Don't Trust The Locals: Investigating the Prevalence of Persistent Client-Side Cross-Site Scripting in the Wild

Marius Steffens*, Christian Rossow*, Martin Johns†, and Ben Stock*
*CISPA Helmholtz Center for Information Security: {marius.steffens,rossow,stock}@cispa.saarland
†TU Braunschweig: m.johns@tu-braunschweig.de

***Abstract*—The Web has become highly interactive and an important driver for modern life, enabling information retrieval, social exchange, and online shopping. From the security perspective, Cross-Site Scripting (XSS) is one of the most nefarious attacks against Web clients. Research has long since focused on three categories of XSS: Reflected, Persistent, and DOM-based XSS. In this paper, we argue that our community must consider at least four important classes of XSS, and present the first systematic study of the threat of Persistent Client-Side XSS, caused by the insecure use of client-side storage. While the existence of this class has been acknowledged, especially by the non-academic community like OWASP, prior works have either only found such flaws as side effects of other analyses or focused on a limited set of applications to analyze. Therefore, the community lacks in-depth knowledge about the actual prevalence of Persistent Client-Side XSS in the wild.**

To close this research gap, we leverage taint tracking to identify suspicious flows from client-side persistent storage (Web Storage, cookies) to dangerous sinks (HTML, JavaScript, and `script.src`). We discuss two attacker models capable of injecting malicious payloads into storage, i.e., a Network Attacker capable of *temporarily* hijacking HTTP communication (e.g., in a public WiFi), and a Web Attacker who can leverage flows into storage or an existing reflected XSS flaw to persist their payload. With our taint-aware browser and these models in mind, we study the prevalence of Persistent Client-Side XSS in the Alexa Top 5,000 domains. We find that more than 8% of them have unfiltered data flows from persistent storage to a dangerous sink, which showcases the developers' inherent trust in the integrity of storage content. Even worse, if we only consider sites that make use of data originating from storage, 21% of the sites are vulnerable. For those sites with vulnerable flows from storage to sink, we find that at least 70% are directly exploitable by our attacker models. Finally, investigating the vulnerable flows originating from storage allows us to categorize them into four disjoint categories and propose appropriate mitigations.

I. Introduction

The Web is arguably the most important platform of today's Internet. It offers a plethora of applications, from widely used social media platforms to full-fledged office applications. Given the growth of functionality implemented for the client, the complexity of client-side code rises. This trend is naturally accompanied by an increase in flaws. One of the most devastating attacks is Cross-Site Scripting (XSS), allowing an adversary to execute arbitrary JavaScript code in the context of a vulnerable application. This can be used to, e.g., exfiltrate sensitive information such as access tokens or to post content in the name of the victim.

Cross-Site Scripting was first discussed in 2000 and was believed to be a server-side issue. Klein [26] was the first to discuss its client-side counterpart, which he dubbed *DOM-based Cross-Site Scripting or XSS of the Third Kind* given that it (ab)used functionality in the Document Object Model (DOM), and appeared to be a third kind of XSS (in addition to reflected and persistent XSS on the server). This notion of *three* types of XSS has been upheld for years in research [32, 36, 43] and widely accepted textbooks [67]. Apart from these, all other types of XSS have been treated as niche problems (e.g., mutation-based XSS [19]). Similarly, due to the absence of empirical evidence about its prevalence, Cross-Site Scripting enabled by persistence APIs on the client has not been acknowledged as an important type of XSS. The detection, mitigation, and prevention of Persistent Server-Side XSS and reflected client-side XSS have received much attention (e.g., [12, 13, 25, 26, 29, 32, 36, 43, 54, 55, 57]). However, while prior work has found anecdotal evidence of Persistent Client-Side XSS [17, 31, 32, 68], the dangers of insecure client-side uses of stored code and data under potential control of an adversary have not been studied systematically.

To close this research gap, in this paper, we investigate the risk of using data from client-side storage in JavaScript, showing that *Persistent Client-Side XSS* must, in fact, be considered as a real threat to modern Web applications. To demonstrate its prevalence, we investigate how many sites could potentially fall victim to Persistent Client-Side XSS flaws. We leverage taint tracking in the browser to find exploitable flows of data from Web Storage or cookies to dangerous sinks, such as `eval`. Considering a Network Attacker capable of *temporarily* hijacking a non-encrypted connection, and a regular Web Attacker capable of forcing her victims to visit arbitrary URLs, we report on a study of vulnerabilities in the Alexa Top 5,000 domains. We show that over 8% of the analyzed sites exhibit exploitable flows from client-side storage to code-executing sinks, and even a 21% ratio if we only consider applications which make use of any client-side persisted data. Of all domains with vulnerable flows from storage, we find that at least 70% are exploitable by widely accepted attacker models.

Network and Distributed Systems Security (NDSS) Symposium 2019
24-27 February 2019, San Diego, CA, USA
ISBN 1-891562-55-X
https://dx.doi.org/10.14722/ndss.2019.23009
www.ndss-symposium.org

Based on our insights, we investigate the intended uses of the vulnerable data flow and discuss appropriate countermeasures.

To sum up, our paper makes the following contributions:

- Based on the notion of Persistent Client-Side XSS, which we present along with two different attacker models to persist their malicious payload on the client (Section III), we outline a methodology to determine which sites on the Web are susceptible to such attacks and discuss our implementation (Section IV).
- We report on a large-scale empirical study of the Alexa Top 5,000 domains w.r.t. the prevalence of Persistent Client-Side XSS flaws (Section V).
- We showcase the underlying problems which enable exploitation on 418 of the Alexa Top 5,000 domains and evaluate specific countermeasures for issues we discovered in our large-scale analysis (Section VI).

II. Technical Background

In this section, we briefly discuss the relevant technical background, i.e., means of persistent storage on the client as well as the concept of (reflected) Client-Side XSS.

A. *Persistent Storage on the Client*

HTTP as a protocol does not have a notion of state, but rather comprises a single connection between client and server to transmit data. Any state is lost once the connection is closed. To overcome this, Netscape Mosaic introduced the idea of cookies in 1994 [67]. These simple key-value stores are used by browsers to persist small pieces of string data, which are sent along in every HTTP request to matching servers. This allows sites to overcome the inherent limitation of HTTP, i.e., the lack of state. Cookies are widely used for session management when sites offer a login and for tracking purposes, but are also often used to store preferences, such as the selected language. Cookies in browsers are bound to a domain or hostname. Depending on the browser, not specifying a host restricts cookies to only the exact host. However, cookies can also be set with a specific host (which must be a parent domain of the current host). In that case, they are valid for *any* subdomain of the set host. This is important to note, since `sub1.example.org` can set cookies for `*.example.org` (including `example.org`), and a non-HTTPS site can set cookies for its HTTPS counterpart [39, 68].

Cookies have a number of drawbacks, specifically related to the type and length of content they can store. RFC 2965 states that browsers should offer at least 4096 bytes of storage per cookie [28], which is, e.g., in Chrome/Chromium implemented as the maximum value[1]. In the interest of implementing more complex applications on the client, which in turn required more storage for persistent data, the WHATWG (and later W3C) introduced the Web Storage API [21]. It consists of two containers, namely Session and Local Storage. While the former only persists data for the duration of a browsing session and is unique for each window, Local Storage makes it possible to indefinitely persist data on the client. In contrast to cookies, this is meant for larger pieces of information and allows for easily setting and retrieving data (in JavaScript, reading a cookie's value requires parsing a string containing all cookies). Unlike cookies, Web Storage is bound to an *origin*. This denotes the tuple of protocol, host, and port of a document. This means that `https://a.com` and `http://a.com` do not share the same Local and Session Storage.

```
var externalScript = document.createElement("script");
script.src = "https://example.com/foo.js";
document.body.appendChild(externalScript);
```

Fig. 1. Example usage of `script.src`

B. *Client-Side Cross-Site Scripting*

The most basic security policy in browsers it the *Same-Origin Policy* [67]. This policy governs interactions between sites, specifically as they relate to the access from JavaScript to another window. Whenever a JavaScript snippet tries to gain access to resources from another window (e.g., popup or iframe), the action is only allowed if the accessed resource shares the origin of the JavaScript. This way, a malicious site including an iframe to another domain cannot read the content of the rendered document from that domain.

Cross-Site Scripting (XSS) is a code injection attack, in which an adversary is able to add JavaScript code of her choosing to a vulnerable site. This code is then executed in the origin of the vulnerable application, allowing the code to interact with the vulnerable page as the user could. In our notion, XSS can be roughly categorized in two dimensions. Specifically, it can be *reflected* or *persistent*, and can be located in the *server-side* or *client-side* code. *Reflected* here refers to the fact that the vulnerable code reflects back some attacker-controllable information, e.g., using PHP's `echo` functionality to print part of the requested URL on the server. *Persistent* in turn means that the malicious payload is not directly echoed back by the code, but rather stored and later retrieved. Since this paper investigates the prevalence of Persistent *Client-Side* XSS, we omit further details on its server-side counterpart.

Abstractly speaking, an XSS attack occurs if some attacker-controllable piece of data flows into a dangerous sink. On the client, these sinks can be classified into three categories: rendering of HTML, execution of JavaScript, and the inclusion of additional script resources. For HTML, sinks include `document.write` or `innerHTML`, whereas for JavaScript, the dreaded `eval` construct (among others) can be used to convert strings to JavaScript code during runtime. The Document Object Model (DOM) in browsers also allows for the direct assignment of properties for any HTML node. This can be used at runtime to dynamically add script resources. Figure 1 shows an example of this. Using `createElement`, a script element is created and the script's `src` property is subsequently specified. When the script element is then appended to the document's body, the rendering engine downloads and executes the referenced script. These sinks merely exemplify the different categories. In addition to the sinks used by prior work [32, 36], we also consider `script.src`.

