their JavaScript projects and they asked many questions about how to proceed. After that, we averaged one question a week with a maximum of four questions. This very low volume of consulting traffic despite a substantial volume of development activity – our central code repository receives on the order of 10,000 submissions involving JavaScript source files per week. The contrast between development activity density and need for expert consulting suggests that developers adapt easily and without significant friction to our hardened and inherently-secure APIs.

B. *Effectiveness*

API hardening is a preventative countermeasure against XSS vulnerabilities. Considering the excessive scale of Google's web applications and the large number of engineers involved, it is exceedingly difficult to accurately measure the effectiveness of our approach with fine-grained controlled experiments.

Therefore, we instead investigate, with observational data, how API hardening has helped the company reduce the number of XSS vulnerabilities in shipped code. We have been actively encouraging external security researchers to report vulnerabilities discovered in our products under a "bug bounty" program, and a significant portion of reports are XSS vulnerabilities. We use this number as the major metric to measure the effectiveness of API hardening.

To exclude potential confounding factors as much as possible, we start with inspecting the vulnerability reduction data with a small-scale case study. In this case, we inspect the numbers of reported DOM-based XSS vulnerabilities in a single product, before, during, and after their adoption of API hardening, as presented in Table II. The product in the study was one of the most actively developed web applications at Google during the time of inspection. The team developing this product is one of the pioneer teams that strictly follow our secure coding practices. The team had a few tens of frontend engineers during the period listed in Table II. It took these developers roughly a year to refactor the code base and clean up most of the legacy violations. As can be seen, the numbers of DOM-based XSS vulnerabilities dropped drastically after the code base became in conformance with API hardening. This case convinced us that it is worthwhile to roll out API hardening as a universal software development paradigm for the whole company.

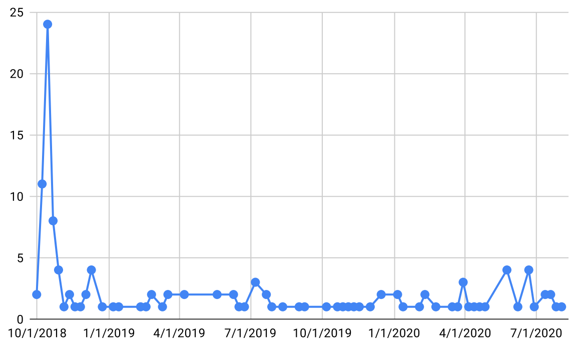


Fig. 5: The Numbers of Questions About API Hardening Asked by Engineers in Each Week

TABLE II: DOM-Based XSS Vulnerabilities Found in A Single Product Before and After Adopting API Hardening

Period	**# of DOM-based XSS**
Year Before Adoption	10
Year During Adoption	2
Year After Adoption	1

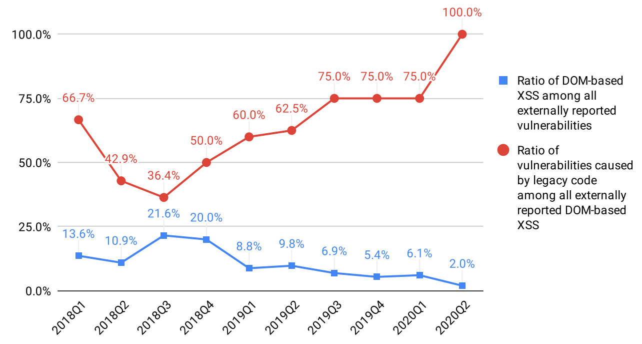


Fig. 6: Proportion of DOM-Based XSS Vulnerabilities in JavaScript Since the Inception of API Hardening

We now demonstrate the company-wide statistics about the effectiveness of our approach. The blue curve in Fig. 6 displays the ratio of DOM-based XSS among all externally reported vulnerabilities that are considered to be of high remediation priority. Meanwhile, the red curve in Fig. 6 shows that, among all externally reported DOM-based XSS vulnerabilities, how many of them are from legacy code that does not use our hardened APIs.

