ScriptProtect: Mitigating Unsafe Third-Party JavaScript Practices

Marius Musch
TU Braunschweig

Marius Steffens
CISPA Helmholtz Center for Information Security

Sebastian Roth
CISPA Helmholtz Center for Information Security

Ben Stock
CISPA Helmholtz Center for Information Security

Martin Johns
TU Braunschweig

ABSTRACT

The direct client-side inclusion of cross-origin JavaScript resources in Web applications is a pervasive practice to consume third-party services and to utilize externally provided libraries. The downside of this practice is that such external code runs in the same context and with the same privileges as the first-party code. Thus, all potential security problems in the code directly affect the including site. To explore this problem, we present an empirical study which shows that more than 25% of all sites affected by Client-Side Cross-Site Scripting are only vulnerable due to a flaw in the included third-party code.

Motivated by this finding, we propose ScriptProtect, a non-intrusive transparent protective measure to address security issues introduced by external script resources. ScriptProtect automatically strips third-party code from the ability to conduct unsafe string-to-code conversions. Thus, it effectively removes the root-cause of Client-Side XSS without affecting first-party code in this respective. As ScriptProtect is realized through a lightweight JavaScript instrumentation, it does not require changes to the browser and only incurs a low runtime overhead of about 6%. We tested its compatibility on the Alexa Top 5,000 and found that 30% of these sites could benefit from ScriptProtect's protection today without changes to their application code.

CCS CONCEPTS

• **Security and privacy → Web application security**.

KEYWORDS

Client-Side XSS; Countermeasure; Web Security

ACM Reference Format:
Marius Musch, Marius Steffens, Sebastian Roth, Ben Stock, and Martin Johns. 2019. ScriptProtect: Mitigating Unsafe Third-Party JavaScript Practices. In *ACM Asia Conference on Computer and Communications Security (AsiaCCS '19), July 9–12, 2019, Auckland, New Zealand.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3321705.3329841

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
AsiaCCS '19, July 9–12, 2019, Auckland, New Zealand
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6752-3/19/07…$15.00
https://doi.org/10.1145/3321705.3329841

1 INTRODUCTION

Modern Web applications consist of functionality originating from many different parties. To allow for easy integration into existing sites, such third-party functionality is often added via client-side JavaScript code. This is enabled by the fact that sites may include external JavaScript from other origins, which is subsequently executed in the origin of the *including* site. Hence, such code runs with full privileges, e.g., can modify the DOM to add a frame pointing to an advertisement or observe user interaction for analytic purposes. Apart from ads and analytics, other use cases for third-party code include location services, social media integration, or support functionality.

At the same time, the insecure usage of attacker-controllable data in functions like `eval` or `document.write` may cause a DOM-based or Client-Side XSS flaw [17, 18]. Given the model of including scripts from other parties, Client-Side XSS flaws in third-party code result in vulnerabilities that directly affect the including Web site. The most recent study by Melicher et al. [18] has shown that this threat is still widespread in modern Web applications. Moreover, as an earlier work from Stock et al. [32] has shown, a significant fraction of the exploitable flaws is caused by third-party content.

The first-party code is under the control of the site's developer, who may employ secure coding practices to avoid vulnerabilities or use tools to find and patch them [23]. In contrast, the developer has no control over third-party code and cannot address vulnerabilities in included code. Specifically, if a third-party script is included, it has the power to add additional scripting content to the including site. Our results show that a significant fraction of the Client-Side XSS flaws is caused by such third-party code. Thus, vulnerabilities are introduced in otherwise secure Web sites merely by the inclusion of such a benign-but-buggy script.

Based on this finding, we propose a lightweight mitigation method dubbed ScriptProtect. At its core, ScriptProtect ensures that third-party code is unable to accidentally add unsafe markup into a document. By transparently instrumenting problematic APIs, ScriptProtect can simply be included by a developer as an external script resource before any other external script is loaded. When such a problematic API, e.g., `document.write` is invoked, ScriptProtect uses filters on the markup to be written to ensure that no script code can be added by third parties, making it *secure-by-default*. If needed, ScriptProtect also provides a reference to the original, unsafe API to the first party under a new name, making the *insecurity explicit*. While we envision ScriptProtect to be used when developing new applications, we also evaluate the feasibility

of retroactively applying it to existing applications (without any changes to that application). In particular, we investigate its compatibility with Web sites in the Alexa Top 5,000 and discuss these results as well as potential future improvements.

To sum up, we make the following contributions:

(1) We show that a significant amount of Client-Side XSS vulnerabilities are solely caused by third-party code (Section 2.3).
(2) Motivated by this finding, we present ScriptProtect, an easy-to-deploy solution to protect sites from vulnerabilities in third-party code (Section 3). The full implementation is available on Github[1].
(3) We analyze the compatibility of ScriptProtect and show that 30% of the Alexa Top 5,000 could be protected by the mere inclusion of our script, without breaking existing functionality (Section 4).

2 UNDERLYING PROBLEM

Cross-Site Scripting (also called *XSS*) is a class of code-injection vulnerabilities in the browser. Web applications run in a protected environment, such that only code from the same origin as the application can interact with it. Therefore, the goal of an XSS attacker is to execute arbitrary JavaScript code in the browser of his victim, in the context (or origin) of the vulnerable Web page. If successful, this allows him to conduct any action in the name of the victimized user, e.g., using the application as the user or retrieving secret information such as cookies.

From a conceptual standpoint, Cross-Site Scripting is caused when an unfiltered *data flow* occurs from an attacker-controlled *source* to a security-sensitive *sink*. While server-side XSS attacks have been known for a number of years, its client-side counterpart, i.e., Client-Side XSS, was first discussed in 2005 by Amit Klein [14] under the term *DOM-based XSS*. Rather than being caused by vulnerable server-side code, this class of vulnerability occurs if user-provided input is insecurely processed on the client side. In the case of such a Client-Side XSS, the source can be, e.g., the URL, whereas an example for a sink is `eval` or `document.write`. Both these APIs accept strings as parameters, which are subsequently parsed and executed as code (JavaScript and HTML, respectively). Therefore, passing unfiltered attacker-controllable input to such functions eventually leads to execution of the attacker-provided code.

2.1 Anatomy of a Client-Side XSS Flaw

To give an intuition of how such a flaw could look like, we consider the example shown in Figure 1. This snippet is meant to dynamically add an image to the DOM, sending both the query parameter as well as the hash (or fragment) of the *including* site to the advertisement company. This snippet suffers from a Client-Side XSS flaw, since both the query parameters and the fragment are simply concatenated with the HTML code — without any sanitization or encoding. All browsers handle automatic encoding of parts of the URL differently; we consider Microsoft's Edge, which automatically encodes the query in `location.search`, but not the URL fragment in `location.hash` [1]. Hence, an adversary can inject markup into the fragment of the URL to trigger the flaw. In this example, a specific payload could be `'><script>alert(1)</script>`, opening an alert box. Naturally, the alert box is merely a proof-of-concept and can be exchanged with arbitrary payloads.

[1] https://github.com/scriptprotect/scriptprotect

The first tool developed specifically to detect such flaws was *DOMinator* by Di Paola [7]. It leveraged taint tracking to follow a client-side flow of data from an attacker-controllable source to a dangerous sink, such as the aforementioned `document.write` or `eval`. This idea was picked up by Lekies et al. [17], who implemented byte-level taint tracking for the Chromium browser and used the discovered flows as input to an automated exploit generator. With this, they were able to automatically detect Client-Side Cross-Site Scripting flaws in approximately 10% of the Alexa Top 5,000 domains. In 2018, Melicher et al. [18] confirmed these findings with their own taint implementation, which they open-sourced.