Prior works have already investigated the prevalence and nature of *reflected* Client-Side XSS [32, 36, 55]. In these cases, a reflected Client-Side XSS occurs whenever data

[1]https://chromium.googlesource.com/chromium/src/+/master/net/cookies/parsed_cookie.h#25

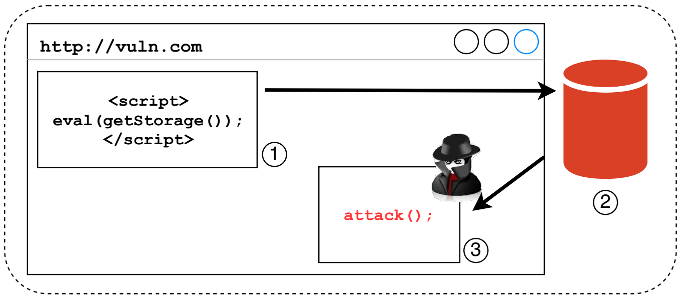

Fig. 2. Persistent Client-Side XSS Attack

originating from the URL is insecurely used by JavaScript. The URL can be accessed over several different APIs, e.g., `location.href`. In the following section, we outline the differences between reflected and persistent XSS on the client and explain the different attacker models we consider for a Persistent Client-Side XSS attack.

III. Persistent Client-Side XSS

In this section, we discuss the notion of persistent Client-Side Cross-Site Scripting, highlighting how the insecure use of persisted data can be abused by an adversary to execute her malicious code. We then relate this concept to its server-side counterpart and introduce two attacker models, which enable storing payloads in her victim's persistence APIs, allowing for a persistent XSS attack.

An XSS attacker's goal is often to hijack the session of their victim, i.e., steal the authentication cookies. This problem is mitigated by the use of HTTP-only cookies, which ensure that cookies cannot be accessed from JavaScript and are therefore out of reach of the adversary. Additionally, the attacker can also force the victim's browser to perform certain actions, such as post content. This attack becomes more powerful if the attacker conducts a *resident XSS* attack [23], which leverages the existing XSS to ensure that all links a user may visit are also XSS-infested. However, once the victim closes the browsing session, even this threat is eliminated. If, however, the malicious payload is persisted on the client, i.e., in cookies or Local Storage, the XSS attacker's code is revived on every subsequent load of the flawed site, even without the necessity to add the payload to, e.g., the URL. Hence, this allows the attacker to mount attacks such as JavaScript keyloggers[2], Cryptominers [11], or the previously outlined scenarios even long after the initial attack has occurred. Especially in the case of a network-based adversary, the danger is aggravated due to the fact that cookies are shared between HTTP and HTTPS sites. We discuss a particularly high-profile case we discovered in our analysis in Section V-D.

A. Vulnerable Use of Persisted Data

Figure 2 shows the basic steps to a successful execution of attacker-controlled code in a Persistent Client-Side Cross-Site Scripting attack. The vulnerable site hosts a JavaScript snippet which extracts additional code from storage and subsequently executes this code using `eval` (1). The browser accesses the storage (2), retrieving the code to be executed. In the third step, if this code is under the control of the adversary, the malicious code is executed in the origin of the vulnerable site (3). In this example, the data originating from Local Storage was passed to `eval` in an unfiltered manner, indicating that the actual purpose of this storage entry was to persist code.

[2]e.g., https://blog.rapid7.com/2012/02/21/metasploit-javascript-keylogger/

```
var value = localStorage.getItem("entryPage");
document.write("<a href='" + value + "'>start over</a>");

// Assume adversary sets entryPage to
↪  '><script>alert(1)</script>
document.write("<a href=''><script>alert(1)</script>'>start
↪  over</a>")
```

Fig. 3. Example vulnerability involving a data flow from Local Storage to `document.write`

Such vulnerable patterns are not specific to the scenario in which *code* is persisted in Local Storage. In fact, the intended purpose of Local Storage is to store *data*. However, in practice, `eval` is also often (ab-)used to parse JSON [45] data, even though secure alternatives exist. Moreover, storage can be used to store unstructured data, which may be used in a flawed way. For example, Figure 3 shows a snippet which uses stored data in an insecure way. The purpose of this snippet is to extract the URL of the page on which a user's workflow started and use it to create a link back to that URL. However, the value extracted from storage is neither checked for its format nor encoded to ensure that the extracted value cannot be abused to add additional HTML markup. Specifically, if the adversary gains control of the stored value, she can modify it to break out of the `a` tag, and inject a new script element (see Figure 3). We report on the specific patterns of such vulnerable flows we discovered in our study in Section VI.

Web Storage is not the only feature that can be abused for Persistent Client-Side XSS. The outlined attack can be transferred to cookie sources as well. While cookies have only limited storage, the size of each cookie is still sufficient to exploit an unfiltered flow from a cookie. Although in general, Session Storage allows to persist data on the client, it is bound to a browsing window and deleted when said window is closed. Hence, we do not consider Session Storage for our work, as its short-term persistence does not add any capabilities a regular XSS attacker lacks.

B. Differences From Persistent Server-Side XSS

Persistent XSS on the server side has been studied for several years and is widely known. In one of the most recent papers, Dahse and Holz [12] found several PHP-based applications to be susceptible to persistent XSS, including the popular OpenConf submission system. Such vulnerabilities occur when an adversary's input is not filtered or encoded before being written to persistent storage, such as a SQL database. A famous example of such an attack is the hack of the Ubuntu forums in 2013 [59]. Attackers leveraged a persistent XSS flaw in the deployed forum software to take over an administrative account, leading to a complete database compromise. The post-mortem analysis showed the evidence for these actions, given that the malicious payload was in fact stored in the database (and hence could be recovered).

In contrast to its server-side counterpart, Persistent Client-Side XSS does not leave any trace in a central database and is therefore harder to detect. Rather, this attack targets a user's storage. For the adversary, this means she must persist her malicious payload into the storage of every user she wants to attack. At the same time, when the JavaScript code causing the vulnerable flow from storage to sink is included in every page of a domain, a single injection means that regardless of which URL on the domain is visited, the attack succeeds (assuming the JavaScript snippet causing the flow is included). Moreover, a single error or misconfiguration on such a domain is sufficient to persist a payload. We discuss the specific requirements for persisting a payload as well as the accompanying attacker models in the following.

C. Persisting Malicious Payloads

The prerequisite to the two outlined attacks is that the adversary controls the content of her victim's storage. In the following, we outline two attacker models capable of injecting arbitrary payloads into the aforementioned persistence APIs *once*, allowing for subsequent exploitation on *every* page visit.

1) Network Attacker: A network-level adversary is able to inject arbitrary packets into any unencrypted connection between a client and server. Moreover, she can choose to completely drop all packets to the actual server requested by the client and instead respond with an HTML page of her choosing. We do not assume that this adversary can obtain valid TLS certificates for the sites she wants to attack. Hence, whenever the connection between the client and desired server is secured via TLS, the adversary cannot inject anything into the HTML page. Given these characteristics, the adversary can conduct attacks over HTTP connections, allowing her to modify cookies as well as Local Storage. This Network attacker, which is in line with prior research in the Web security area [17, 51, 68], might be the owner of a coffee shop providing free WiFi to her customers. A security-aware user might refrain from performing any sensitive actions in such an open network, e.g., performing a login or doing online banking. However, using a Persistent Client-Side XSS, the attacker can implant a malicious payload which lies dormant and is used only later to attack a victim. One such scenario is a JavaScript-based keylogger, which is triggered upon visiting the site with infected persistent storage in a seemingly secure environment, e.g., at home. If we consider a recent attack on MyEtherWallet [44], in which attackers used rogue BGP advertisements to take over routes, the threat of a Network Attacker becomes even more severe. For such an attack to work, the malicious payload needs to be persisted in the browser of the victim. In the following, we briefly outline how this can be achieved for the two persistence APIs we consider.

Cookies can be manipulated either by setting them in the faked HTTP response, or by delivering JavaScript code which achieves the same. Moreover, given the fact that cookies are not bound to an origin, but rather the domain, this allows the adversary to set cookies for any parent domain as well as the HTTPS-enabled variant. The ability to inject arbitrary cookies is hindered by HTTP Strict Transport Security (abbreviated *HSTS*) [22]. This HTTP header ensures that a site will only be accessible via HTTPS. This way, even if an adversary tries to force her victim to connect to the HTTP site, the victim's

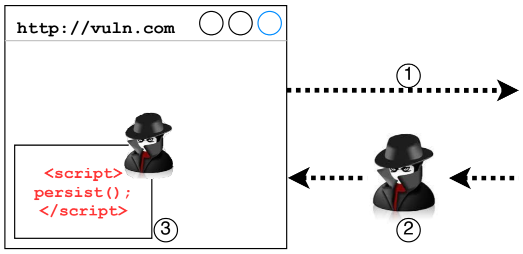

Fig. 4. Network Attacker Persistence

browser would automatically establish an HTTPS connection to the site. Given that the attacker has full control over the HTML presented to her victim, she can also force the victim's browser to connect to any other domain over HTTP, for which the attacker again controls the HTML. This means that if the victim visits `a.com` over an unencrypted HTTP connection, the adversary can inject an iframe pointing to `b.com` over HTTP (assuming this site did not deploy HSTS). Intercepting this subsequent request allows the adversary to also deploy code running on `b.com`. Even if `b.com` deploys an HSTS header, but is missing the `includeSubDomains` option, the adversary can simply point the iframe to `foo.b.com`, enabling her to set cookies for `b.com`. If a cookie with the same name has already been set by the parent domain, an attacker can leverage the fact that browsers only allow a fixed limit of cookies per *domain* [46]. As of this writing, Chrome and Firefox support up to 180 cookies per domain. Therefore, the cookie with the duplicate name can be flushed easily, and subsequently be set to the attacker's chosen value.

Unlike to cookies, the Local Storage is bound to the origin of a domain, i.e., a site loaded via HTTP cannot inject any content into its HTTPS counterpart. Therefore, we assume that the adversary can set or remove any item from the Local Storage for the *origin*. Hence, while in the aforementioned scenario, the adversary could control the Local Storage for `http://b.com`, she cannot access or modify the Local Storage for `https://b.com`.

Figure 4 shows the attack by a Network Attacker. The user visits a vulnerable site (containing a flow from persistent storage to a sink) over HTTP (1). Since the attacker can arbitrarily modify packets, she intercepts the response from the server (2), manipulating the content to her liking. Specifically, she adds an additional script resource (3), which is used to persist the payload necessary to conduct subsequent attacks, which work even in the absence of the Network Attacker.