Since the inception of API hardening in the first quarter of 2018, the percentage of DOM-based XSS vulnerabilities has steadily declined. Indeed, this trend alone may not directly reflect the impact of API hardening, since the company-wide statistics may be contributed to by other security measures deployed during the sampled period of time. This concern is partially mitigated by the fact that, during the same period of time, the proportion of XSS vulnerabilities found in legacy code has been increasing significantly. Recall that in Section VII-A, we have shown the amount of legacy code exempted by API hardening checks is decreasing. All these trends combined together indicate that freshly written code, which is all covered by API hardening, is less likely to introduce new vulnerabilities.

With both the small-scale case study and the company-wide statistics, it is reasonable to conclude that applying API hardening in large-scale web development is very effective at reducing DOM-based XSS vulnerabilities.

C. Code Review Workload

One major part of the operational cost of employing API hardening is the manual effort of reviewing exemption requests from developers who feel safe types and hardened APIs are not expressive enough to implement the functionality they need. To demonstrate that this cost is completely manageable, we analyzed five months of code review activities related to JavaScript API hardening. The analyzed interval is from 2018-10-7 to 2019-2-10, which is the early period of large-scale adoption of API hardening in the company.

By manually analyzing our code review history, we classified the code reviews into three categories:

1) The request is valid and and can be approved without developers revising the code.
2) The request is valid but the code needs to be revised before we can approve it, leading to back-and-forth interactions between developers and reviewers.
3) The requested exemption is not necessary and therefore rejected. Developers are advised to use the adequate safe value builders to solve their problems.

The weekly counts of each kind of review activities are displayed in Fig. 7. As can be seen, the numbers of requests that received a revision or reject decision are mostly flat and stay at a low level. There were a high volume of trivially approved requests during the first few weeks, but then the number went down and stayed on the same level as other kinds of requests. On average, we received less than eight review requests each week during the analyzed period, which is fairly modest considering how many developers Google has and how much code they maintain. Similar to the trend of questions we received from the internal forum (Fig. 5), we received much more code review requests in the first few weeks due to the company-wide campaign for reducing technical debts in legacy JavaScript code. Most of these cleanup changes were trivially approved.

It is evident from Fig. 7 that deploying API hardening is much less laborious than it appears to be, even for an organization with tens of thousands of web developers. Although we do not have the detailed breakdown of code review requests after February 2019, we notice that *the ratio of security reviewed JavaScript code commits has been constantly below 0.1% at the time of writing this paper*. This also proves that the methodology of API hardening is very practical.

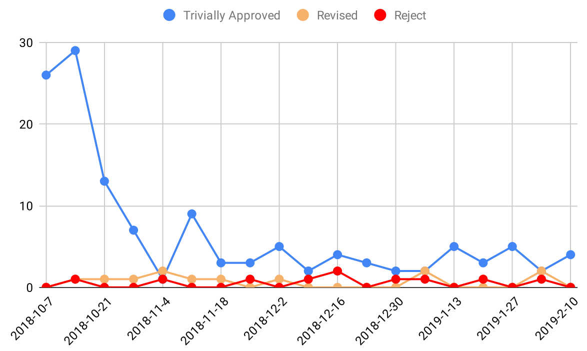


Fig. 7: Breakdown of Code Review Requests for API Hardening Policy Exemptions

VIII. Threats to Validity

A. Soundness

Due to the highly dynamic nature of JavaScript, the type system employed by existing JavaScript compilers[6] is unsound. The TypeScript language has a highly advanced native type system, but it is also designed to be unsound to reduce the cost of migrating legacy JavaScript projects to TypeScript. As a consequence, our type-based API prohibition analysis is also unsound. The code snippet below demonstrates a simple trick developers can use to evade compiler checks on assignments to the `href` property on `Location` objects, which is a common XSS sink.

```
let bypass =
  document.location as unknown as {href: string};
bypass.href = 'javascript:alert("XSS!")';
```

JavaScript typing is still being actively researched [45], [46]. To the best of our knowledge, there is no mature solution that can help us resolve the unsoundness problem at the scale of Google's development force and code base. Nevertheless, our experience has shown that consistent coding guidelines and mature code review practices make intentional security bypasses using non-idiomatic code patterns mostly a negligible concern.