2.2 Third-Party JavaScript Providers

Not all flaws in a website are necessarily caused by the first party, i.e., the Web site's developer. HTML's `script`-tag allows the direct reference and subsequent inclusion of remote JavaScript files (see Figure 2). As the inclusion of scripts is exempt from JavaScript's Same-Origin Policy [37], the code is downloaded by the browser and executed in the context of the including first-party Web document. Thus, it inherits the origin of the *hosting* document and runs in the same security context as the first-party code. This is helpful since it allows developers to, e.g., include libraries from a common URL (such as jQuery.org), which reduces the network load on their servers and increases the chances of the file already being cached on the client. Moreover, it allows for seamless updates of the included code, since changes to the third-party code are immediately "applied" to all including sites.

At the same time, this paradigm means that whenever third-party code carries an XSS vulnerability, any site that includes such code is susceptible to an XSS attack. Hence, a vulnerable data flow contained solely in third-party code causes an exploitable flaw in every site including that code resource, even in sites which are otherwise secure. Researchers have also shown that the number of domains from which third-party code is included is on the rise and, moreover, the complexity of the including JavaScript steadily rose as well over the past 20 years [22, 30]. Given this trend, it is not surprising that third-party code is a significant contributor to all Client-Side XSS vulnerabilities, especially with respect to outdated libraries [15, 18, 32].

```
document.write("<img src='http://ad.com/ad.jpg?query=" +
↪  location.search + "&hash=" + location.hash + "'>");
```

Figure 1: Example of a Client-Side XSS vulnerability

```
<head> <!-- HTML code delivered from https://a.com -->
    <script src="https://b.com/service.js"></script>
</head>
```

Figure 2: Remote JavaScript inclusion

2.3 Prevalence of Third-Party Caused Vulnerabilities

To investigate how common these third-party-caused flaws are in the wild, we used the existing taint tracking approaches [17, 18] to detect client-side flows in the Alexa Top 5,000. We combined the taint tracking with our own instrumented browser based on Chromium to detect third-party resources and to validate the exploitability of the previously collected flows. We crawled these up to a depth of two with a maximum of 1,000 pages per site, resulting in about 3.5 million visited pages.

In general, we found that almost 99% of the successfully crawled sites had at least one page that included scripting resources from third-party hosts, which leaves them susceptible to the possibly lacking security standards of the third-party as discussed by Nikiforakis et al. [22]. Using the aforementioned tainting methodology, we could verify exploitability of a Client-Side XSS on 351 sites of the Alexa Top 5,000 and an additional 122 framed sites which were integrated into the top websites but not part of the Top 5,000 themselves. Investigating the host of the script which initiated the vulnerable sink access shows that 129 sites are vulnerable due to scripts originating from third-party hosts. On 37 of these, we also found a vulnerable flow originating from the first-party code, leaving us with 92 sites, which are solely vulnerable due to third-party code. This means that with 92 out of 351, more than 25% of all vulnerable sites are exploitable through no fault of their own, other than the decision to include a third-party script.

Furthermore, we also investigated the cases in which a third-party contributes to the flaw by being part of the vulnerable call stack. We can see that this is the case on 173 of our vulnerable 351 sites. This, however, does not necessarily mean that the third-party can be blamed for the flow, e.g., jQuery is often included from a remote host and provides functionality (such as `.html`) which eventually calls sensitive sinks. Nonetheless, it shows that nearly every other vulnerability consists of deeply intertwined code parts which originate from multiple different parties.

This paints a grim picture for the security-aware developer who invests time in securing the Web applications against the threat of XSS, only then to notice that a third party introduced a vulnerability. Especially given that modern Web applications and their interconnectivity appear to grow even further [30], it is unreasonable to burden developers with banishing third-parties from their security perimeter. To that end, we propose ScriptProtect which functions as a lightweight drop-in solution to harden the Web application against benign but buggy third-parties.

3 SCRIPTPROTECT

The significant amount of vulnerabilities introduced by third-party code shows that solely hardening first-party code against XSS is often times insufficient. Instead, we would also want an environment that prevents other parties from inadvertently introducing new vulnerabilities. This way, we enable first-party developers to focus on the security and functionality of their own code. Coming back to the XSS example shown in Figure 1, we want to allow the third party to add benign content like images (without event handlers), but ensure they cannot add markup containing script code. Our proposed solution called ScriptProtect is distributed as a single JavaScript file, which the first party needs to include. ScriptProtect modifies the JavaScript environment in a such a way that the accidental introduction of a Client-Side XSS vulnerability through a third party is prevented. This lightweight approach is easy to deploy and does not rely on modifications to the browser, which means ScriptProtect can be used today already.

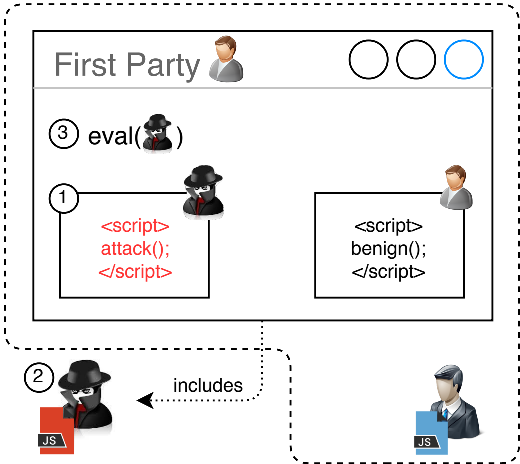

Figure 3: Considered Attacker Model

3.1 Threat Model

Over the past decade, there have been many attempts to allow the inclusion of third-party code without compromising the security of the including site itself [2, 12, 25, 33]. However, these papers assume a *malicious* third-party and hence require very strict isolation of the third-party code, which in turn tends to break many use cases like analytics and enrichment of pages in general.

In contrast, in the context of this paper, we consider the third-party scripts to be *non-malicious* but *vulnerable*. The overview of the potential attack scenarios are depicted in Figure 3. While the trusted third party is meant to operate within the security bounds of the first-party application, the attacker tries to attack in three distinct ways. First, he can try to inject malicious inline scripts (1). Second, when an HTML injection is possible, he can resort to including an externally hosted JavaScript file containing his payload (2). Finally, he can use the `eval` functionality to conduct a string-to-code transformation and gain code execution (3). With respect to the third party which also provides code to the application, this means that the initially executed third-party code has no intent of undermining our protection scheme. The attacker in this scenario is a malicious actor, who aims to exploit the Client-Side XSS problem, which was unwillingly introduced by the third-party script. As such, the attacker has to perform a string-to-code conversion while abusing an XSS flaw to introduce his new, malicious code into the origin. Hence, while a malicious third party could undermine our protection scheme, e.g., by getting a reference to an unprotected version of `document.write` from a newly created frame, the attacker in our scenario does not have that capability.

The honest third party, on the other hand, was trusted through the action of including a script from its host. As such, a malicious third-party has no need to circumvent our protection, e.g. of `document.write`, as it can already execute arbitrary JavaScript code without requiring additional scripts elements in the DOM. Therefore, our goal is to prevent third parties to *accidentally* introduce new code where writing passive markup (without scripts and event handlers) to the page would be sufficient. Consequently, attacks that involve compromising the third-party servers are out of scope for our protection.

3.2 Concept

On a conceptual level, ScriptProtect works as follows: All potentially dangerous browser APIs and properties (such as `innerHTML` or `document.write`), which could cause Client-Side XSS vulnerabilities, are instrumented at runtime. This is done through our protection mechanism which is included early in the `head` portion of the hosting HTML document. After the execution of ScriptProtect's code, all instrumented APIs are *secure-by-default*, meaning they cannot be used to add additional script content to the DOM. To allow trusted first-party code to still leverage the script-introducing functionality of these APIs, two strategies can be used:

New applications — Unsafe API variants. While the standard API remains unable to introduce new script content into the web document, ScriptProtect introduces a second version of the API (e.g., `document.unsafeWrite`) to be used explicitly by first-party code in cases in which *additional script code should be added* to the document. As these cases are in the minority and clearly marked through the explicit usage of the unsafe versions, it is straight forward to audit such occurrences during development to avoid vulnerabilities. Since our threat model considers a benign-but-buggy third party, the insecure variants of the APIs are not used by third-party code.

