2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER) | 978-1-7281-9630-5/20/$31.00 ©2021 IEEE | DOI: 10.1109/SANER50967.2021.00023

XSnare: Application-specific client-side cross-site scripting protection

José Carlos Pazos, Jean-Sébastien Légaré, and Ivan Beschastnikh
Department of Computer Science
University of British Columbia
{jpazos, jslegare, bestchai}@cs.ubc.ca

***Abstract*—We present XSnare, a client-side Cross-Site Scripting (XSS) solution implemented as a Firefox extension. The client-side design of XSnare can protect users before application developers release patches and before server operators apply them.**

XSnare blocks XSS attacks by using previous knowledge of a web application's HTML template content and the rich DOM context. XSnare uses a database of exploit descriptions, which are written with the help of previously recorded CVEs. It singles out injection points for exploits in the HTML and dynamically sanitizes content to prevent malicious payloads from appearing in the DOM. XSnare displays a secured version of the site, even if is exploited.

We evaluated XSnare on 81 recent CVEs related to XSS attacks, and found that it defends against 93.8% of these exploits. To the best of our knowledge, XSnare is the first protection mechanism for XSS that is application-specific, and based on publicly available CVE information. We show that XSnare's specificity protects users against exploits which evade other, more generic, XSS defenses.

Our performance evaluation shows that our extension's overhead on web page loading time is less than 10% for 72.6% of the sites in the Moz Top 500 list.

I. Introduction

Cross-Site Scripting (XSS) is still one of the most dominant web vulnerabilities. A 2017 report showed that 50% of websites contained at least one XSS vulnerability [1]. Countermeasures exist, but many of them lack widespread deployment, and so web users are still mostly unprotected.

Informally, the cause of XSS is a lack of input validation: user-chosen data "escapes" from the page's template and makes its way into the JavaScript engine, or modifies the Document Object Model (DOM). Consequently, many of the XSS defenses published so far propose to fix the problem at the source, by properly separating the template from the user data on the server, or by modifying browsers [2], [3], [4], [5], [6]. There are also similar solutions that can be implemented in the front-end code of an application [7]. In all cases, these technologies must be adopted by the application software developers, otherwise users are left unprotected.

One barrier to adoption of existing XSS defenses is that developers may not have the necessary expertise, or sufficient resources, to use the approach. Luckily, users wishing to gain reassurance over the safety of the sites they visit can install browser extensions to filter malicious scripts and content. Unfortunately, some of the most popular of these extensions, like NoScript [8], achieve most of their security by disabling functionality, such as JavaScript. Doing so unfortunately impairs usability[1]. A study by Snyder et al. [10] showed that browser security can be increased by disabling some rarely used JavaScript APIs, largely retaining usability. Our work builds on this idea, retaining website usability after an exploit is disabled.

When an XSS vulnerability is disclosed, some software vendors respond with patches. If the affected software is released in the form of packages, frameworks, or libraries, and used by several web applications, there is delay before users can benefit from the patch. Most importantly, the patched software must be re-deployed by site administrators.

Unfortunately, website administrators will not, and often cannot, apply software updates immediately: one study found that 61% of WordPress websites were running a version with known security vulnerabilities [11]. In another report, we learn that 30.95% of Alexa's top 1 Million sites run a vulnerable version of WordPress [12].

Users are at the mercy of developers and administrators if they want to access safe, up-to-date, applications. Our solution, XSnare, helps with this problem – based on information from past disclosures, XSnare patches known page vulnerabilities *directly in the browser*.

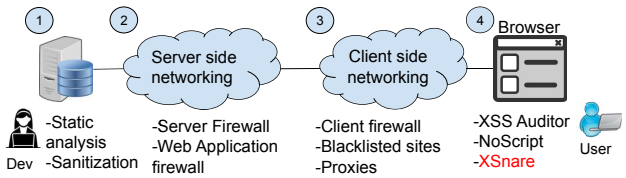

Fig. 1: Different web security solutions with XSnare on the client-side.

Each layer of the web application stack (Figure 1) presents different options to defend against XSS. Note that solutions at different layers are often complementary:

1) The application logic is the first line of defence. Code safety can be enhanced with third-party vulnerability scanning solutions, and a thorough code-review process. Taint, and static code analysis tools can detect unsanitized inputs.