2) Web Attacker: Contrary to the Network Attacker, this type of adversary cannot inject any packets into arbitrary connections. Instead, she can host her own site and lure the victim there. On that site, she can force her victim's browser to load any resource from arbitrary origins, controlling the HTTP parameters. This enables two scenarios for persisting a malicious payload into cookies or Web Storage.

The first vector for abusing an existing reflected XSS flaw to persist the payload is to use a regular flow of data from the URL into any of the persistence APIs. This pattern is similar to a persistent XSS attack on the server side. This does not require any other flaw on the site itself. Instead, the

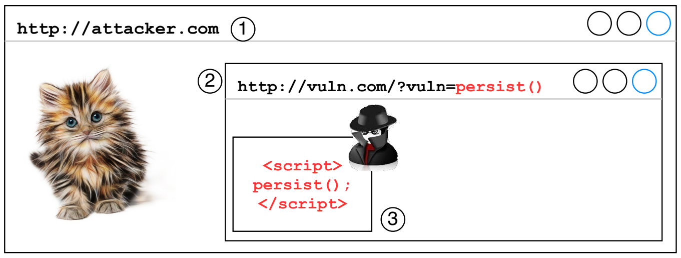
http://attacker.com
1
2
http://vuln.com/?vuln=persist()
<script> persist(); </script>
3

Fig. 5. Web Attacker Persistence

site must execute JavaScript code which conducts said data flow. More importantly, just *any* flow from URL to storage is not sufficient: it must be a flow into the storage item which is later insecurely used. Moreover, while an adversary executing JavaScript can easily control all parameters to the invocation of `document.cookie` (including the domain for which the cookies are to be set), she might not have the same level of control when trying to abuse an existing flow into a cookie.

An alternative attack vector requires a site to be vulnerable to a *reflected* Client-Side Cross-Site Scripting flaw. By forcing her victim's browser to load the vulnerable page, the adversary can execute her JavaScript code in the origin of the vulnerable site. This code can then access and modify cookies (for any parent domain) as well as Local Storage (for the specific origin). Hence, the adversary can craft this initial payload such that rather than exploiting the existing flaw (e.g., to exfiltrate sensitive data), it persists the actual payload into cookies or Local Storage. If the manipulated value is used later on, e.g., in a call to `eval`, the actual payload gets executed. Notably, this then occurs on every page load, without the adversary needing to force her victim to visit a crafted URL.

The different steps of this attack are depicted in Figure 5. First, the attacker lures her victim to a site of her choosing, e.g., by hosting interesting content such as cat pictures (1). Apart from the actual image of a cat, this page also contains a hidden iframe, which points to the vulnerable site (2). Specifically, the attacker carefully crafts the URL such that the existing *reflected* XSS vulnerability is triggered. The injected code, which runs in the origin of the vulnerable site, now puts the desired payload into storage (3).

IV. Methodology

In this section, we outline our methodology to detect Persistent Client-Side XSS in Web applications. Technically, we search for exploitable flows from either cookies or Web Storage to a dangerous sink. Specifically, we are interested in answering the following questions:

- How many sites make use of data originating from storage in their client-side code?
- On how many sites can such a data flow be abused if an adversary can gain control over the storage?
- Out of these, how many sites can be successfully attacked by a Network and Web adversary?

To that end, this section presents our flow and storage collection, followed by an explanation of our enhanced exploit generation scheme. With this generation in place, we outline how we can determine whether a site is vulnerable, assuming an attacker controls arbitrary storage items. We end the section with a discussion on how we can determine whether a site which is theoretically vulnerable can be exploited by a network or Web adversary.

A. Flow and Storage Collection

Previous taint tracking approaches to discover XSS vulnerabilities have primarily focused on reflected XSS. Our prior work in 2013 introduced an automated taint-aware system [32] to detect this type of XSS and in 2018, Melicher et al. [36] open-sourced a reimplementation on a more current version of Chromium.

Our engine is based on Chromium and is capable of attaching taint information to strings. Moreover, all JavaScript string operations (e.g., concatenation or substrings) as well as the built-in encoding functions like `escape` pass on the taint. We taint all values from relevant sources (e.g., Local Storage and cookies). We modified Chromium to report invocations with tainted data to numerous sinks. Specifically, this includes all sinks which allow for creation of HTML (e.g., `document.write` or `innerHTML`, execution of JavaScript code (e.g., `eval`) as well as adding additional script resources (via `script.src`). Apart from these flows which might lead to a direct JavaScript execution, the engine also collects those flows that end in a persistent storage, specifically cookies and Local Storage. On top of this, we build a crawling extension, which recursively crawls a given domain, and reports all found data flows to a database.

Besides the collection of relevant flows, our crawling extension iterates over all cookies and all elements in Local Storage. This collection is done via JavaScript injected into the page itself. This does not allow us to collect *HTTPOnly* cookies (which are meant only to be sent in HTTP requests, and are not accessible from JavaScript). Nevertheless, if a vulnerable flow within JavaScript is to occur, it cannot originate from such an HTTPOnly cookie. Therefore, our collection method suffices to gather all cookies which can be the source of a flow. For all elements, we store both key and values, allowing us to determine which cookies or storage items to modify in the exploit generation phase.

B. Exploit Generation

The second step towards finding Persistent Client-Side XSS in the wild is a fully automated approach to generating exploits. We extend the approaches previously presented by us [32] and incorporate techniques presented by Melicher et al. [36], which were tailored to build *reflected* XSS payloads. We refer the reader to the previous works for exact technical details and instead provide a brief overview of the techniques here.

In general, exploits generated by either method use a *break-out sequence*, which, depending on where the attacker controls parts of the executed code, closes all preceding elements (e.g., string quotations or function declarations), allowing for subsequent injection of the attacker's code. We consider the example from Figure 6. To craft an exploit, our original approach would generate a sequence which breaks out of the existing context and subsequently calls some function, e.g., `alert`. It would then add a *generic* break-in sequence, i.e., a

```
var userinfo = getCookieValue("userinfo");
eval("var user = '" + userinfo + "';");
```

Fig. 6. Example of context-aware break-out/break-in

sequence which comments out the rest of the line (and differs between HTML and JavaScript sinks). Hence, the algorithm determines that the value of `userinfo` should be set to `';alert(1);//`. If this value originates from the URL (not shown in the example for brevity), the resulting sequence is then substituted into the URL at the position of the tainted data. However, this neglects the possibility that the tainted data is part of structured data. Additionally, the break-out sequence is overly aggressive, which leads to problems given the simplistic break-in sequence proposed, especially in scenarios where the usable character set is limited (such as cookies). Thus, we extend both approaches in three dimensions: a context-aware break-out and break-in sequence, an improved replacement strategy, and a fuzzy matching approach to find the correct storage item to replace. Our codebase is open-source[3] and we outline our enhancements in the following.

1) Context-Aware Break-Out/Break-In: The enhanced break-out and break-in strategy can be best illustrated with the snippet shown in Figure 6, assuming that the function `getCookieValue` returns the value of the cookie with the key `userinfo`. As previously discussed, the result of the exploit generation from prior works is `';alert(1);//`. However, the `;` acts as the delimiter between two cookies. Hence, setting the cookie's value accordingly would have no effect, since only the `'` would be extracted when the cookie is accessed. To overcome this drawback, we implement our exploit generation such that in these cases, instead of trying to completely break out of the existing context, we use an alternative that does not require the use of a `;`. In JavaScript, almost all types of data can be concatenated. Therefore, a payload that triggers our injected code is `' + alert() + '`. This way, we concatenate two empty strings with the invocation of `alert`, hence executing the injected function. Naturally, `alert` is merely a placeholder, which could be replaced with arbitrary JavaScript code. Note also that it is feasible to craft a payload to add additional scripts merely using `eval` and `String.fromCharCode`, which does not require the use of a `;`.

2) Improved Replacement Strategy: As the third improvement over existing approaches, we implement an enhanced replacement algorithm. For this, consider the code snippet shown in Figure 7. The code here reads a value from the Local Storage, parses it with `JSON.parse` and then uses the property `id` in an invocation of `document.write`. Based on the invocation of `document.write`, our exploit generation scheme determines that the required payload is `"><script>alert(1)</script>`, which first breaks out of the `a` tag and subsequently adds a new `script` element. The previous approaches [32, 36] use simple string replacement. If we apply this here, i.e., replace the string `test123` with the aforementioned payload, and store the value accordingly in the Local Storage, JSON.parse will fail. This is because the stored value would now be `{"id": ""><script>alert(1)</script>"}`. In this case, the injected double quote breaks the JSON format. Therefore, instead of simply replacing the values, our replacement algorithm first checks if the stored data is JSON. If this is the case, it is parsed to a Python dictionary, in which the value `test123` is replaced. As the last step, the dictionary is converted back to JSON, ensuring that the necessary double quote is properly encoded. This is done in an analogous manner for data that is encoded, e.g., using JavaScript's built-in functions `encodeURI` or `escape`.

```
// Local Storage userinfo originally contains
↪  {"id":"test123"}
var userinfo = JSON.parse(localStorage.getItem("userInfo"));
document.write('<a href="/profile/' + userinfo["id"] +
↪  '">Profile</a>');
```

Fig. 7. Example of exploit generation, involving use of JSON.parse before sink access

3) Fuzzy Matching: Another difference between exploits for a URL and a storage item is the fact that the storage items are regularly modified during runtime. Due to its light-weight nature, our taint tracking approach can be used to track that a piece of data originates from a persistent source, but not from which specific entry. Therefore, we need to implement a matching approach to find the correct item to replace. Consider the example in Figure 8. Here, `eval` is used to parse JSON, meaning that our taint tracking engine records an invocation of `eval`, passing `{"visits": 1}`. However, the value is subsequently increased and updated in the Local Storage. Since our crawlers only record the final state of the storage after the page has been loaded, they would collect `{"visits": 2}`. As a result, the value which was originally involved in the flow is no longer in the storage. To tackle this issue, when no direct match can be found, we determine whether the value in the sink contained JSON. If so, we parse the value and determine if our database contains an entry, which (after being parsed) has the same keys. This way, we know which storage item to replace in the next step.