Existing applications — Stacktrace-based value treatment. In case a significant code base for the target application already exists, ScriptProtect offers the option to dynamically adapt the behavior of the APIs and DOM properties, depending on the calling party. In this case, the mechanism transparently checks the stack trace of the current execution thread to obtain the top-most execution context causing the execution chain, which then ended up in a potentially harmful operation. If the call originally was initiated by a trusted first-party functionality, the value is passed unaltered to the API/property. If the original call came from a third-party script, the value is automatically sanitized, so that no additional script content is added to the web document.

3.3 Dangerous APIs

The purpose of ScriptProtect is to ensure that vulnerabilities in third-party code cannot be leveraged for an XSS attack. Hence, our security policies are set such that creation of additional script content cannot occur, whereas other types of content injection (e.g., including iframes or images hosted by other domains) are outside of our threat model.

In essence, these considerations result in a policy which strictly *forbids third-party JavaScript code to conduct string-to-code conversions*, i.e., the introduction of additional executable JavaScript code that was derived or referenced from potentially untrustworthy data. For one, ScriptProtect prevents third-party code from creating additional `script` tags. Furthermore, a given third party is not allowed to introduce code via the inline event handlers of newly introduced HTML elements. The creation of an `iframe` with a `srcdoc` attribute or `javascript:` URI is prevented, as otherwise code execution on the origin of the first party can be achieved. Finally, ScriptProtect also strips third-party code from the ability to conduct direct string-to-code conversion via APIs like `setTimeout`.

There exist a variety of different APIs and DOM properties which have the ability to introduce new code through one of the means above. These APIs and properties can further be divided into different classes of functionality, types of access and accept different kinds of inputs. In the remainder of this section, we briefly describe the characteristics of each of the classes of functionality and conclude with a list of all dangerous APIs in Table 1.

Raw DOM content at rendering time. For one, JavaScript offers a set of APIs which add additional HTML code to the document directly during the initial rendering of the document. The code is seamlessly interpreted right after the script terminates and before subsequent HTML code is parsed/interpreted. The best known and most widely used API here is `document.write`.

Runtime creation of DOM content. The next relevant class are DOM APIs and properties that allow the alteration of the HTML content at runtime, i.e., after the initial rendering process has terminated. Unlike `document.write` this functionality is not provided as a global API. Instead, it is achieved through a set of properties and APIs which are directly attached to DOM elements. This way, the relative location of the new content is provided implicitly. The most used element in this class is the `innerHTML` property, which on assignment inserts new HTML subtree-structures as a DOM child of the hosting HTML element (see Figure 4). However, this class also includes all means to modify existing DOM elements, e.g. modifying the `src` attribute of a `script` tag.

Direct code conversion. Apart from the addition of HTML markup to the document, JavaScript code may also be executed directly. In particular, a set of APIs and DOM properties allow the direct conversion of string data into JavaScript code. The most used representative of this class is the function `eval`.

3.4 Instrumentation of HTML Sinks

ScriptProtect is implemented in the form of a JavaScript library that instruments dangerous APIs with HTML sinks in such a way that all values passed to them are sanitized, before the API's or property's DOM altering functionality is executed. In the context of this paper, the process of sanitizing HTML input is regarded as an orthogonal problem and we assume that a safe implementation exists. As the browser does not (yet) provide a native way to

```
var element = document.querySelector("#container");
element.innerHTML = "<div>Hello World!</div>";
```

Figure 4: Example usage of the `innerHTML` API

Table 1: APIs and properties that can introduce new JavaScript code. Asterisks denote non-standard APIs.

DOM modification at rendering	**Type**	**Sink**
document.write	Global API	HTML
document.writeln	Global API	HTML
DOM modification at runtime	**Type**	**Sink**
Element.innerHTML	Property	HTML
Element.outerHTML	Property	HTML
Element.setAttribute	Local API	all
Element.insertAdjacentHTML	Local API	HTML
HTMLScriptElement.src	Property	URI
HTMLScriptElement.text	Property	JS
HTMLScriptElement.textContent	Property	JS
HTMLScriptElement.innerText	Property	JS
HTMLIFrameElement.src	Property	URI
HTMLIFrameElement.srcdoc	Property	HTML
HTMLTag.on*EventName*	Property	JS
Range.createContextualFragment	Local API	HTML
Direct code conversion	**Type**	**Sink**
eval	Global API	JS
Function	Global API	JS
setTimeout	Global API	JS
setInterval	Global API	JS
setImmediate*	Global API	JS
execScript*	Global API	JS

sanitize JavaScript values for safe DOM inclusion, we leverage the established DOMPurify [6, 11] library for this purpose.

We achieve our protection by wrapping all functionality that could lead to code execution so that each call to an unsafe API is intercepted by our hook. Therefore, when deployed, ScriptProtect needs to be the first script that is included into the page to ensure all following code automatically uses the secured APIs. This instrumentation is completely transparent to the rest of the application: pre-existing code which intends to call the original API keeps functioning without requiring any code changes. Each of three types from Table 1 requires a different instrumentation strategy:

Global APIs. The reference to global APIs is readily available after document initialization. We first preserve a link to the original implementation for future usage with sanitized values (see Listing 5, line 4). Subsequently, we overwrite the global reference and replace it with a reference to a function which first executes the sanitizing step, before calling the original functionality to achieve a transparent instrumentation. Please refer to Listing 5 for a full example securing the `document.write` and `document.writeln` API.

Local APIs. In the case of the instrumentation of element-local APIs that are directly attached to DOM node, no single global reference to an API exists as each individual DOM element exposes the API to the calling code. In this case, we have to leverage JavaScript's prototype-based object oriented features. All DOM nodes are decedents of JavaScript's `Element` object class. Thus, via altering `Element`'s prototype, we are able to change the behavior of *all* DOM nodes transparently. Figure 6 shows this process. Specifically, we get a reference to the original `insertAdjacentHTML` method in line 2. Subsequently, we overwrite the method in the prototype and only invoke the original variant after sanitizing the HTML markup (passed as the second parameter to `insertAdjacentHTML`. For APIs that exist only for a subclass of DOM nodes, such as the `createContextualFragment` API of `range` elements, we instrument the respective, more specialized prototype.

```
(function() {
    let old = {};
    function wrap(name) {
        old[name] = document[name];
        document[name] = function() {
            arguments = sanitizeAll(arguments);
            old[name].call(document, ...arguments);
        };
    }
    wrap("write"); wrap("writeln");
})();
```

Figure 5: Transparent instrumentation of global APIs

```
(function() {
    let old = Element.prototype.insertAdjacentHTML;
    Element.prototype.insertAdjacentHTML = function() {
        if (arguments.length == 2) {
            arguments[1] = sanitize(arguments[1]);
        }
        return old.call(this, ...arguments);
    }
})();
```

Figure 6: Transparent instrumentation of local APIs

```
(function () {
    var old = Element.prototype.innerHTML;
    Object.defineProperty(Element.prototype, "innerHTML", {
        set: function (val) {
            val = sanitize(val);
            old.call(this, val);
        }
    });
})();
```

Figure 7: Transparent instrumentation of properties

Properties. DOM properties, on the other hand, cannot be instrumented directly. Instead, their setter property has to be replaced with the safe wrapper. As the DOM properties themselves are again attached to individual DOM nodes, we have to change the elements' prototype, similar to the element-local APIs. Figure 7 shows how we achieve this. `Object.defineProperty` allows us to overwrite the `set` property, i.e., the setter to be called when assigning a value to `innerHTML`. We then proceed to sanitize the input and invoke the original, stored setter.