B. Practicality in Other Organizations

In general, the methodology of API hardening does not depend on the specifics of the development environment or policies inside our company; however, these factors shaped some of the initial implementation. We aim to make DOM API hardening applicable to different development workflows, including non-commercial, open-source projects.

With the analyzer and safe API library available, any individual project can employ API hardening. However, as previously discussed, it requires additional efforts other than technical supports to apply API hardening to a large development organization. We identified several key factors that help minimize the adoption cost of API hardening,

- A mandatory code review process for submitting code.
- A monolithic code repository that has as few access-control restrictions as possible. This allows a small team of security experts to review all projects developed in the same organization.
- A code ownership mechanism that requires changes to certain files to be reviewed by designated personnel, i.e., the "owner" of the code.
- A unified build environment that makes sure all code is compiled by the same toolchain.

The more aforementioned traits an organization bears, the less costly it is for the organization to adopt API hardening, since they help amortize the deployment cost.

C. Server-Side Vulnerabilities

The approach introduced in this paper is for mitigating DOM-based XSS vulnerabilities in client code only. However, there is no fundamental difficulty in extending it to harden server-side code. The design and technical constructs discussed in Section III and Section V are not limited to JavaScript and can be extended to other languages. Our analyzer only requires typing information and is simple enough to be ported to common languages used for implementing web servers, e.g., PHP, Java, and C++.

IX. Related Work

A. XSS Countermeasures

The IE 8 browser introduced the XSS filter [8] that can detect on-going reflected XSS attacks when the same JavaScript payload appears in both the HTTP query and the returned web page. The content security policy [7], following BEEP [47] and SOMA [48] which are the earlier embodiment of similar ideas, provides a mechanism to allow websites to explicitly indicate what kinds of JavaScript code can be executed within their origins to prevent injection of attacker-controlled scripts. Both XSS filters and CSP try to mitigate existing XSS vulnerabilities and they are known to be bypassable by sophisticated attacks. In contrast, API hardening prevents XSS-infested code from being introduced into web applications in the first place.

Traditional pre-deployment XSS detection techniques can be either static or dynamic. Early static analyzers focus on reasoning about the information flow of server-side code written in PHP [16], [49] and Java [17]. Later advancement in this direction led to the capability of formalizing and analyzing a subset of the highly dynamic JavaScript language [50], [51]. Similarly, there have been proposals to improve the effectiveness of dynamic testing in detecting XSS vulnerabilities in the server-side [19] and client-side code [20], [52] of web applications. The major challenge for these techniques is that XSS exploits often have complicated information flows across different components in the network, e.g., servers, databases, and browsers. It is difficult for a single analysis tool to capture bugs across all these components.

[6]The type system implemented by the Closure compiler is based on the ECMAScript 4 specification [44].

B. Language and API Hardening

Empirical user studies show that bad API design has a significant negative effect on software quality [53]. A series of field studies on the usability of cryptographic libraries show that robust APIs and security-oriented warnings from development tools can help developers write more secure code [54], [55].

Sharing an idea similar to API hardening but applied to a different problem domain, Ironclad C++ [56] reduces the likelihood of developers writing memory-unsafe code by enforcing them to use a specially designed library for memory management. The library augments the type safety of the original C++ type system by introducing additional static constraints powered by new types and a new static analyzer. The Rust programming language infuses memory safety into its type system and makes sure that code with memory bugs does not compile [57]–[59]. In Java, the plugable type system [60] denoted through Java annotations allows software developers to statically enforce additional security properties in the code. Some other efforts like FixDroid [61], CogniCrypt [62], and ASIDE [63] try to help individual developers be more aware of security when they code with various UI supports, e.g., flagging likely insecure code in IDEs. In industry, Mozilla embeds security checks against a small number of well known XSS sinks in their browser extension development toolkit [64].

As far as we know, none of the existing work has explored how to aggressively enforce secure coding directives in a large organization like we do. Part of our work has evolved into an emerging web security standard called Trusted Types [65] which introduces run-time DOM API checks into browsers, ensuring that these APIs only accept non-spoofable and typed values rather than raw strings.