C. Determining Exploitability

Our exploit generation scheme produces a tuple of a potentially vulnerable URL, the type of storage (cookie or Local Storage), the storage key, and the corresponding value to which said key must be set. To determine how many domains are potentially susceptible to an attack, we visit each of the URLs with our crawlers. Before starting to analyze a new URL, all storage entries are cleared as to avoid side-effects from previous checks. For each of the tuples in our database, our analysis system then visits the URL under examination once to populate the storage as the site would normally do. After a timeout of two seconds, we overwrite the storage key with the value generated by our exploit generator. Specifically,

```
// Local Storage visitinfo originally contains {"visits":1}
var visitinfo = eval(localStorage.getItem("visitinfo"));
visitinfo["visits"] += 1;
localStorage.setItem("visitinfo", visitinfo);
```

Fig. 8. Example of fuzzy matching of storage elements

[3]https://github.com/cispa/persistent-clientside-xss

the payload output by the generator contains a call to a function which logs the successful exploitation to our database. Once the storage entry has been overwritten, we reload the page to see if our injected payload is executed.

To ensure that the payload is indeed persisted, the adversary can craft her payload such that write access to the storage is blocked. Given the dynamic nature of JavaScript, almost all functionality can be modified at runtime. This allows the attacker to deploy code which ignores invocations of Local Storage's `setItem` or `removeItem` functionality, and drops assignments to `document.cookie` if either of them would overwrite the malicious payload stored there. This can be implemented by overwriting the functions for `setItem` and `removeItem` on Local Storage, as well as by modifying the setter for `document.cookie` (using `Object.defineProperty` [40]). The implementation of a single case is trivial; automatically crafting this payload for each site we test, however, is not feasible. Therefore, we do not implement this for each specific case. Additionally, cookies may also be set by the server. However, this assignment occurs in the initial response phase, before any JavaScript is executed. Hence, if our payload is, in fact, overwritten by the server-sent cookie headers, our attack on the second load of the page fails.

With the aforementioned techniques, we collect all those sites that are potentially susceptible to a Persistent Client-Side XSS attack. "Potentially" here refers to the fact that we assume an omnipotent adversary, who can control the storage of arbitrary origins. In practice, however, this does not hold true. Therefore, to determine whether a site is vulnerable under our attacker models, we conduct additional analyses, which we outline in the following.

1) Network Attacker: The Network Attacker is able to hijack any HTTP communication, i.e., any site delivered over HTTP is vulnerable to be attacked, allowing the adversary to persist her payload. In an actual attack, we assume that the adversary would deploy an HTML page of her choosing to overwrite the storage values. This holds true for both cookies as well as Local Storage. While Local Storage is bound to an origin (i.e., the combination of protocol, host, and port), cookies are stored for a domain, i.e., ignore both protocol and port. Hence, when a site is delivered over HTTPS, but does not deploy HSTS, it is vulnerable. Note that HSTS must use the `includeSubDomains` modifier [27]. Otherwise, since any *child* domain can set cookies for its parent, an attacker could simply force the victim to visit a site for which HSTS is not set and overwrite the cookies. We conservatively assume that our victim has visited the vulnerable site at least once before, meaning that if a site deploys HSTS, the victim's browser has already recorded this and will only connect via HTTPS. To that end, we connect to each HTTPS origin on which we had discovered an exploitable data flow. If the site does not send an HSTS header, and moreover the domain is not contained in the HSTS preload list [9], we mark the origin (and its corresponding second-level domain) as vulnerable.

2) Web Adversary: To measure how many sites are susceptible to a Web Attacker, we need to analyze two types of attack vectors. First, if a site contains flows from the URL to a storage, where that stored value is later used insecurely in a flow to a sink, it is exploitable. Second, if the site carries a Reflected XSS flaw, it also has a *persistable* XSS vulnerability, i.e., a single-shot XSS can be turned into a persistent XSS due to the insecure use of storage data on the site. Our taint-aware browsing engine records all flows from the URL, i.e., those that directly end in sinks like `document.write` or `eval`, as well as flows into the storage.

To tackle the first attack vector, i.e., a flow of data from the URL to one of our considered persistance APIs and a subsequent flow (potentially on a different URL) from that storage to a dangerous sink, we first determine which storage keys were used in the vulnerable flow from storage to sink. Given this information, we search for occurrences of a write access to the corresponding keys. If such a write access is found, our exploit generator builds a URL which it believes can cause this flow. Subsequently, we set our crawlers to first visit the crafted URL, such that the payload is persisted. We then visit a URL on which the flow from storage to sink was detected and mark this as exploitable if visiting the second URL results in a successful invocation of our logging function.

For a persistable XSS, given the information about a flow from URL to execution sinks, we use the exploit generation scheme to derive payloads and to determine how each URL must be modified to craft an exploit. This is in line with our prior work [32] and the results presented by Melicher et al. [36]. In doing so, once we collect all origins on which a reflected XSS vulnerability could be discovered, we check these against the origins on which a flow from storage to sink has occurred. Again, given the nature of cookies, it is sufficient to find an XSS on a subdomain to exploit its parents, whereas for Local Storage the origins must match.

V. Results of Large-Scale Analysis

In this section, we analyze the prevalence of Persistent Client-Side XSS vulnerabilities on the Web. As a seed, we extracted the 5,000 highest ranked domains from the Alexa Top 1M list of April 28th, 2018 and ran our crawlers from May 2nd to May 4th, 2018. For each domain, we crawled up to 1000 sub-pages of at most depth 2 (in a breadth-first manner), where the start page is considered to have depth 0. In total, we visited 3,078,360 URLs. As most sites are not merely a single HTML page, but rather incorporate additional content in frames, our crawlers analyzed a total of 12,489,576 frames/documents.

A. Collected Data Flows

Table I shows the total number of flows into each of these sinks, split by the originating source. Within each combination of source and sink, the table also shows how many of the data flows ended in the sink without any encoding applied to them (denoted as the *Plain* columns). We observe that for HTML and JavaScript sinks, between 71% and 89% of flows from the URL are not encoded. As shown by previous work [32, 36], this leads to a number of exploitable flows which cause a *reflected* Client-Side XSS flaw. For flows originating from cookies, the fraction of plain flows ranges from 69% to 98%. Most interestingly, though, we observe that virtually all flows that originate from a Local Storage source have no encoding applied to them, indicating that this data appears to be trusted by the developers of the JavaScript applications.

Sink	URL Sources Total	URL Sources Plain	URL Sources Fraction	Cookie Source Total	Cookie Source Plain	Cookie Source Fraction	Local Storage Source Total	Local Storage Source Plain	Local Storage Source Fraction
HTML	11,388,607	10,161,040	*89.2%*	555,323	382,608	*68.9%*	2,180,680	2,149,839	*98.6%*
JavaScript	77,360	54,910	*71.0%*	535,047	522,205	*97.6%*	635,843	635,798	*100.0%*
Script Source	4,252,532	640,977	*15.1%*	1,458,687	256,034	*17.6%*	377,626	103,418	*27.4%*
Cookie	922,761	621,695	*67.4%*	31,391,553	12,615,945	*40.2%*	732,407	461,334	*63.0%*
Local Storage	890,808	878,139	*98.6%*	2,000,863	1,932,335	*96.6%*	66,635,820	66,175,494	*99.3%*

TABLE I. FLOW OVERVIEW, SHOWING HOW MANY DATA PARTS ORIGINATED FROM SOURCES (COLUMNS), ENDING IN THE SINKS OF INTEREST (ROWS). BESIDES THE TOTAL NUMBER OF FLOWS, IT SHOWS THE ABSOLUTE AND RELATIVE NUMBER OF FLOWS WHICH ARE NOT ENCODED.

Considering the results of the use of tainted data in the assignment of a script's source, we find that a much smaller fraction is used without encoding. This would seem to indicate that additional care is taken by developers when incorporating potentially attacker-controllable data in such assignments. The pattern, however, is due to the use of such tainted values, which are for the most part used in parameters to a URL. Hence, applying a function like `encodeURIComponent` merely ensures that the parameters are properly sent to the server, and is not necessary to avoid injections.

In addition to the flows to directly exploitable sinks, we find that more than 1.8M flows occurred from the URL to either cookies or storage (the last two rows in Table I), with many of them being unencoded. We also find evidence for numerous flows from cookies to cookies, as well as from storage to storage. This is to be expected, given that these are meant to store state, which is modified at runtime with JavaScript. In addition, we find that around 2M flows originate from cookies and end in a Local Storage sink, whereas another 732,407 flow in the opposite direction.

Note that the table shows absolute numbers, not a unique set of flows. This is due to the fact that determining uniqueness for these flows is infeasible. Earlier works [32, 36] used the combination of sink, domain, and code location of the sink access for uniqueness purposes. This, however, does not guarantee unique results, given that if multiple parts of an application's code use the same wrapper function (e.g., jQuery's `html`), all such flows would be counted as one.

Instead of identifying individual flows in the complete dataset, we now focus on the number of domains with invocations of HTML, JavaScript, and script source sinks. Table II shows the result of this analysis, indicating how many domains had any flow from cookies or Local Storage to a sink, as well as how many of these contained unencoded data. Note that the row *Total* is not a sum of the rows above, but rather a unique count of domains—attributed to the fact that a domain may have more than one type of flow. In total, we find that 1,946 domains in the 5,000 highest ranked sites make use of data from persistence APIs in a flow to either HTML, JavaScript, or a script's source. Within the domains, 1,645 have flows from cookies to sinks, and 941 use data from Local Storage in the invocation of sinks. For us, however, not all these domains are of interest, as we focus on those domains that have at least one *unencoded* flow from the persistence APIs to a sink. Therefore, in the following, we analyze the 906 cookie and 654 Local Storage domains with unencoded flows in more detail. This amounts to 1,324 unique domains for our analysis.

Sink	Cookie Total	Cookie Plain	Cookie Expl.	Local Storage Total	Local Storage Plain	Local Storage Expl.
HTML	496	319	132	234	226	105
JavaScript	547	470	72	392	385	108
Script Src	1,385	533	17	626	297	11
Total	1,645	906	213	941	654	222

TABLE II. NUMBER OF DOMAINS WHICH MAKE USE OF A COOKIE/STORAGE VALUE IN A SINK ("TOTAL"), ON WHICH AT LEAST ONE OF THESE FLOWS IS UNENCODED ("PLAIN"), AND ON WHICH AN ATTACKER COULD THEORETICALLY EXPLOIT SUCH A FLOW ("EXPL.").