3.5 Instrumentation of JS and URI Sinks

For APIs like `eval`, which directly result in code execution of the complete string, there exists no harmless subset of inputs. This is also true for the assignment of DOM properties that take JavaScript code as a parameter like `script.innerText`. Therefore, calls to these

```
(function () {
    let old = window.setTimeout;
    window.setTimeout = function() {
        //Allow only if there is no string-to-code conversion
        if (typeof arguments[0] != "string") {
            old.call(window, ...arguments);
        }
    };
})();
```

Figure 8: Instrumentation of `setTimeout`

APIs with a JS sink do not involve a sanitization step, as shown in Figure 8. Instead, the call is completely blocked unless the introduction of new code was allowed through one of the two mechanisms described in the following sections.

The same is also mostly true for the assignment of a URI, e.g. `script.src`, with the only exception that including further scripts from the host of third-party could be allowed as this domain is already trusted. However, allowing the inclusion of same-site scripts has some subtle drawbacks: For one, if the third-party is a CDN, then the attacker could abuse this to include older versions of libraries hosted on that CDN, allowing potential script gadget attacks [16]. Moreover, these might contain publicly disclosed vulnerabilities, which then could re-enable old attacks on patched and up-to-date Web sites. Furthermore, it is notoriously difficult to correctly identify the effective top-level domain. A script hosted on `*.amazonaws.com` should not trust other hosts on `amazonaws.com`, as they are all part of Amazon's public cloud infrastructure and anyone can obtain a subdomain for this domain to host malicious scripts. This could be solved by using the public suffix list [8], which includes a list of both official ICANN suffixes, e.g. `co.jp`, and private suffixes like `cloudfront.net` or those for Amazon's AWS. With over 20,000 entries the list is rather large and weighs about 200KB, which is about 10 times the size of ScriptProtect itself and would have a negative impact on the loading time. While there are public discussions of Google engineers about exposing this feature to JavaScript[2], this has not yet happened. For these reasons, we decided to block the assignment of URIs in general and treat these sinks in the same way as the JS sinks.

3.6 Unsafe API Variants

As previously discussed, ScriptProtect's measures create API variants that are *secure by default*. This means that if no script-reenabling steps are taken it is impossible to introduce additional JavaScript content into the web document. However, first-party code might require this capability occasionally. The safest method of enabling this functional requirement is to introduce potentially unsafe API/property variants. In this case, the majority of the application code uses the standard – now safe – APIs/properties. Only in selected cases, in which the introduction of further script code is explicitly intended, a second, newly introduced variant of the API/property is called which allows the potentially unsafe action. The occurrence of such cases is most likely seldom and can be audited thoroughly, as the *insecurity is now explicit*.

[2]https://twitter.com/slekies/status/1064213702528954368

```
(function () {
  var oldSet = Object.getOwnPropertyDescriptor(Element.prototype,
  ↪  "innerHTML").set;
  Object.defineProperty(Element.prototype, "unsafeInnerHTML", {
    set: oldSet
  });
})();
```

Figure 9: Introduction of an unsafe innerHTML property variant

Third-party scripts won't use the unsafe variants, as they are designed and implemented for standard browser functionality, and thus, are completely safe. If third parties are aware of the unsafe variant, they could obviously simply call that. However, as discussed before, we assume a benign-but-buggy third party, and therefore this threat is out of scope for our attacker model (cf. Section 3). We expect no intentional circumvention of our protection from a party that already has achieved code execution. Implementation-wise, for global APIs we just attach the original, native function to a global object outside of our instrumentation closures under a new name like `document.unsafeWrite`. In case of local DOM properties, additional properties are added to the respective element's prototypes (see Figure 9 for how an element's unsafe innerHTML may be exposed).

Using the unsafe API variants (if really necessary) is the recommended way in case a new Web application is created from scratch. However, as this change merely renames the original functionality, this can also be used for existing applications in combination with a rewrite proxy or static analysis tool. While giving the first party full access to the dangerous APIs by default is not ideal, we see this option useful for a transitional phase: access for the third party is immediately blocked while the first-party code is rewritten to use the unsafe variant by default. Then gradually all usages of the dangerous APIs need to reviewed and, depending on each individual case, changed to use the safe variant, if possible.

3.7 Dynamic Access Control

In case a significant code base already exists that cannot be adjusted to use the unsafe API variants, e.g. legacy code that includes minified components, ScriptProtect supports an alternative mechanism that is based on dynamic access control. In this scenario, whether or not a call to a problematic API is allowed, depends on the party that induced that call in the first place. Calls that originate from the first party are then routed to the original, unaltered API while all calls induced by a third party will use the safe, instrumented APIs and properties.

By inspecting the current execution thread through the `Error` object, we can obtain the stack trace from within JavaScript code at runtime without requiring an external debugger. We then proceed to extract the URL of script at the top of this call stack, as shown in Figure 10. The script at the top of the stack trace represents the initiator of the actions that lead to the call in the first place. If the hostname of the script's URL matches the first party the call is allowed without modification. Otherwise, the function argument is subjected to value sanitization or blocked, depending on the sink.

```
function isAllowed() {
    //Extract all URLs from the stack trace
    var regex = /(https?:\/\/.+?):\d+:\d+/g;
    var urls = (new Error).stack.match(regex);

    if (urls && urls.length > 0) {
        //Use last entry and extract its hostname from URL
        var topCaller = getHost(urls[urls.length - 1]);
        return topCaller == location.hostname;
    }
    return true;
}
```

Figure 10: Code snippet showing how stack trace is parsed and the top caller extracted, simplified for brevity

4 EVALUATION

Generally speaking, ScriptProtect is designed as a measure for security-conscious operators of Web applications, who want to prevent the introduction of vulnerabilities by third parties. As access to the dangerous APIs and properties from Section 3.3 is common, we expect that ScriptProtect cannot be immediately picked up by all existing sites. Therefore, after reporting on ScriptProtect's performance impact we evaluate its compatibility with the sites in the Alexa Top 5,000.

4.1 Runtime Performance

As ScriptProtect is an always-on mechanism that adds additional access-control checks to important APIs we expect it to have some performance impact during runtime. To evaluate this we randomly sampled 50 different pages from a set of compatible sites with the condition that these specific pages included at least one third-party script. We used a 2015 Macbook Pro with a 2,2 GHz Intel i7 processor, 16 GB RAM, running Mac OS X 10.14.1 and Google Chrome 70.0.3538 for these experiments.

After an initial visit to populate the cache each page was visited 20 times: 10 times completely unmodified to establish a baseline and 10 times with ScriptProtect enabled. After all these visits we took the median for each of the two configurations to be more resistant against outliers caused by the network or remote server. ScriptProtect was locally injected by us, but we argue that after the first page load it would be loaded from cache anyway. The final overall results were obtained by averaging over the medians of all 50 pages and are shown in Table 2.

The minified version of ScriptProtect used in the evaluation consists of 19 KB and increases the load time by about 6%. However, our proof-of-concept relies on DOMPurify to sanitize inputs and with 14 KB most of the size is from this library, while ScriptProtect itself only weighs about 5 KB. Consequently, most of the increase in load time is caused by the parsing and initialization of the script itself. Fortunately, in the current standardization of trusted types for the DOM [9] the introduction of a browser-native sanitizer is on the roadmap [10]. Thus, as soon as this functionality is available directly in the browser, ScriptProtect also loses DOMPurify's network traffic and parsing time. In addition, it is to be expected that a native sanitizer vastly outperforms any JavaScript solution, further adding performance improvements.