X. Conclusion

In this paper, we presented API hardening, a mature methodology to reduce the likelihood of developers writing code vulnerable to cross-site scripting. We shared our experience with applying this methodology to real-world, large-scale web application development. By deploying extra programming conformance rules enforced by compilers, we prevent developers from using the XSS-prone JavaScript and DOM APIs. We designed and implemented a set of safe APIs that allow developers to securely interact with the JavaScript engine and DOM. We have deployed API hardening inside Google, a large software development organization that builds some of the most complex web applications in the world. Empirical data shows that API hardening can effectively prevent XSS vulnerabilities from being introduced into the products with only modest cost.

Acknowledgement

API hardening is the collective effort of many current and former members of the Hardening team at Google's Information Security Engineering. We thank all of them for their contributions to the work. We thank Emilia Kasper, Jen Bourey, Thomas Shadwell, Emanuel Tesař, Damien Engels and anonymous reviewers for their constructive comments on the paper draft.

References

[1] "OWASP top 10 web application security risks." [Online]. Available: https://owasp.org/www-project-top-ten/

[2] "2019 application security statistics report," 2019. [Online]. Available: https://www.whitehatsec.com/resources/2019-application-security-statistics-report/

[3] "Web application attacks statistics," 2017. [Online]. Available: https://www.ptsecurity.com/upload/corporate/ww-en/analytics/Web-application-attacks-2018-eng.pdf

[4] "The HackerOne top 10 most impactful and rewarded vulnerability types." [Online]. Available: https://www.hackerone.com/top-10-vulnerabilities

[5] "Reshaping web defenses with strict content security policy." [Online]. Available: https://security.googleblog.com/2016/09/reshaping-web-defenses-with-strict.html

[6] J. Weinberger, A. Barth, and D. Song, "Towards client-side html security policies." in *Proceedings of the 6th USENIX Workshop on Hot Topics on Security*, 2011.

[7] S. Stamm, B. Sterne, and G. Markham, "Reining in the web with content security policy," in *Proceedings of the 19th International Conference on World Wide Web*, ser. WWW '10, 2010, p. 921–930.

[8] D. Ross, "IE8 security part IV: The XSS filter." [Online]. Available: https://docs.microsoft.com/en-us/archive/blogs/ie/ie8-security-part-iv-the-xss-filter

[9] D. Bates, A. Barth, and C. Jackson, "Regular expressions considered harmful in client-side XSS filters," in *Proceedings of the 19th International Conference on World Wide Web*, ser. WWW '10, 2010, p. 91–100.

[10] C. Kern, "Securing the tangled web," *Communications of the ACM*, vol. 57, no. 9, pp. 38–47, 2014.

[11] A. Barth, J. Weinberger, and D. Song, "Cross-origin JavaScript capability leaks: Detection, exploitation, and defense," in *Proceedings of the 18th USENIX Security Symposium*, ser. USENIX Security '09, 2009, p. 187–198.

[12] S. Chen, H. Chen, and M. Caballero, "Residue objects: A challenge to web browser security," in *Proceedings of the 5th European Conference on Computer Systems*, ser. EuroSys '10, 2010, p. 279–292.

[13] R. Rogowski, M. Morton, F. Li, F. Monrose, K. Z. Snow, and M. Polychronakis, "Revisiting browser security in the modern era: New data-only attacks and defenses," in *Proceedings of the 2nd IEEE European Symposium on Security and Privacy*, ser. EuroS&P '17. IEEE, 2017, pp. 366–381.

[14] S. Tang, H. Mai, and S. T. King, "Trust and protection in the Illinois browser operating system," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI '10, 2010, p. 17–31.

[15] C. Reis, A. Moshchuk, and N. Oskov, "Site isolation: Process separation for web sites within the browser," in *Proceedings of the 28th USENIX Conference on Security Symposium*, ser. USENIX Security '19, 2019, p. 1661–1678.

[16] G. Wassermann and Z. Su, "Static detection of cross-site scripting vulnerabilities," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08, 2008, p. 171–180.

[17] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman, "TAJ: Effective taint analysis of web applications," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '09, 2009, p. 87–97.

[18] F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis," in *In Proceeding of the 14th Network and Distributed System Security Symposium*, ser. NDSS '07, 2007.