B. Exploitable Flows from Persistent Storage

In order to determine how many of these flows could, in fact, be exploited, we first determined how many domains would be attackable by an unlimited adversary, i.e., an adversary capable of modifying cookies or Local Storage for arbitrary origins. To that end, we used a Chrome extension to first visit each URL in question, modify the storage accordingly, and reload the site (see explanation in Section IV-C1). If on the second page visit, the payload is triggered, we mark the site as exploitable. In total, we found that 418 of the 1,324 domains we considered in fact contained an exploitable flow from cookies or Local Storage. The exact number of domains for the combination of sinks and sources is shown in Table II in the respective third columns. Note that the sum of all exploitable domains in the table amounts to more than 418, as several domains suffered from more than one flaw.

We find that for HTML, 132 of 319 domains with unencoded flows from a cookie were exploitable, whereas 105 of 226 domains were determined to contain an exploitable flow from Local Storage to an HTML sink. This high ratio of domains, i.e., 40–46%, does not hold up for JavaScript sinks, where 72 of 470 (15%) and 108 of 385 (28%), respectively, were vulnerable. For both sink types, the data indicates the fraction of exploitable sites is higher for Local Storage than for cookie sites. In our experiments, we found this to have two reasons. First, since cookies are sent along to the server in every HTTP request, they are subject to inspection by deployed Web Application Firewalls (*WAFs*). Although we did not specifically record when a page was not loaded due to our cookies containing JavaScript or HTML markup, sampling sites on which our payload had not triggered frequently led to error pages clearly caused by WAFs. Second, our payload frequently required either `;` (for JavaScript) or = (for HTML) characters to work. Whenever a JavaScript program accesses the `document.cookie` property to gain access to the cookies, all cookies are extracted at once, in the format `key1=value1;key2=value2`. Similarly, when setting a cookie via JavaScript, the `;` character carries a special meaning, since it separates multiple directives. Therefore, through the use of these characters, our payload is destroyed. Although we believe that some sites may decode cookie values before use (therefore allowing us to properly encode the two characters when setting the cookie), our fully automated approach does not specifically target these corner cases.

For both cookies and Local Storage, the remaining cases where we could not exploit flows were caused by input validation, e.g., ensuring that only integer values were used or that the data being passed to `eval` matched the format of a JSON string. We defer the investigation of the use cases of persisted data in all sinks to the following section. It is worth noting, however, that for HTML and JavaScript, more than half of the domains that had a flow from Local Storage to a sink could be exploited, indicating that little care is taken in ensuring the integrity and format of such data.

The number of exploitable flows for the assignment of a script's source with (partially) tainted data is much lower than for the other two sink types, at around 3%. We investigated these domains to determine the exact reason and found that the values from storage were often merely used to send along unique identifiers or version numbers of scripts to be included. In the 28 cases where we could successfully exploit the flows, however, the persisted values contained the domain name of the script to be included.

C. Mapping Domains to Our Attacker Models

Our analysis shows that more than 8% of the top 5,000 domains are potentially susceptible to a Persistent Client-Side XSS vulnerability. Moreover, considering only such domains which make any use of tainted data in dangerous sinks, a staggering 21% (418/1,946) are vulnerable. Considering only the top 1,000 domains, we even found that 119 of them contained an unfiltered and unverified flow from cookies or Local Storage to an execution sink. While this fraction of almost 12% already indicates that such insecure use of persistently stored values is a widespread problem, our investigation of failed exploits also shows that in several cases the existing flaw was mitigated by WAFs (we verified that the injection did in fact work by using benign HTML markup). Given this fact, as well as the limited coverage of the crawled applications (e.g., without logins), we believe that these results are lower bounds on the actual number of potentially vulnerable sites.

Until now, we have considered an omnipotent adversary, who can inject arbitrary content into cookies and Local Storage. This model, naturally, does not hold true in practice. Instead, in the following, we consider the two models introduced in Section III, namely a Network Attacker capable of injecting packets into an unencrypted HTTP connection, as well as a Web Attacker, who can force the victim's browser to make arbitrary requests (e.g., to leverage a *reflected* XSS attack).

Considering the Network Attacker, we found that 293 of the 418 domains would, in fact, be exploitable, either due to a complete lack of HTTPS or due to a missing or incomplete (no `includeSubDomains`) deployment of HSTS, which allows for an HTTP-hosted site to set cookies for its HTTPS counterpart and parents. For the 213 cookie domains, we found that only 86 made use of HTTPS at all and 29 additionally deployed HSTS. Only 9 sites deployed the `includeSubDomains` flag, leaving a total of 204 cookie flow domains exploitable by the Network Attacker. The remaining 89/293 domains stem from HTTP domains, i.e., an attacker can also poison the Local Storage to persist her payload.

For the Web Attacker, we first check for flows from a URL to a storage entry, where that storage entry is later used in an unfiltered flow to a sink. However, out of the 20 domains for which we discovered such a flow, none could be exploited, for three reasons. First, flows originated from GET parameters in the URL, which when changed led to a 404 page. Second, only the host part of the URL was used to set the `domain` property of a cookie, meaning that we were unable to overwrite the cookie's `value`. Third, data from the URL was sanitized (e.g., the HTML brackets), rendering our payload non-functional.

To determine the number of persistable XSS flaws, we follow the methods outlined by prior works [32, 36] and found that 468 of the top 5,000 domains were susceptible to a *reflected* Client-Side XSS attack. Combining these domains with those domains that have flows from persistent storage to a sink, we determined that 65 of 418 domains allowed an adversary to persist her payload, making them altogether susceptible to Persistent Client-Side XSS. Again, it must be noted that this merely represents a lower bound, as our approach only considered reflected Client-Side XSS flaws. Additionally, since our crawlers neither log in nor try to cover all available code paths, the number of sites susceptible to such Client-Side XSS flaws is likely higher. In practice there are some minor caveats which need to be considered when using reflected XSS to persist a malicious payload, e.g., Safari separates the Web Storage and cookies of the normal and framed version of an origin[4]. An adversary thus needs to resort to a noisier infection vector such as pop-ups.

Overall, however, our results show that the flaws we discovered are not only theoretically exploitable. The high fraction of Web sites with a combination of code snippets which are susceptible to both *reflected* and *Persistent* Client-Side XSS (Web Attacker) underlines the relevance of this threat. Considering a more powerful adversary (like similar works in this space did [17, 51, 68]), capable of injecting arbitrary packets into any unencrypted HTTP connection (e.g., an arbitrary untrusted WiFi access point), shows that about 6% of the most frequented sites are susceptible to a *Persistent* Client-Side Cross-Site Scripting attack.

D. Case Study: Stealing Credentials from Single Sign-On

Based on the data we collected, we determined which types of sites could be susceptible to an end-to-end attack. In our study, we found the single sign-on part of a major Chinese website network to be susceptible to both a persistent and a reflected Client-Side XSS flaw. While abusing the reflected XSS could have been used to exfiltrate the cookies of the user, these were protected with the `HttpOnly` flag. Given the fact that the same origin also made insecure use of persisted code from Local Storage, however, rather than trying to steal the cookie, we built a proof of concept that extracted credentials from the login field right before the submission of the credentials to the server. This way, although a single session cannot be hijacked, the attacker can easily steal the credentials

[4]https://webkit.org/blog/8311/intelligent-tracking-prevention-2-0/

when these are entered by the unknowing victim. Given that the credentials are extracted before the form is posted, this also allows extending the exploits discussed by prior works on stealing credentials from password managers [50, 53]. To counter this specific exploit as well as the other flaws we found in our analysis, we investigate the different use cases that were meant to be implemented by the vulnerable code and show how each class of cases can be secured in the following section.

VI. Resolving Problematic Patterns

In this section, we investigate the root causes of the vulnerabilities and analyze the developer's underlying intention. Subsequently, we explore secure alternatives to the identified problematic practices. Addressing the issues found is not straightforward, as the applications' requirements leading to exposed insecurities are diverse. For this reason, we divide the identified cases into classes, representing the desired functionality. For each class, we then propose suitable application-level measures to address the Persistent Client-Side XSS flaw without loss of functionality. We identified four distinct types of data from client-side persisted data ending up in a potentially problematic sink: unstructured data, structured data, code, and configuration information. Each of these patterns requires specific defensive practices that we present in the following. We defer a discussion of more generic defenses to the end of the section.

A. Storage of Unstructured Data

The most common cases are scenarios in which the application's client-side code uses the persistence mechanisms to persist and retrieve textual data. This data is used in HTML sinks to add textual data to the Web document. In our dataset, we identified 214 domains, in which the legitimate data within the client-side storage did not contain any syntactical components, i.e., had no traces of HTML or JavaScript code. Thus, in such scenarios, the application can be protected effectively and robustly using context-aware sanitization [32, 36], which applies the appropriate encoding based on the syntactical context of the insertion point in the DOM.

B. Storage of Structured Data

The second unsafe use pattern that we were able to isolate in our dataset involved cases in which the applications persist JSON-like data structures. However, instead of using the safe browser-provided `JSON.parse` API or a custom parser after retrieval of the data from storage, these unsafe implementations instead passed the data structures directly into the `eval` instruction – an archaic way of parsing JSON dating back to the days in which browsers had no native support for the format. In consequence, manipulated data items directly lead to execution of injected JavaScript (similar to Figure 2).

Hence, replacing `eval` with the browser's native JSON capabilities is sufficient to robustly secure these cases. Indeed, a total of 81 vulnerable domains in our dataset could be fixed by simply using `JSON.parse`. However, `eval`-based JSON parsing is much more forgiving with respect to syntactical constraints compared to the browser's fairly strict JSON parser. Besides these straightforward cases, 27 domains use data formats that resemble JSON and can be "parsed" with `eval`, but are incompatible with `JSON.parse`. Thus, resolving these issues either requires changes to the stored format or introducing a safe custom parser to replace `eval`.