Table 2: Performance measurements of ScriptProtect, showing the time in milliseconds until the load event fired.

	Avg. Median	Std. deviation	Slowdown
Baseline	1280	1278	-
ScriptProtect	1360	1377	1.06

4.2 Compatibility

Ideally, ScriptProtect is used when creating new applications. Then, the unsafe API variants clearly indicate which code parts could lead to vulnerabilities, aiding both manual and automated security analysis. Still, existing applications can also profit from the protection today without requiring any code changes by using the trace-based inspection of calls. A site is compatible as long as all included third-party scripts do not add additional JavaScript (through external scripts, inline scripts, or event handlers) on their own during normal operation. In that case, ScriptProtect would not block any action of the third party during a normal visit without an attempted attack — the site could add our protection without breaking existing functionality.

4.2.1 Data collection. To analyze the compatibility of ScriptProtect in the Alexa Top 5,000 we let our instrumented browsers crawl these Web applications. However, we found that the Alexa list contains 103 google.tld domains and a total of 82 subdomains of tmall.com. To gravitate our analysis to a more diverse set of Web applications we opted to skip those entries in the list for which we either already had a site included which has the same eTLD+1 or the same second-level domain. Additionally, we remove any entry for which we are unable to connect to the Web application according to the following pattern `http://ENTRY`. After this preparation step, we arrive at a new list of 5,000 sites to be crawled. Our crawlers follow each same-site link up to depth 2 with a maximum of 1,000 unique links per site. This allows us to analyze the Web applications in more depth than the previous approaches [17, 18] and leaves us with around 3.5 Million pages on 4528 different sites.

On the remaining 472 sites, however, we were unable to visit more than one link successfully. Investigating these cases reveals that for 106 sites we were unable to connect to the main site via HTTP due to the connection being preemptively terminated (e.g., connection resets, unresolvable hostnames) or the site needing more than 30 seconds to load which triggers a timeout in our infrastructure in order to prevent infinite loading sites. Randomly sampling 5% of the other 366 reveals that on 9 sites our crawlers were blocked visiting the site and were instead served a static site indicating the block, with 1 location and 3 IP-based blocks. Further 6 sites only served static content (CDN main sites, domain selling sites) and 3 sites would have required to circumvent interstitial JavaScript dialogs(e.g., GDPR interstitials). Of these remaining 4528 sites with more than one successfully visited page, only 65 do not include any third-party script on their Web presence. Thus, for our further analyses in the rest of this section, we focus only on these 4463 sites which could theoretically benefit from ScriptProtect.

4.2.2 Sink usage. By artificially injecting the protection via our instrumented browser and visiting these sites (and all their subpages in the set of the initial 3.5 Million pages), we observe that

Table 3: Number of sites for which a third-party used a sink in order to add new code. See Table 1 for a mapping of APIs and properties to sinks.

Description	# of sites
Successfully crawled	4528 (—)
With third-party scripts	4463 (100%)
Third-party adds code via URI sinks	4180 (94%)
Third-party adds code via HTML sinks	3122 (70%)
Third-party adds code via JS sinks[3]	1562 (35%)

on the vast majority of sites, a third party dynamically adds new code at least once. This is due to the fact that constructing and assigning script URLs at runtime is extremely popular (e.g., in advertisements) and used by third parties on about 94% of sites. On the other hand, only 70% of these sites include a third party, which uses APIs and properties with an HTML sink like `innerHTML` *to insert new JavaScript code*, triggering the sanitization step to block that new code. Furthermore, only 35% sites have a third-party which use methods with a direct JavaScript sink like `setTimeout`[3]. Our results on the usage of the different sinks are summarized in Table 3.

Coming back to the 129 sites from Section 2.3, which are vulnerable due to flaws in third-party code, we investigated which sink was actually responsible for the vulnerability in the first place. We find that 122 of the 129 sites were exploitable due to the injection of HTML markup, 4 due to an injection into `eval`, 2 sites had an injection into `script.src`, and on another 2 sites the attacker can hijack the content of a `script.text` attribute[4]. This shows that the APIs and properties with a JS or URI sink are preventing compatibility with a significant amount of existing sites, while the real-world vulnerabilities are only rarely caused by them. Intuitively this makes sense, as the direct assignment of a `script.text` or call to `eval` makes it very obvious to the developer that new code is created. On the other hand, for example, a call to `innerHTML` to adjust the content of a `div` tag does not make its security implications completely obvious, again highlighting the need to make the insecurity explicit. Therefore, to achieve backward-compatibility with existing applications, we activate our instrumentation only for the HTML sinks, which still mitigates most risks.

4.2.3 Mitigation Effectiveness. To verify that this backward-compatible version of ScriptProtect indeed provides the targeted protection, we artificially added our `scriptprotect.js` to the top of the `head` of all pages in the set of the 129 sites with vulnerable third-parties. Subsequently, we checked whether the proof-of-concept exploits discovered by the taint engine were blocked by our protection mechanism. Due to the design choice to exclude them, the 8 sites with non-HTML sinks could not be protected. However, another 13 also continued to be vulnerable, as their third-party scripts were vulnerable to an *HTML injection*, but then proceeded to insert our payload without using one of the protected *HTML sinks*. Manual investigation of these 13 sites showed that this is due to the fact that, when using certain libraries, the line between the different sinks begins to blur. For example, jQuery's `.html` function is a more convenient version of `innerHTML`, that internally uses `script.text` (or `eval` in older versions) to execute inline scripts, which is not possible using the standard `innerHTML` function. As jQuery is widely used we extended ScriptProtect to also wrap and protect all of jQuery's HTML sinks. Adjusting our protection to correctly function with all other popular libraries, however, is out of scope for this work, but would be straightforward on a case-by-case basis. After adding the additional protection of jQuery another 6 sites were protected, leaving only 15 sites vulnerable despite ScriptProtect's presence.

Overall, this backward-compatible version of ScriptProtect prevents the exploitation of the discovered third-party vulnerabilities on 114 of the 129 sites. While a more complete protection certainly would be desirable, a tradeoff between security and compatibility needs to be made. With our approach of only instrumenting the HTML sinks, ScriptProtect can be used on 1341 (30%) of the 4463 sites with third-party code *without any code changes* while still preventing almost 90% of all third-party caused vulnerabilities.

To understand if other approaches could mitigate the risk of an exploitable flaw in a similar fashion, we also checked if the sites in question could deploy a strict Content Security Policy (CSP). In doing so, we found that *all* sites with a vulnerable third-party script also made use of inline scripts or event handlers. Therefore, these sites could not easily deploy a secure policy without modifications; at the very least, securing inline scripts would require to use nonces. For event handlers, though, while there is a discussion about allowing these via `unsafe-hashed-attributes`[5], there is currently no solution other than using the `unsafe-inline` keyword. This would entirely undermine CSP's protection capabilities. Hence, the only way to use CSP would be to rewrite large parts of the application [38]. We see this as a further evidence that ScriptProtect indeed fills the much needed gap of an easy-to-deploy mechanism that helps to prevent Client-Side XSS attacks.

5 DISCUSSION

In the following, we describe the limitations of both our approach and the conducted evaluation and discuss the resulting impact on our results. We also shortly introduce other security-relevant areas that would likely benefit from a ScriptProtect-like solution in the future. We conclude this section by comparing our approach to the Trusted Types proposal, which also intends to get rid of Client-Side XSS problems.

5.1 Limitations

Our approach comes along with a few limitations, which we outline in the following.