[19] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst, "Finding bugs in dynamic web applications," in *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ser. ISSTA '08, 2008, p. 261–272.

[20] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, "A symbolic execution framework for JavaScript," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, ser. S&P '10, 2010, pp. 513–528.

[21] "Closure compiler." [Online]. Available: https://developers.google.com/closure/compiler

[22] "Babel: The compiler for next generation javascript." [Online]. Available: https://babeljs.io/

[23] E. Aftandilian, R. Sauciuc, S. Priya, and S. Krishnan, "Building useful program analysis tools using an extensible Java compiler," in *Proceedings of the 12th International Working Conference on Source Code Analysis and Manipulation*, ser. SCAM '12, 2012, pp. 14–23.

[24] C. Sadowski, J. v. Gogh, C. Jaspan, E. Söderberg, and C. Winter, "Tricorder: Building a program analysis ecosystem," in *Proceedings of the 37th IEEE International Conference on Software Engineering*, ser. ICSE '15, 2015, pp. 598–608.

[25] D. Distefano, M. Fähndrich, F. Logozzo, and P. W. O'Hearn, "Scaling static analyses at Facebook," *Commun. ACM*, vol. 62, no. 8, p. 62–70, Jul. 2019.

[26] P. Skolka, C.-A. Staicu, and M. Pradel, "Anything to hide? studying minified and obfuscated code in the web," in *Proceedings of the 30th The World Wide Web Conference*, ser. WWW '19, 2019, p. 1735–1746.

[27] C. Jaspan, M. Jorde, A. Knight, C. Sadowski, E. Smith, C. Winter, and E. Murphy-Hill, "Advantages and Disadvantages of a Monolithic Repository: A Case Study at Google," in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice Track*, ser. ICSE-SEIP '18, 2018, pp. 225–234.

[28] G. Richards, S. Lebresne, B. Burg, and J. Vitek, "An analysis of the dynamic behavior of javascript programs," in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 1–12. [Online]. Available: https://doi.org/10.1145/1806596.1806598

[29] "Html5 security cheetsheet." [Online]. Available: https://html5sec.org/

[30] M. Zalewski, *The tangled Web: A guide to securing modern web applications*. No Starch Press, 2012.

[31] "Web IDL." [Online]. Available: http://www.w3.org/TR/WebIDL

[32] S. Bae, H. Cho, I. Lim, and S. Ryu, "SAFEWAPI: Web API misuse detector for Web Applications," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '14, 2014, p. 507–517. [Online]. Available: https://doi.org/10.1145/2635868.2635916

[33] A. Barth, J. Caballero, and D. Song, "Secure content sniffing for web browsers, or how to stop papers from reviewing themselves," in *Proceedings of the 30th IEEE Symposium on Security and Privacy*, ser. S&P '09, 2009, pp. 360–371.

[34] 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, 2016, p. 1376–1387.

[35] M. Heiderich, M. Niemietz, F. Schuster, T. Holz, and J. Schwenk, "Scriptless attacks: Stealing the pie without touching the sill," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12, 2012, p. 760–771.

[36] "Closure library." [Online]. Available: https://developers.google.com/closure/library

[37] D. Gibson, K. Punera, and A. Tomkins, "The volume and evolution of web page templates," in *Special Interest Tracks and Posters of the 14th International Conference on World Wide Web*, ser. WWW '05, 2005, p. 830–839.

[38] M. Samuel, P. Saxena, and D. Song, "Context-sensitive auto-sanitization in web templating languages using type qualifiers," in *Proceedings of the 18th ACM conference on Computer and communications security*, ser. CCS '11, 2011, pp. 587–600.

[39] "RefasterJS." [Online]. Available: https://github.com/google/closure-compiler/wiki/RefasterJS

[40] Q. Gao, Y. Xiong, Y. Mi, L. Zhang, W. Yang, Z. Zhou, B. Xie, and H. Mei, "Safe memory-leak fixing for C programs," in *Proceedings of the 37th IEEE/ACM 37th IEEE International Conference on Software Engineering*, ser. ICSE '15, 2015, pp. 459–470.

[41] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "Context-aware patch generation for better automated program repair," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18, 2018.