C. Storage of Code

The most challenging pattern in our dataset consists of scenarios in which applications use the persistence mechanisms to deliberately store HTML or JavaScript code, e.g., for client-side caching purposes. In this setting, the attacker is able to completely overwrite the contents of the corresponding storage entry with their own code. We could identify that in several cases these flaws are actually introduced by third-party libraries, among them Cloudflare and Criteo. In these scenarios, the use case mandates that the code semantics remain intact, as the HTML is meant to be reintroduced into the DOM and the JavaScript is meant to be executed. Hence, fully sanitizing the persisted information is not an option to resolve the issue, as this would break the application. Instead, we explore dedicated measures tailored to specific subclasses.

a) JavaScript Resources: A total of 90 sites persist JavaScript code, which is passed directly to `eval` on page load. Based on an analysis of the stored JavaScript code, we find that in the majority of the examined cases, the site caches JavaScript libraries to speed up page load time — such as Cloudflare's "Rocket Loader" [10], which we observed on 33 sites. This functionality ensures that all external scripts are cached in the Local Storage by substituting them with custom script media types, such that the browser does not fetch them on page load. The library then ensures that these resources are fetched, executed, and cached, allowing each subsequent visit to use the cached version. Thus, to securely enable this pattern, a method is needed that allows explicit caching of JavaScript resources on the client while keeping the code out of the adversary's reach. In modern browsers, *Service Workers* [47], which were introduced as a more capable replacement of the AppCache mechanism, can be leveraged for this purpose. They allow Web sites to introduce their own custom caching and offline mechanism, which is based on JavaScript-driven interception of HTTPS requests. For brevity, we omit further technical details on Service Workers and refer the reader to [47] for more information. In the context of the targeted functionality — client-side caching of JavaScript libraries — Service Workers can be used as follows: On the first retrieval of the JavaScript resource, the Service Worker intercepts the associated HTTPS response and persists the code, e.g., using the browser's Cache API or the IndexedDB. From now on, whenever a Web document of the application uses a JavaScript resource that is persisted this way, the Service Worker intercepts the request and directly provides the requested JavaScript. In parallel, after the script code has been provided to the page, the Service Worker can update its cached version asynchronously, thus removing any potential temporary poisoning of the stored data, without affecting the site's performance. As the Service Worker code runs completely separated from the document's JavaScript, this functionality cannot be influenced by the Web Attacker and, as Service Workers are an HTTPS-only feature, the Network Attacker is also rendered harmless. Unfortunately, Service Workers are comparatively new and, thus, browser support is not yet universal[5].

[5]https://caniuse.com/#search=serviceworker

```
var hostname = localStorage.getItem("hostname");
var script = document.createElement("script");
script.src = hostname + "foo.js";
document.body.appendChild(script);
```

Fig. 9. Example vulnerability involving a stored hostname

b) Pure HTML: On eleven domains, we identified HTML fragments in the client-side storage that did not contain any interwoven JavaScript, neither as `script`-tags nor inline event handlers. These cases can be secured by client-side sanitization that allows (harmless) HTML syntax but robustly removes all JavaScript from the code. For instance, the well-established DOMPurify [20] library offers such functionality. Alternatively, structure-based approaches like BLUEPRINT can be used to ensure that only benign markup is used [58].

c) HTML/JavaScript Mix: Five sites in our dataset persisted HTML code that also contained inline JavaScript. In such cases, none of the available defensive coding measures can be applied, as it is not possible to determine which stored JavaScript code originated from an attacker and which from the developer. Hence, securing these sites requires removing the insecure feature altogether.

D. Storage of Configuration Information

Finally, in 28 cases, we observed that hostnames were stored on the client side and then used within the application to reference further resources, as depicted in Figure 9. In general, this pattern appears to be a case of client-side configuration with respect to resource location, e.g, for the purpose of client-side load balancing. To securely implement this functionality, a whitelist check for the retrieved values can be introduced, as the set of legal values is probably bounded. One prime example of a feasible whitelist check is the case of Google's Firebase when using the Realtime Database Feature [16]. On an abstract level, Firebase periodically requests resources from a host which is stored in the Local Storage as depicted in Figure 9. Investigating these cases, we observed that each of the stored hosts was a subdomain of firebaseio.com, thus allowing the library to simply check the second-level domain.

E. Applicability of General Defenses

The risks of a persisted malicious payload being executed code on the client side have been acknowledged by standards bodies. Specifically, the W3C has proposed the Clear-Site-Data response header [64]. This mechanism allows site operators to truncate all client-side storage and moreover shut down all JavaScript contexts to ensure that an attacker-controlled context is unable to re-poison the storage. However, making use of this mechanism regularly inevitably destroys the purpose of having client-side storage, be it code or configuration data storage. A security-aware user can achieve the same effect by completely removing the browser profile before starting the browser or by making use of equivalent browser features. Other research approaches focused on the integrity of cookies by proposing Origin Cookies [4]. These would, given appropriate deployment, prevent exploitation by the Network attacker on all HTTPS origins due to separated cookie storage (analogous to the Local Storage). Additionally, the Web Attacker would need a reflected XSS on the exact origin, not just on the same domain. Browser vendors, however, favor prefixed cookies [65] as opposed to Origin Cookies [5]. These prefixed cookies make sure that secure cookies can only be set from secure origins (i.e., HTTPS origins) and only for a specific origin (i.e., the cookie must have been set without the `Domain` attribute). Although these are implemented in Chrome and Firefox[6], we only found that two of the 5,000 domains we analyzed use these in the first place (and they did not interfere with our attacks). Moreover, this naturally does not impair a Web Attacker and does not help secure Local Storage.

There are promising approaches which revolve around a finer-grained origin construct in the form of either Isolated Origins [52] or Suborigins [62], which would allow developers to further mitigate the risk of XSS exhibited by, e.g., some legacy portion of their Web application. Another approach to tackle the problem of Client-Side XSS is the concept of Trusted Types [8]. Trusted Types require developers to mark any data which contains code intended to be executed in a sink as safe, which allows these sinks to discard any code not marked as safe. This prevents an attacker from exploiting flows which developers intended to carry only data. This concept, however, is only in its early development stages as of now.

VII. DISCUSSION

Here, we discuss limitations of our analysis and give an outlook on potential future work in this research space.

Drawbacks of Dynamic Analysis — As with any dynamic analysis, our approach does not guarantee discovery of the entire application functionality. Specifically, our crawlers do not log in to any application, and have a fixed depth when crawling. Hence, our analysis cannot cover all available pages, and therefore, flows. Moreover, even if our crawlers were able to visit every available URL, the analysis would not guarantee code coverage, given that certain actions may only be triggered by user action. Additionally, our taint tracking engine only covers *explicit* flows. Hence, if an implicit conversion is applied to the input data (e.g., base64 decoding), our engine is not able to track the flow of data anymore, resulting in missed exploitable flaws. Moreover, our analysis cannot account for modifications in the flow of data, e.g., removal of quotes. While a scenario may exist in which such a filtered flow is exploitable, we leave the analysis of such flows to future work.

Exploitability in Current Browsers — While the exploitability of flows originating from storage does not depend on the browser, the susceptibility of a site to a reflected Client-Side XSS varies between browsing engines. Specifically, Firefox automatically encodes all parts of the URL when accessed via the `location` object; Chrome did not do so until version 65[7]. Since version 65, however, the auto-encoding has changed such that the URL fragment is also encoded, meaning that exploits which target an unfiltered and unmodified flow from the fragment to a sink will not be exploitable anymore. To validate the discovered reflected Client-Side XSS flaws, we used an older version of Chromium, which does not encode the fragment. Note, however, that Microsoft's Edge also does

[6]https://www.chromestatus.com/feature/4952188392570880
[7]https://bugs.chromium.org/p/chromium/issues/detail?id=803103

not encode the fragment, meaning *all* discovered flaws are (at least) exploitable in Edge. Moreover, as of this writing, Microsoft has announced that Edge's XSS filter will be removed in upcoming versions, allowing for trivial exploitation [48].

Lower Bound for Web Attacker — Out of the 418 domains with exploitable flows from persistence to sink, 65 could be attacked by a Web Attacker, due to the presence of a *reflected* Client-Side Cross-Site Scripting flaw. The analysis of these reflected XSS cases suffers from the same limitations as our overall analysis. Moreover, any type of Cross-Site Scripting on a site is sufficient to persist the attacker's malicious payload. The most recent WhiteHat Security report indicates that in their work, they discovered an XSS vulnerability in 33% of all analyzed applications [66]. This indicates that the problem is likely more severe than what our analysis highlighted. In addition to XSS flaws caused by buggy application code, browsers also often carry bugs which allow for universal XSS [38], i.e., all sites visited in a certain browser may be attackable. These attacks, along with self-XSS[8], make the threat of Persistent Client-Side XSS even more severe, which is why we report the total number of vulnerable flows as well as the end-to-end exploitable flaws.

Higher-Order Flows — In our work, we did not find any flow originating from a URL to persistent storage, which could, in turn, be leveraged to exploit a flow from storage to a sink. However, considering the data shown in the last two rows of Table I, we find that there are 98M data flows within cookies and Local Storage, as well as over 2.7M flows between cookies and Local Storage (in either direction). Similarly, in a separate analysis, we found numerous flows between Local and Session Storage. Since we did not consider Session Storage in our analysis, these numbers are omitted from our results. However, these intra- and inter-storage flows indicate that depending on the application, there may be flows which traverse the different persistence APIs. Hence, investigating such more involved flows is a promising area for future work.

Domain Relaxation — To determine exploitability of an origin, we checked whether an unfiltered flow from persistent storage occurred on the same origin as a reflected XSS. It must be noted, though, that JavaScript documents may use a process dubbed *Domain Relaxation*. This involves two documents, which share a common parent domain, setting their `document.domain` property to the common parent [67]. Now, the documents operate under the same origin, i.e., can access each other. Both sides have to explicitly opt into this. However, this means that if a site that has a flow from Local Storage to a sink relaxes its origin to, e.g., the second-level domain (SLD), an XSS on *any* sub-domain of this SLD is sufficient for an attack. The attacker can simply set the `document.domain` property accordingly, allowing her to access the Local Storage of the other document. We believe that this warrants further investigation as part of future work.