5.1.1 Wrapping of eval. Wrapping all JavaScript sinks suffers from a drawback: instrumenting the `eval` statement is problematic, as it is a language construct rather than a simple function in JavaScript. This results in potential scoping-related side effects caused by the instrumentation. We consider the example code in Figure 11. In an unmodified environment, the JavaScript engine will execute the

[3] This excludes the usage of eval, as it cannot be safely wrapped without potentionally breaking other functionality. We discuss this further in Section 5.1

[4] One site had both an injection into an HTML context and into eval, resulting in 130 sinks on 129 sites

[5] https://www.chromestatus.com/feature/5867082285580288

```
function x() {
  // Function scope
  var a = 2;
  eval("a += 1");
  // Some more code ...
}
```

Figure 11: Example to highlight issues with `eval` and the invoking function's local scope

`eval` statement in line 4 in the local scope of the function `x`. This means that starting from line 5, the local variable `a`'s value will be 3. If, however, we now overwrite `eval` to be a function we define ourselves, this function does not contain a local variable `a` in its scope. Hence, evaluating the statement in line 4 will result in a *ReferenceError: a is not defined*. This does not affect the backward-compatibility of ScriptProtect, as the JavaScript sinks are not instrumented for existing applications (see Section 4.2). For the unsafe API variants, the first party needs to forgo using `eval` at all, so that in can safely be disabled for the third parties, without breaking first-party code.

5.1.2 *Discovery of Incompatibilities.*

Naturally, a limitation of our approach to determine the compatibility of ScriptProtect with existing sites, is the coverage we achieved during our crawl. Regarding page coverage, we are both limited in terms of which pages we can discover and how many we can visit. Specifically, we could only visit those URLs which were linked to from another page we visited and our crawler does not have the capability to login to any of the applications (if such a login exists). Furthermore, we limited the crawl to a depth of 2 and a maximum of 1000 pages per Web application, to finish in a reasonable time frame. Regarding code coverage, the crawler does not interact with the page beyond the initial load request. Therefore, we cannot determine if a dangerous API or property would be invoked after some interaction by the user. As a result, the compatibility of ScriptProtect is likely *lower* than reported.

5.1.3 *Attribution of Parties.*

Both the inspection of stack traces and our compatibility evaluation uses the hostname to detect third-party scripts. This notion, however, does not necessarily match the facts on the Web, where major players like Google have specific domains for each type of service. An example of this is the combination of Google's DoubleClick advertisement service and the domain `googlesyndication.com`. The syndication domain is merely used to host scripting and ad resources, yet belongs to the same party. Similarly, Google also owns `gstatic.com`, which is used to store content that is supposed to be static. Hence, when a script from `google.com` includes additional script resources from `gstatic.com`, these look like third-party scripts from the hostname, but in fact are from the same entity. As a result, the compatibility of ScriptProtect is likely *higher* than reported, as manual exceptions could be added to the script.

5.1.4 *Completeness of Stack Traces.*

Unlike the stack traces in an external debugger like Chrome's DevTools, the traces obtained through `Error().stack` from within the JavaScript environment are somewhat limited. For example, the newly introduced keyword `await` will suspend execution until another asynchronous task has finished. However, when the execution is then resumed from the microtask queue (basically an event loop), we can no longer access the stack trace of calls that happened before the `await`. Additionally, we have no way of knowing that the stack trace was truncated at this point. Therefore, if the first-party code calls directly into third-party code and then the trace is truncated, we might unwarrantedly block a call to a dangerous API. While the other way round is also possible, we assume that direct calls from generic third-party code into application-specific first-party code are less likely. However, browser vendors are aware of this problem and are working on possible improvements. As time of writing, the so-called "zero-cost async stack traces" are already available in the alpha version of Chrome 72, albeit still hidden behind a flag [19]. As a result, the compatibility of ScriptProtect is likely *higher* than reported, as soon as more accurate traces become available.

Note that the last three limitations only affect the backward-compatible version of ScriptProtect, as the introduction of unsafe API variants does not rely on the stack trace.

5.2 Future Work

In addition to ScriptProtect's focus on APIs and DOM properties that enable Client-Side XSS problems, other classes of JavaScript APIs could also be considered.

5.2.1 *Cross-domain Communication.*

The `postMessage` API allows to communicate within the Web browser and across the origin boundaries set by the Same-Origin Policy. Unsafe usage of this API could lead to the leakage of confidential information. Thus, to prevent third-party scripts from accidentally leaking information to untrusted sites, the proposed mechanism can be used to restrict the API usage to trusted first-party code.

5.2.2 *Client-Side Persistence.*

Several browser APIs including `localStorage` and `IndexedDB` allow the persistent storage of information in the browser. Depending on the first-party functionality, this storage might contain sensitive data. Also, allowing third-parties to store information into the first-party's origin bound storage might lead to pollution of the application's data and might lead to second order injection vulnerabilities as recently shown [29]. Hence, depending on the first-party usage of these persistence methods, restricting third-party scripts in this respect would increase the application's security robustness.

5.2.3 *Audio and Video Communication.*

With modern APIs, such as `MediaDevices.getUserMedia`, the browser has access to the user's camera and microphone. Permission to access these devices is obtained in the context of the web origin of the hosting site, i.e., under the context of the first-party application provider, but all third-party scripts are executed in the context of the first-party web origin. This means, if the user trusts the first-party to access his A/V devices, the third-party code is implicitly granted access as well. Thus, using ScriptProtect's technique, third-party scripts can be denied access to the A/V resources, without disrupting legitimate first-party functionality.

5.3 Comparison to Trusted Types

Trusted Types [9] is a recent proposal promising to eradicate Client-Side XSS by making unsafe DOM interaction explicit. In its current form, it requires that any string which is passed to DOM manipulating APIs needs to correspond to a "safe" type (e.g., `TrustedHTML` for `innerHTML`); otherwise, an exception is raised. Generating such a safe type requires interaction with a `Policy` object, which ideally takes care of a context-aware sanitization. An example of policy generation and subsequent sink access can be found in Figure 12. To the best of our knowledge the currently available specification for Trusted Types has yet to be finalized, thus could still undergo API level changes.

```
const myPolicy = TrustedTypes.createPolicy(name, {
    createHTML: (s) => { return sanitizeHTML(s) }
})
let unsafeString = /*potentially unsafe string*/;
let safeString = myPolicy.createHTML(unsafeString);

document.body.innerHTML = unsafeString; // leads to an exception
document.body.innerHTML = safeString;
```

Figure 12: Example of using Trusted Types with the innerHTML sink

While Trusted Types are undoubtedly promising, they exhibit similar shortcomings as CSP. The complete codebase of the Web application would need to be rewritten to conform with the changed API of DOM access, which includes code originating from third-parties. However contrary to CSP, which allows developers to add hosts required by the third party to the policy, Trusted Types mandates changes in the codebase of the third party, over which the first-party developer has no control. Especially, the lack of deployment of CSP [30] and the insecurity of most policies [5, 38] paints a grim picture for the at least equally challenging to deploy mechanism of Trusted Types. Therefore, we designed ScriptProtect in the most backward-compatible way possible, while providing strong protection against the most common attack vectors. ScriptProtect lifts the burden from the developer to design and enforce appropriate policies while still being reasonably compatible with modern Web applications.

If we compare the core concepts of Trusted Types and ScriptProtect, the former introduces a capability-based access control mechanism while the latter rather resembles a reference monitor. The capability in the case of Trusted Types is the Policy object. Access to this object allows the generation of the safe types, this capability cannot be reasonably contained in our setting of benign-but-buggy third parties, if the developer wants to, e.g. allow one party to interact with a lax policy whereas he only provides a strict policy to other parties. The approach of ScriptProtect, while not part of our core proposal, allows resolving exactly this scenario by, e.g., introducing an additional ACL mechanism into the decision routine of the instrumented API.