[42] X. Xu, Y. Sui, H. Yan, and J. Xue, "VFix: Value-flow-guided precise program repair for null pointer dereferences," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19, 2019, pp. 512–523.

[43] K. An and E. Tilevich, "Client insourcing: Bringing ops in-house for seamless re-engineering of full-stack JavaScript applications," in *Proceedings of The Web Conference 2020*, ser. WWW '20, 2020, p. 179–189.

[44] "Proposed ECMAScript 4th edition – language overview." [Online]. Available: {}https://www.ecma-international.org/activities/Languages/Language%20overview.pdf

[45] B. S. Lerner, J. G. Politz, A. Guha, and S. Krishnamurthi, "TeJaS: Retrofitting type systems for JavaScript," in *Proceedings of the 9th Symposium on Dynamic Languages*, ser. DLS '13, 2013, p. 1–16.

[46] S. Chandra, C. S. Gordon, J.-B. Jeannin, C. Schlesinger, M. Sridharan, F. Tip, and Y. Choi, "Type inference for static compilation of javascript," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '16, 2016, p. 410–429.

[47] T. Jim, N. Swamy, and M. Hicks, "Defeating script injection attacks with browser-enforced embedded policies," in *Proceedings of the 16th International Conference on World Wide Web*, ser. WWW '07, 2007, p. 601–610.

[48] T. Oda, G. Wurster, P. C. van Oorschot, and A. Somayaji, "SOMA: Mutual approval for included content in web pages," in *Proceedings of the 15th ACM Conference on Computer and Communications Security*, ser. CCS '08, 2008, p. 89–98.

[49] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: a static analysis tool for detecting web application vulnerabilities," in *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, ser. S&P '06, 2006.

[50] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner, "Staged information flow for javascript," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '09, 2009, p. 50–62.

[51] S. Guarnieri, M. Pistoia, O. Tripp, J. Dolby, S. Teilhet, and R. Berg, "Saving the world wide web from vulnerable JavaScript," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ser. ISSTA '11, 2011, p. 177–187.

[52] W. Melicher, A. Das, M. Sharif, L. Bauer, and L. Jia, "Riding out DOMsday: Towards detecting and preventing DOM cross-site scripting," in *2018 Network and Distributed System Security Symposium (NDSS)*, ser. NDSS '18, 2018.

[53] D. S. Oliveira, T. Lin, M. S. Rahman, R. Akefirad, D. Ellis, E. Perez, R. Bobhate, L. A. DeLong, J. Cappos, and Y. Brun, "API blindspots: Why experienced developers write vulnerable code," in *Proceedings of the 14th Symposium on Usable Privacy and Security*, 2018, pp. 315–328.

[54] M. Green and M. Smith, "Developers are not the enemy!: The need for usable security apis," *IEEE Security Privacy*, vol. 14, no. 5, pp. 40–46, 2016.

[55] P. L. Gorski, L. L. Iacono, D. Wermke, C. Stransky, S. Möller, Y. Acar, and S. Fahl, "Developers deserve security warnings, too: On the effect of integrated security advice on cryptographic {API} misuse," in *Proceedings of the 14th Symposium on Usable Privacy and Security*, 2018, pp. 265–281.

[56] C. DeLozier, R. Eisenberg, S. Nagarakatte, P.-M. Osera, M. M. Martin, and S. Zdancewic, "Ironclad C++: A library-augmented type-safe subset of C++," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '13, 2013, p. 287–304.

[57] H. Wang, P. Wang, Y. Ding, M. Sun, Y. Jing, R. Duan, L. Li, Y. Zhang, T. Wei, and Z. Lin, "Towards memory safe enclave programming with Rust-SGX," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19, 2019, p. 2333–2350.

[58] P. Wang, Y. Ding, M. Sun, H. Wang, T. Li, R. Zhou, Z. Chen, and Y. Jing, "Building and maintaining a third-party library supply chain for productive and secure SGX enclave development," in *Proceedings of the 42nd International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP '20, 2020.

[59] A. N. Evans, B. Campbell, and M. L. Soffa, "Is Rust used safely by software developers?" in *Proceedings of the 42nd International Conference on Software Engineering*, ser. ICSE '20, 2020.