Responsible Disclosure — Several vulnerabilities we discovered were caused by third-party code. We notified those parties which were responsible for at least three vulnerable domains. As of this writing, the four largest providers have acknowledged the issues and/or deployed fixes for the flawed code. Additionally, we tried to disclose the vulnerabilities to those sites for which a centralized reporting system like Hackerone can be used, as these promise an increased success rate over attempting direct notification [15, 35, 56]. Unfortunately, our reports were denied due to incorrectly being classified as self-XSS or flagged as out of scope since our report mentioned a potential Network Attacker, thus they never reached the affected party.

[8]https://www.facebook.com/help/246962205475854

VIII. Related Work

Our work touches on different areas of Web Security. We discuss four areas and how our work relates to them, specifically works which investigated the lack of integrity in client-side data storage, the detection of Cross-Site Scripting flaws as well as their mitigation, and works investigating dangerous patterns on the Web.

A. Lack of Integrity in Client-Side Storages

Although no works so far have developed a methodology to investigate the prevalence of Persistent Client-Side XSS in the wild, a number of papers have hinted at potential dangers related to the lack of integrity for cookies and Local Storage. The potential attack surface of trusting persisted code was hinted at with the introduction of Local and Session Storage to the Web by Hanna et al. [17], who investigate security issues related to new Web APIs. In contrast to our work, the authors only hinted at XSS as a side note, instead investigating cases in which data under the control of an attacker might alter applications' states. As such, they manually analyzed 11 applications to find flaws related to APIs such as WebSQL, using the same, well-established attacker models we apply. However, our work focuses on finding verified, exploitable vulnerabilities at a much larger scale.

In 2012, Lekies and Johns [31] applied simple heuristics to survey the use of Local Storage, concluding that there are cases in which HTML or JavaScript code is stored in Local Storage. To mitigate the associated risks, the authors propose a JavaScript-based solution which wraps Local Storage functionality and checks the integrity of items before they are returned from the storage API. Although the authors did not evaluate the practicability of real-world attacks, the general idea of the defense mechanism is applicable to a multitude of our observed cases. Their integrity checks, however, require changes to the applications codebase. In contrast, for situations in which Service Workers can be used, the applications can simply be moved into the Service Worker, allowing for easy integration and therefore providing a better fit to ensure code integrity on the client side.

In 2015, Zheng et al. [68] analyzed the general risks associated with the lack of integrity of cookie data. Their threat model also covers a Network and Web Attacker, which allows them to find instances of session hijacking, history stealing, and even XSS flaws. From their discussion of these flaws, however, it remains unclear whether these were caused by insecure server- or client-side code. In contrast to their work, we use an end-to-end methodology to detect flows from cookies (and Local Storage), and subsequently, use that information to automatically generate payloads to verify the exploitability of the observed flows.

B. Cross-Site Scripting Detection

At its inception, Cross-Site Scripting was thought to be a pure server-side issue, which is why a large fraction of work focused on the server-side portion of Web applications. Early work conducted an analysis of server-side XSS using data flow analysis [24], allowing the authors to find 15 previously unknown vulnerabilities. Follow-up work from Balzarotti et al. [2] investigated the sanitization process of user input, showing that common sanitization techniques such as regular expressions, blacklists, and string manipulation are often incorrectly implemented, leaving the application vulnerable.

In 2005, Klein [26] unveiled what we now know as Client-Side Cross-Site Scripting, which unlike its server-side counterpart is exhibited purely in the victim's browser. This led to work by Saxena et al. [49], in which they used taint-enhanced black box fuzzing in order to find injection vulnerabilities in JavaScript code. In 2013, Lekies et al. [32] presented the first automated, large-scale analysis of Client-Side Cross-Site Scripting vulnerabilities. The authors built a taint-aware Chromium, allowing them to discover and verify the exploitability of 10% of the Alexa top 5,000 sites. Their approach was recently picked up by Melicher et al. [36], who improved the exploit generation process, allowing for 83% more exploits to be found when directly compared with the techniques of Lekies et al. Our analysis follows the selection of domains and crawling parameters (such as crawl depth) of the former study, to allow a reasonable comparison between the threats of persistent and reflected Client-Side XSS. As discussed in Section IV, we extend their approach to fix shortcomings related to context-aware break-out/break-in sequences, and issues related to matching and replacing storage items. Moreover, their work only focused on the detection of *reflected* XSS, whereas we investigate *persistent* XSS. Building on the findings of Lekies et al. [32], Stock et al. [55] investigated the complexity of the uncovered vulnerable flows, concluding that a large fraction of flows are actually rather simple in nature. In more recent work, Stock et al. [57] also showcased a study of the prevalence of reflected Client-Side Cross-Site Scripting over the last 20 years, using the Internet Archive. They showed that even 8 years before Klein's first mention of the new class of Cross-Site Scripting in his blog post [26], Web sites were susceptible to reflected Client-Side XSS. Our reflected XSS results confirm all these prior findings.

C. Cross-Site Scripting Mitigation

Based on our findings of real-world flaws, we presented a number of secure alternatives for using persisted data. Another approach to tackle the specifics of Client-Side XSS is extending the taint tracking approach from Lekies et al. [32] to the parser level. In doing so, Stock et al. [54] were able to prevent tainted values from being interpreted as code, thus stopping all previously verified cases of reflected Client-Side XSS. Extending this to storage, however, would not work, given the inability to distinguish attacker-injected payloads in the storages, resulting either in false negatives or impaired functionality. The general idea of this solution was since refactored into Trusted Types [8], which prevent undesired code flows into code executing sinks, but have yet to be introduced into browsers.

In the more general area of XSS mitigations, Chrome has a built-in reflected XSS filter [3]. Moreover, CSP promises to mitigate the impact of XSS. This is, in its basic form, enforced by only allowing resources to be included from whitelisted hosts and preventing the use of inline scripting [60]. However, real-world adoption lags behind, given the grave changes required for legacy applications. In 2013, Doupé et al. [14] presented an automated tool which separates code and data in ASP.net applications. In 2014, Weissbacher et al. [63] showed that CSP is only adopted by a minuscule fraction of the most important sites and that—even if a policy is present—it actually does not prevent content injection attacks as intended, even though an automated generation of such policies is possible [42]. This trend was also observed by Weichselbaum et al. [61]. Analyzing 26,000 unique CSP policies, they were able to show that around 95% of these were circumventable. Also, they proposed the concept of `strict-dynamic`. Based on similar findings from Calzavara et al. [6], Calzavara et al. [7] proposed an extension to CSP which allows for dynamic policy composition. Even in the presence of CSP, Lekies et al. [34] showed that its protective capability can be circumvented with script gadgets.

D. Dangerous Patterns on the Web

Our work follows a long line of papers investigating different dangerous patterns on the Web. In 2011, Richards et al. [45] conducted an analysis of the dangerous use cases of JavaScript's `eval`, concluding that a wide variety of uses can be replaced with more secure alternatives, thus preventing potential vulnerabilities altogether. Similarly to them, we were able to investigate various cases in which developers were unable to achieve their goal in a secure fashion, although a secure solution exists. Nikiforakis et al. [41] investigated third-party inclusions by analyzing the Alexa top 10,000 domains. The authors were able to show the dangers of third-party inclusions and also present several attacks targeting faulty inclusions. Another dangerous practice which is tied to this development is the inclusion of outdated libraries. Lauinger et al. [30] conducted a large-scale analysis of the library inclusion behavior of around 60,000 domains. The authors were able to show that more than 37% of the investigated domains make use of vulnerable libraries and that a single HTML document may even include two different versions of a given library; results aligning with Stock et al. [57].

Lekies et al. [33] analyzed the danger of Cross-Site Script Inclusion vulnerabilities on 150 high-ranked domains, finding 40 vulnerable Web applications which allow an attacker to exfiltrate sensitive data, e.g., access tokens. These could in turn be used to conduct targeted XSS attacks. The privacy dangers of HTTP were highlighted by Sivakorn et al. [51] through an analysis of information leakage of cookies in the presence of a passive Network Attacker, concluding that 15 of 26 top-ranked domains were, in fact, leaking sensitive user data due to the lack of deployed HSTS. Moreover, Mendoza et al. [37] investigated differences between desktop and mobile versions of Web applications, showing grave inconsistencies between deployed security headers, giving attackers an increased attack surface. In contrast to classical XSS injections, Arshad et al. [1] presented the first large-scale analysis of Relative Path Overwrite flaws, which allow an attacker to inject style directives into a Web application, enabling a scriptless attack [18].

IX. Conclusion

Cross-Site Scripting has been extensively researched in the past, ranging from the reflected and persistent variants on the server in the early days, followed by their reflected counterpart on the client side in more recent work. In these works, the persistent variant of Client-Side Cross-Site Scripting has been noted, but has not been investigated to understand its actual prevalence in the modern Web. We close this research gap by delivering factual evidence that the neglected type of Persistent Client-Side XSS is indeed an actionable threat to modern Web applications, by providing the first systematic study of its prevalence in the wild. With a fully automated pipeline using taint-aware browsing engines and enhanced exploit generation and validation schemes, we show that more than 8% of the 5,000 highest-ranked sites exhibit exploitable data flows from client-side storage to code-executing sinks. If we consider only those domains which make use of data from persistent storage in their client-side code, the fraction of flawed domains rises to 21%, highlighting that developers tacitly assume that the integrity of the stored information is not compromised. Moreover, we show that for two attacker models, i.e., Network and Web Attackers, this problem can be exploited in more than 70% of domains which use persisted data in an insecure fashion. Our findings indicate *four* distinct scenarios in which client-side storage is insecurely used, allowing us to propose application-layer mitigations while at the same time showcasing the lack of more general and at the same time usable defense mechanisms. Thus, our results highlight that contrary to popular belief, XSS is not yet a solved vulnerability class, nor is it understood in its entirety, demonstrating the need for appropriate countermeasures against all of its flavors.

Acknowledgements

We would like to thank the anonymous reviewers for their feedback. 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) and has received funding from the EU's Horizon 2020 research and innovation programme under grant agreement No 700178 ("SISSDEN"). This research was supported by the Lower Saxonian Ministry for Science and Culture as part of the research program MOBILISE (Mobility in Engineering and Science).

References

[1] S. Arshad, S. A. Mirheidari, T. Lauinger, B. Crispo, E. Kirda, and W. Robertson, "Large-scale analysis of style injection by relative path overwrite," in *WWW*, 2018.