Nonetheless, both concepts try to reach the same goal of banishing Client-Side XSS from modern Web applications and can potentially benefit from each other. Trusted Types could incorporate the Dynamic Access Control of ScriptProtect to mitigate the problems of the new capability system. At the same time, ScriptProtect's general setup of API instrumentation allows for a rather simple introduction of Trusted Types into existing applications. Furthermore, we hope that our empirical results provide insights into the threat landscape of Client-Side XSS in *real* Web applications, which could steer the future development of Trusted Types. In particular to find an appropriate trade-off between security and usability while not repeating previous mistakes.

6 RELATED WORK

In the following, we discuss how our work relates to previous papers in the area of Client-Side Cross-Site Scripting as well as defenses against different types of Cross-Site Scripting and previous work on the secure inclusion of untrusted code.

6.1 Client-Side XSS

The notion of Cross-Site Scripting caused by client-side code was first identified by Klein [14]. In 2013, Son and Shmatikov [28] investigated the insecure usage of postMessages and found that in many cases, flows from postMessages to sinks like `eval` lead to exploitable Client-Side XSS flaws. Later that year, Lekies et al. [17] developed an automated system to find such flaws at scale, showing that 9.6% of the Top 10,000 sites had at least one vulnerability. Based on their methodology, the same group analyzed the nature of Client-Side XSS flaws [32] based on a set of 1,273 real-world vulnerabilities, showing that 273 were solely caused by vulnerable third-party code. The general problem of outdated and vulnerable libraries was highlighted in 2017 by Lauinger et al. [15], showing that up to 37% of the Top 75,000 sites used at least one outdated library. This pattern was also observed by Stock et al. [30], who analyzed the evolution of client-side security over a course of 20 years using the Internet Archive. Moreover, the authors also showed the trend of Client-Side Cross-Site Scripting flaws, finding that since 2004, more than 8% of the 500 most important sites contained at least one such flaw. More broadly, Saxena et al. [26] proposed FLAX to systematically find client-side validation flaws, e.g., unfiltered flows to a cookie, which could enable an attacker to conduct a session fixation attack.

6.2 XSS Defenses and Mitigation

Given that XSS has been around for almost 20 years, a number of researchers have proposed defenses and mitigations against these attacks. Early research focused on deploying such tools on the server, such as Vogt et al. [36] or Bisht and Venkatakrishnan [4]. Later, Ter Louw and Venkatakrishnan [34] proposed BluePrint, a tool enabling Web sites to provide a specification of the expected DOM structure; this way, any anomalous script content could be easily identified and removed. This approach, however, requires changes to the browser itself, which our approach does not.

In 2010, Bates et al. [3] analyzed the security of existing browser-based XSS detection. In doing so, they found a number of flaws in Internet Explorer's filter, which would even allow for XSS in error-free Web sites. Based on their insights, they proposed a new concept for an XSS filter, dubbed XSSAuditor. This filter is nowadays deployed in all Webkit-based browsers, such as Google's Chrome. However, as Stock et al. [31] showed in their work, design choices in the XSSAuditor make it susceptible to bypasses: in 2014, 73%

of the sites with vulnerabilities carried at least one flaw for which the Auditor could be bypassed. Pelizzi and Sekar [24] proposed an improvement variant with more aggressive filtering, naturally accompanied by an increased chance of false positives. Hence, the improved changes were not applied to the original Auditor.

In the area of defenses against Cross-Site Scripting, Stock et al. [31] proposed to use taint tracking to stop code injections. They extended their taint engine to forward taint into the JavaScript parser and enforcing policies to ensure that user-provided data could only be interpreted as literals and not lead to code execution. While their approach protects against all types of Client-Side Cross-Site Scripting, it requires immense changes to the code base of the browser and causes an overhead between 7% and 17%. Consequently, this has not been implemented into browsers as of now.

6.3 Securing Third-party Code

Over the years there have been many attempts to allow the inclusion of third-party code without compromising the security of the including site itself. In 2007, Jim et al. [13] proposed BEEP, a browser-enforced embedded policy that controls which scripts are allowed to run. Similar to today's CSP, it allows whitelisting of specific scripts by including their SHA1 hash in the policy. For a more fine-grained approach, Meyerovich and Livshits [20] designed ConScript, which allows for specific security policies that are added to each script tag. Possible policies could, for example, forbid the use of dynamic scripts or calls to specific functions like `postMessage`. In a similar fashion, Van Acker et al. [35] created WebJail in 2011. While certainly powerful, these approaches require drastic modifications of the browser to enforce the policy and, without adoption by popular vendors, have not found widespread use. Finally, Snyder et al. [27] built a browsing extension that works in a similar fashion to our hooking approach. In particular, this enables a user to selectively disable DOM features which are not needed by a site. In contrast, however, in our work we do not require a user with an extension or a study what a given site requires to function correctly. More importantly, though, our approach allows the continued usage of all DOM APIs, yet securing access to them.

In a different kind of approach, Miller et al. [21] created a subset of JavaScript called Caja, an object-capability language which isolates objects from the outside world. However, this means that all untrusted code must be written in Caja. Following the same concept, Agten et al. [2] proposed JSand in 2012, a system that isolates third-party scripts from each other and the DOM through the use of an object-capability model. In the same year, Ingram and Walfish [12] presented Treehouse, a JavaScript sandbox based on web workers. While these and similar approaches can be implemented without modifications to the browser, they require changes to the untrusted code. Hence, adoption of this type of defense is inhibited by the lack of support from the third-parties like advertisement networks.

Finally, there are defensive mechanisms which are implemented as transparent wrappers, so that neither the browser nor existing code needs to be modified. In 2009, Phung et al. [25] published their work on self-protecting JavaScript, in which they intercept security relevant events by monitoring the methods and fields of built-in objects. One year later, Ter Louw et al. [33] proposed AdJail, an isolation framework specifically designed for advertisements. The isolation is achieved by running the code in a so-called shadow page and by providing a controlled interface for interaction with the real page. Yet, all these papers assume a malicious third-party in their attacker model and hence require very strict isolation of the third-party code. This, in turn, tends to break many use cases like analytics and enrichment of pages in general. In contrast, our concept is much more lightweight and simpler to integrate. In particular, there is no need to tamperproof our hooked functions, as we block all untrusted code *before* it is executed.

Still, there also has been some work on securing benign-but-buggy third parties, confirming the relevance of our threat model: In 2015, Weissbacher et al. [39] presented their system ZigZag, which transparently instruments JavaScript code to perform anomaly detection during runtime. After an initial learning phase, the generated models are used to harden the client-side code.

7 CONCLUSION

With the increased reliance on client-side code to enable a more interactive Web, the risk of Client-Side XSS has also risen. At the same time, Web site operators increasingly rely on third parties to provide functionality. Notably, this means that whenever a third-party code has a vulnerability, a site including said third-party code becomes vulnerable. In our study of the Top 5,000 Alexa domains we found that of the 351 sites with such a Client-Side Cross-Site flaw, 129 sites were vulnerable to due insecure third-party code. Additionally, in 92 of these cases, *only third-party code* was to fault for the discovered vulnerability. Such vulnerability instances are especially problematic for security-conscious Web site operators, as they have no control over the externally included JavaScript and no influence over the secure development lifecycle of the third-party script provider.

Motivated by this observation, we designed ScriptProtect — a lightweight and robust countermeasure that allows operators to protect their Web sites against Client-Side XSS that was introduced by benign-but-buggy third-party code. ScriptProtect is an effective and easy to deploy tool that is flexible enough to be adapted to a site's specific functional needs and even works out of the box with 30% of the currently existing Web code. To achieve the backward compatibility, we leveraged code-provenance based access control, i.e., the assignment of privileges according to the initial origin of the code. Therefore, ScriptProtect is significantly more permissive than earlier sandboxing approaches, while still robust enough to reliably mitigate the attacks in focus.