[60] M. M. Papi, M. Ali, T. L. Correa, J. H. Perkins, and M. D. Ernst, "Practical pluggable types for java," in *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ser. ISSTA '08, 2008, p. 201–212.

[61] D. C. Nguyen, D. Wermke, Y. Acar, M. Backes, C. Weir, and S. Fahl, "A stitch in time: Supporting Android developers in writing secure code,"

in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17, 2017, p. 1065–1077.

[62] S. Krüger, S. Nadi, M. Reif, K. Ali, M. Mezini, E. Bodden, F. Göpfert, F. Günther, C. Weinert, D. Demmler, and R. Kamath, "CogniCrypt: Supporting developers in using cryptography," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '17, 2017, pp. 931–936.

[63] J. Xie, B. Chu, H. R. Lipford, and J. T. Melton, "Aside: Ide support for web application security," in *Proceedings of the 27th Annual Computer Security Applications Conference*, ser. ACSAC '11, 2011, p. 267–276.

[64] "mozilla/web-ext: A command line tool to help build, run, and test web extensions." [Online]. Available: https://github.com/mozilla/web-ext/

[65] "Trusted types." [Online]. Available: https://w3c.github.io/webappsec-trusted-types/dist/spec/

Appendix A
Availability

We have open sourced the extended compilers and safe API libraries described in this paper, whose code locations are displayed below.

Language	Component	GitHub Repository
JavaScript	Checker	Part of `google/closure-compiler`
	Safe API	Part of `google/closure-library`
TypeScript	Checker	`google/tsec`
	Safe API	`google/safevalues`

It should be noted that the open source version of these artifacts are revised to better fit the demands of external developers. In particular, both the TypeScript checker and the safe API library are customized to cooperate with Trusted Types [65], a new web security standard that originated from API hardening at Google.

Appendix B
XSS Sinks Inferred from DOM/HTML5 Specifications

A. Code Execution Sinks

- `eval()`.
- `setTimeout()` when the first argument is a string.
- `setInterval()` when the first argument is a string.
- Constructor of `Function`.
- `innerHTML` on `<script>`.
- `outerHTML` on `<script>`.
- `text` on `<script>`.
- `textContent` on `<script>`.
- `appendChild()` on `<script>`.

B. URL Navigation Sinks

- `href` on `<anchor>`.
- `href` on `<area>`.
- `action` on `<form>`.
- `formaction` on `<button>`.
- `formaction` on `<input>`.
- `location` on the `Document` interface.
- `location` on the `Window` interface.
- `href` on the `Location` interface.
- `assign()` on the `Location` interface.
- `replace()` on the `Location` interface.
- `open()` on the `Window` interface.

C. Loadable URL Sinks

- Dynamic `import()`.
- `importScripts()` on the `WorkerGlobalScope` interface.
- `src` on `<script>`.
- `src` on `<embed>`.
- Constructor of `Worker`.
- Constructor of `SharedWorker`.

D. HTML Sinks

- `write()` and `writeln()` on the `Document` interface.
- `srcdoc` on `<iframe>`.
- `innerHTML` on all elements except `<script>`.
- `outerHTML` on all elements except `<script>`.
- `insertAdjacentHTML()` on all elements.

E. CSS sinks

- `cssText` on the `CSSStyleDeclaration` interface.
- `cssText` on the `CSSStyleSheet` interface.
- `cssText` on the `CSSRule` interface.
- `innerText` on `<style>`.
- `textContent` on `<style>`.

F. Other Banned APIs

- `href` and `rel` on `<link>`. `href` can be either a URL navigation sink or a loadable URL sink, depending on the value of `rel`. See Section V-C.
- `href` on `<base>`. This attribute is not a sink that directly causes XSS. It is exceedingly security sensitive because it affects how browsers interpret other relative URLs in the page.
- `setAttribute` on all DOM elements. This API bypasses all other checks on DOM attributes.
- `URL.createObjectURL`. This API may cause XSS due to content sniffing.
- `DOMParser.parseFromString`. With the other XSS sinks monitored, we can trust in-memory DOM objects. This API compromises the trust by creating DOM objects directly from untrusted strings.