[2] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Saner: Composing static and dynamic analysis to validate sanitization in web applications," in *IEEE S&P*, 2008.

[3] D. Bates, A. Barth, and C. Jackson, "Regular expressions considered harmful in client-side xss filters," in *WWW*, 2010.

[4] A. Bortz, A. Barth, and A. Czeskis, "Origin cookies: Session integrity for web applications," *W2SP*, 2011.

[5] bsittler, "What about origin cookies?" 2016. [Online]. Available: https://github.com/WICG/cookie-store/issues/6

[6] S. Calzavara, A. Rabitti, and M. Bugliesi, "Content security problems?: Evaluating the effectiveness of content security policy in the wild," in *CCS*, 2016.

[7] ——, "CCSP: Controlled Relaxation of Content Security Policies by Runtime Policy Composition," in *USENIX Security*, 2017.

[8] W. I. CG, "Trusted types for dom manipulation," 2017. [Online]. Available: https://github.com/WICG/trusted-types

[9] Chromium Team, "Hsts preload list submission," 2018. [Online]. Available: https://hstspreload.org/

[10] Cloudflare, "Website Optimization," https://www.cloudflare.com/en/website-optimization/, 2018.

[11] Coinhive, "Coinhive – monero javascript mining," 2018. [Online]. Available: https://coinhive.com/

[12] J. Dahse and T. Holz, "Static detection of second-order vulnerabilities in web applications," in *USENIX Security*, 2014.

[13] S. Di Paola, "DominatorPro: Securing Next Generation of Web Applications," https://dominator.mindedsecurity.com/, 2012.

[14] A. Doupé, W. Cui, M. H. Jakubowski, M. Peinado, C. Kruegel, and G. Vigna, "dedacota: toward preventing server-side xss via automatic code and data separation," in *CCS*, 2013.

[15] M. Finifter, D. Akhawe, and D. Wagner, "An empirical study of vulnerability rewards programs." in *USENIX Security*, 2013.

[16] Google, "Firebase Realtime Database," https://firebase.google.com/docs/database/, 2018.

[17] S. Hanna, R. Shin, D. Akhawe, A. Boehm, P. Saxena, and D. Song, "The emperor's new apis: On the (in) secure usage of new client-side primitives," in *W2SP*, 2010.

[18] M. Heiderich, M. Niemietz, F. Schuster, T. Holz, and J. Schwenk, "Scriptless attacks: stealing the pie without touching the sill," in *CCS*, 2012.

[19] M. Heiderich, J. Schwenk, T. Frosch, J. Magazinius, and E. Z. Yang, "mxss attacks: Attacking well-secured web-applications by using innerhtml mutations," in *CCS*, 2013.

[20] M. Heiderich, C. Späth, and J. Schwenk, "Dompurify: Client-side protection against xss and markup injection," in *ESORICS*, 2017.

[21] I. Hickson, "Web storage (second edition)," 2016. [Online]. Available: https://www.w3.org/TR/2016/REC-webstorage-20160419/

[22] J. Hodges, C. Jackson, and A. Barth, "RFC6797: Http strict transport security (hsts)," November 2012.

[23] A. Janc, "Rootkits in your web application," Talk at 28c3, 2011. [Online]. Available: https://media.ccc.de/v/28c3-4811-en-rootkits_in_your_web_application

[24] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A static analysis tool for detecting web application vulnerabilities," in *IEEE S&P*, 2006.

[25] A. Kieyzun, P. J. Guo, K. Jayaraman, and M. D. Ernst, "Automatic creation of sql injection and cross-site scripting attacks," in *ICSE*, 2009.

[26] A. Klein, "DOM based cross site scripting or XSS of the third kind," *Web Application Security Consortium, Articles*, 2005.

[27] M. Kranch and J. Bonneau, "Upgrading https in mid-air: An empirical study of strict transport security and key

pinning," in *NDSS*, 2015.

[28] D. Kristol and L. Montulli, "RFC2965: Http state management mechanism," October 2000.

[29] C. Kruegel, E. Kirda, and S. McAllister, "Leveraging user interactions for in-depth testing of web applications," in *RAID*, 2008.

[30] T. Lauinger, A. Chaabane, S. Arshad, W. Robertson, C. Wilson, and E. Kirda, "Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web," in *NDSS*, 2017.

[31] S. Lekies and M. Johns, "Lightweight integrity protection for web storage-driven content caching," in *Web 2.0 Security & Privacy (W2SP)*, 2012.

[32] S. Lekies, B. Stock, and M. Johns, "25 million flows later: large-scale detection of DOM-based xss," in *CCS*, 2013.

[33] S. Lekies, B. Stock, M. Wentzel, and M. Johns, "The unexpected dangers of dynamic javascript," in *USENIX Security*, 2015.

[34] S. Lekies, K. Kotowicz, S. Groß, E. A. Vela Nava, and M. Johns, "Code-reuse attacks for the web: Breaking cross-site scripting mitigations via script gadgets," in *CCS*, 2017.

[35] F. Li, Z. Durumeric, J. Czyz, M. Karami, M. Bailey, D. McCoy, S. Savage, and V. Paxson, "You've got vulnerability: Exploring effective vulnerability notifications." in *USENIX Security*, 2016.

[36] W. Melicher, A. Das, M. Sharif, L. Bauer, and L. Jia, "Riding out DOMsday: Toward detecting and preventing DOM cross-site scripting," in *NDSS*, 2018.

[37] A. Mendoza, P. Chinprutthiwong, and G. Gu, "Uncovering http header inconsistencies and the impact on desktop/mobile websites," in *WWW*, 2018.

[38] V. Metnew, "uxss-db," 2018. [Online]. Available: https://github.com/Metnew/uxss-db

[39] M. D. Network, "HTTP cookies," 2018. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies

[40] ——, "Object.defineproperty()," 2018. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty

[41] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. V. Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna, "You are what you include: large-scale evaluation of remote javascript inclusions," in *CCS*, 2012.

[42] X. Pan, Y. Cao, S. Liu, Y. Zhou, Y. Chen, and T. Zhou, "Cspautogen: Black-box enforcement of content security policy upon real-world websites," in *CCS*, 2016.

[43] I. Parameshwaran, E. Budianto, S. Shinde, H. Dang, A. Sadhu, and P. Saxena, "Auto-patching dom-based xss at scale," in *FSE*, 2015.

[44] L. Poinsignon, "Bgp leaks and cryptocurrencies," 2018. [Online]. Available: https://blog.cloudflare.com/bgp-leaks-and-crypto-currencies/

[45] G. Richards, C. Hammer, B. Burg, and J. Vitek, "The eval that men do," in *ECOOPS*, 2011.

[46] I. Roberts, "Browser cookie limits," 2018. [Online]. Available: http://browsercookielimits.squawky.net/

[47] A. Russell, J. Song, J. Archibald, and M. Kruisselbrink, "Service Workers," W3C Working Draft, 2 November 2017, https://www.w3.org/TR/service-workers-1/, 2017.

[48] D. Sarkar and B. LeBlanc, "Announcing windows 10 insider preview build," 2018. [Online]. Available: https://blogs.windows.com/windowsexperience/2018/07/25/announcing-windows-10-insider-preview

[49] P. Saxena, S. Hanna, P. Poosankam, and D. Song, "Flax: Systematic discovery of client-side validation vulnerabilities in rich web applications." in *NDSS*, 2010.

[50] D. Silver, S. Jana, D. Boneh, E. Y. Chen, and C. Jackson, "Password managers: Attacks and defenses." in *USENIX Security*, 2014, pp. 449–464.

[51] S. Sivakorn, I. Polakis, and A. D. Keromytis, "The cracked cookie jar: Http cookie hijacking and the exposure of private information," in *IEEE S&P*, 2016.

[52] E. Stark, T. Vyas, D. Ross, M. West, and J. Weinberger, "Isolated origins," 2017. [Online]. Available: https://wicg.github.io/isolation/

[53] B. Stock and M. Johns, "Protecting users against xss-based password manager abuse," in *siaCCS*, 2014.

[54] B. Stock, S. Lekies, T. Mueller, P. Spiegel, and M. Johns, "Precise client-side protection against DOM-based cross-site scripting," in *USENIX Security*, 2014.

[55] B. Stock, S. Pfistner, B. Kaiser, S. Lekies, and M. Johns, "From facepalm to brain bender: Exploring client-side cross-site scripting," in *CCS*, 2015.

[56] B. Stock, G. Pellegrino, C. Rossow, M. Johns, and M. Backes, "Hey, you have a problem: On the feasibility of large-scale web vulnerability notification." in *USENIX Security*, 2016.

[57] B. Stock, M. Johns, M. Steffens, and M. Backes, "How the web tangled itself: Uncovering the history of client-side web (in) security," in *USENIX Security*, 2017.

[58] M. Ter Louw and V. Venkatakrishnan, "Blueprint: Robust prevention of cross-site scripting attacks for existing browsers," in *IEEE S&P*, 2009.

[59] J. Troup, "Ubuntu forums are back up and a post mortem," 2013. [Online]. Available: http://blog.canonical.com/2013/07/30/ubuntu-forums-are-back-up-and-a-post-mortem/

[60] W3C, "Content Security Policy Level 3 ," https://www.w3.org/TR/CSP3/, 2016.

[61] 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 *CCS*, 2016.

[62] J. Weinberger, D. Akhawe, and J. Eisinger, "Suborigins," 2017. [Online]. Available: https://w3c.github.io/webappsec-suborigins/

[63] M. Weissbacher, T. Lauinger, and W. Robertson, "Why is csp failing? trends and challenges in csp adoption," in *RAID*, 2014.

[64] M. West, "Clear site data," 2017. [Online]. Available: https://www.w3.org/TR/clear-site-data/

[65] ——, "Cookie prefixes," 2016. [Online]. Available: https://tools.ietf.org/html/draft-ietf-httpbis-cookie-prefixes-00

[66] WhiteHat Security, "Application security statistics report," 2017. [Online]. Available: https://goo.gl/ULdVvC

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

[68] X. Zheng, J. Jiang, J. Liang, H.-X. Duan, S. Chen, T. Wan, and N. Weaver, "Cookies lack integrity: Real-world implications." in *USENIX Security*, 2015.