Looking at the bigger picture, this paper's empirical results once more highlight the limitations of the Web's current Same-Origin Policy. The all-or-nothing, document-level approach of the policy forces Web site operators to either fully isolate third-party services or to hand them complete and unmitigated control over all properties of the application's client side. While being over-permissive with respect to the script inclusion process, the SOP is also too restrictive with respect to handling multiple domains that are under the control of the same organization. Maybe, after more than 25 years of WWW, it is time to revisit this fundamental building block of the Web and adapt it to accommodate the security needs of today's applications.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable feedback. The authors gratefully acknowledge funding by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972 and funding from the state of Lower Saxony under the project Mobilise. Furthermore, this work was partially supported by the German Federal Ministry of Education and Research (BMBF) through funding for the Center for IT-Security, Privacy and Accountability (CISPA) (FKZ: 16KIS0345).

REFERENCES

[1] 2017. Percent-encode additional characters in URL's "fragment state". . https://developer.microsoft.com/en-us/microsoft-edge/platform/issues/14951215/

[2] Pieter Agten, Steven Van Acker, Yoran Brondsema, Phu H Phung, Lieven Desmet, and Frank Piessens. 2012. JSand: complete client-side sandboxing of third-party JavaScript without browser modifications. In *ACSAC*.

[3] Daniel Bates, Adam Barth, and Collin Jackson. 2010. Regular expressions considered harmful in client-side XSS filters. In *WWW*.

[4] Prithvi Bisht and VN Venkatakrishnan. 2008. XSS-GUARD: precise dynamic prevention of cross-site scripting attacks. In *DIMVA*.

[5] Stefano Calzavara, Alvise Rabitti, and Michele Bugliesi. 2016. Content security problems?: Evaluating the effectiveness of content security policy in the wild. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1365–1375.

[6] cure53. 2018. DOMPurify Github Repository. https://github.com/cure53/DOMPurify

[7] Stefano Di Paola. 2012. DominatorPro: Securing Next Generation of Web Applications. https://dominator.mindedsecurity.com/.

[8] Mozilla Foundation. 2019. Public Suffix List. https://publicsuffix.org/

[9] Web Incubator Community Group. 2017. Explainer: Trusted Types for DOM Manipulation. https://github.com/WICG/trusted-types.

[10] Web Incubator Community Group. 2017. Support application-specific sanitizers / type builders. https://github.com/WICG/trusted-types/issues/32.

[11] Mario Heiderich, Christopher Späth, and Jörg Schwenk. 2017. DOMPurify: Client-Side Protection Against XSS and Markup Injection. In *ESORICS*.

[12] Lon Ingram and Michael Walfish. 2012. Treehouse: Javascript Sandboxes to Help Web Developers Help Themselves.. In *USENIX ATC*.

[13] Trevor Jim, Nikhil Swamy, and Michael Hicks. 2007. Defeating script injection attacks with browser-enforced embedded policies. In *WWW*.

[14] Amit Klein. 2005. DOM based cross site scripting or XSS of the third kind. *Web Application Security Consortium, Articles* (2005).

[15] Tobias Lauinger, Abdelberi Chaabane, Sajjad Arshad, William Robertson, Christo Wilson, and Engin Kirda. 2017. Thou Shalt Not Depend on Me: Analysing the Use of Outdated JavaScript Libraries on the Web. In *NDSS*.

[16] Sebastian Lekies, Krzysztof Kotowicz, Samuel Groß, Eduardo A Vela Nava, and Martin Johns. 2017. Code-Reuse Attacks for the Web: Breaking Cross-Site Scripting Mitigations via Script Gadgets. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1709–1723.

[17] Sebastian Lekies, Ben Stock, and Martin Johns. 2013. 25 million flows later: large-scale detection of DOM-based XSS. In *CCS*.

[18] William Melicher, Anupam Das, Mashmood Sharif, Lujo Bauer, and Limin Jia. 2018. Riding out DOMsday: Toward Detecting and Preventing DOM Cross-Site Scripting. In *NDSS*.

[19] Benedik Meurer and Yang Guo. 2018. Zero-cost async stack traces. https://bit.ly/v8-zero-cost-async-stack-traces

[20] Leo A Meyerovich and Benjamin Livshits. 2010. ConScript: Specifying and enforcing fine-grained security policies for Javascript in the browser. In *Oakland*.

[21] Mark S Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. 2008. Safe active content in sanitized JavaScript. *Google, Inc., Tech. Rep* (2008).

[22] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. 2012. You are what you include: large-scale evaluation of remote javascript inclusions. In *CCS*.

[23] Inian Parameshwaran, Enrico Budianto, Shweta Shinde, Hung Dang, Atul Sadhu, and Prateek Saxena. 2015. Auto-patching DOM-based XSS at scale. In *Joint Meeting on Foundations of Software Engineering*.

[24] Riccardo Pelizzi and R Sekar. 2012. Protection, usability and improvements in reflected XSS filters.. In *ASIACCS*.

[25] Phu H Phung, David Sands, and Andrey Chudnov. 2009. Lightweight self-protecting JavaScript. In *International Symposium on Information, Computer, and Communications Security*.

[26] Prateek Saxena, Steve Hanna, Pongsin Poosankam, and Dawn Song. 2010. FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications.. In *NDSS*.

[27] Peter Snyder, Cynthia Taylor, and Chris Kanich. 2017. Most Websites Don't Need to Vibrate: A Cost-Benefit Approach to Improving Browser Security. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 179–194.

[28] Sooel Son and Vitaly Shmatikov. 2013. The Postman Always Rings Twice: Attacking and Defending postMessage in HTML5 Websites. In *NDSS*.

[29] Marius Steffens, Christian Rossow, Martin Johns, and Ben Stock. 2019. Don't Trust The Locals: Investigating the Prevalence of Persistent Client-Side Cross-Site Scripting in the Wild.. In *NDSS*.

[30] Ben Stock, Martin Johns, Marius Steffens, and Michael Backes. 2017. How the Web Tangled Itself: Uncovering the History of Client-Side Web (In) Security. In *USENIX Security*.

[31] Ben Stock, Sebastian Lekies, Tobias Mueller, Patrick Spiegel, and Martin Johns. 2014. Precise Client-side Protection against DOM-based Cross-Site Scripting. In *USENIX Security*.

[32] Ben Stock, Stephan Pfistner, Bernd Kaiser, Sebastian Lekies, and Martin Johns. 2015. From Facepalm to Brain Bender: Exploring Client-Side Cross-Site Scripting. In *CCS*.

[33] Mike Ter Louw, Karthik Thotta Ganesh, and VN Venkatakrishnan. 2010. AdJail: Practical Enforcement of Confidentiality and Integrity Policies on Web Advertisements.. In *USENIX Security*.

[34] Mike Ter Louw and VN Venkatakrishnan. 2009. Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. In *Oakland*.

[35] Steven Van Acker, Philippe De Ryck, Lieven Desmet, Frank Piessens, and Wouter Joosen. 2011. WebJail: least-privilege integration of third-party components in web mashups. In *ACSAC*.

[36] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. 2007. Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis.. In *NDSS*.

[37] W3C. 2010. Same Origin Policy. https://www.w3.org/Security/wiki/Same_Origin_Policy.

[38] Lukas Weichselbaum, Michele Spagnuolo, Sebastian Lekies, and Artur Janc. 2016. CSP is dead, long live CSP! On the insecurity of whitelists and the future of Content Security Policy. In *CCS*.

[39] Michael Weissbacher, William K Robertson, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. 2015. ZigZag: Automatically Hardening Web Applications Against Client-side Validation Vulnerabilities.. In *USENIX Security*.