[1]As early as 2012, JavaScript was used by almost 100% of the Alexa top 500 sites [9]

978-1-7281-9630-5/21/$31.00 ©2021 IEEE
DOI 10.1109/SANER50967.2021.00023

2) In the hosting environment, network firewalls, specifically Web Application Firewalls (WAFs) can defend against attacks such as DDoS, SQL injections and XSS.
3) In the client's environment (residential or commercial), users may install network firewalls, network content filters, and web proxies.
4) The last line of defence is the browser. Browser have built-in defences, such as Chrome's XSS Auditor [13]. Users can also install third-party extensions to block malicious requests and responses, such as NoScript [8], and XSnare.

We make two observations about existing solutions: (a) server-side solutions have to be applied independently on each server, and (b) solutions on the client are typically written as generic filters which attempt to catch everything, and consequently do not take full advantage of the specificity of the application or the vulnerability.

For example, a WAF can effectively protect the users, but users cannot realistically expect every site to be protected by a WAF. At the opposite end, in the client's environment, a user might configure a network proxy for all website traffic, with generic rules achieving maximum coverage, but this will often lead to an elevated rate of false positives (FPs).

Similarly, browser built-in defences are coarse-grained, and work on just a subset of exploits. Chrome's XSS Auditor, for example, only attempts to defend against reflected XSS. Google recently announced its intention to deprecate XSS Auditor, for reasons including *"Bypasses abound"*, *"It prevents some legit sites from working"*, and *"Once detected, there's nothing good to do"* [14]. Stock et al. [15] propose enhancements to XSS Auditor and cover a wider range of exploits than the auditor, but are limited to DOM-based XSS. By contrast, our work covers all types of XSS.

Implementing adequate server-side protections requires time [16], [17], [18], [19]. A 2018 study found that the average time to patch a known exploit in the form of a Common Vulnerability and Exposures (CVE), all severities combined, is 38 days, increasing to as much as 54 days for low severity CVEs, and the oldest unpatched CVE was 340 days old [20].

Server-side defences also rarely protect against client-only forms of XSS, e.g., reflected XSS, or persistent client-side XSS, which use a browser's local storage or cookies as an attack vector. Steffens et al. [21] present a study of persistent client-side XSS across popular websites and find that as many as 21% of the most frequented web sites are vulnerable to these attacks. To provide users with the means to protect themselves in the absence of control over servers, we believe that a client-side solution is necessary.

A number of existing solutions in this area also suffer from high rates of false-positives and false-negatives. For example, NoScript [8] works via domain allow-listing: by default, JavaScript and other code will not execute. However, not all scripts outside of the allowlist should be assumed to be malicious. Browser-level filters like XSS Auditor use general policies and can incorrectly sanitize non-malicious content.

We posit that the DOM is the right place to mitigate XSS attacks as it provides a full picture of the web application. While most of the functionality we provide could be done by a network filter for the browser, we take advantage of additional browser context. Particularly, when an exploit occurs as a result of user interactions, like in response to a click, our approach benefits from knowing the initiating tab to filter the response. Previous client-side solutions have opted for detectors that were generic and site-agnostic [22], [3], [23]. Our work goes in the opposite direction, and tries to instead prevent precisely-defined exploits in specific applications.

If a patch for a server-side vulnerability can be "translated" into an equivalent set of operations to apply on the fully formed HTML document in the browser, then we seize the opportunity to defend *early* against exploits of that vulnerability. Our extension, which has access to the user's browsing context, can identify vulnerable pages based on a database of signatures for previous disclosures. This way, XSnare can protect users as soon as a patch is implemented and added to its database. The client-side patch will remain beneficial until *all* server operators running that software have had a chance to upgrade their deployments.

A similar philosophy is adopted by the client-side firewall-based network proxy Noxes [22]. However, due to their position in the stack, these policies do not defend against attacks invisible to the network, e.g., deleting local files.

Our system's signatures are designed to be application-specific, both in terms of exploit detection and sanitization. Application-specific signatures accurately dispose of exploits while retaining the web site's usability.

We evaluate XSnare by testing it on 81 recent XSS CVEs. We also report XSnare's performance overhead on page load times across a wide range of sites and show that it does not significantly impact the user's browsing experience.

To summarize, our contributions include:

- XSnare: a novel client-side framework that protects users against XSS vulnerabilities with a database of signatures for these vulnerabilities, written in a declarative language.
- A mechanism to correctly isolate a vulnerable injection point in a web page and to apply the intended server-side patch on the client-side.
- A collection of signatures to protect users against real XSS CVEs (Section V), demonstrating the practicality of XSnare; and the evaluation of its impact on browsing (Section VI).

II. XSnare Design

We now present the design of XSnare and its components (Figure 2). We begin by reviewing our threat model.

A. Threat model

Our work makes no assumptions about the web server. In particular, the server may run out of date and vulnerable software that delivers pages to the user's browser with XSS exploits.

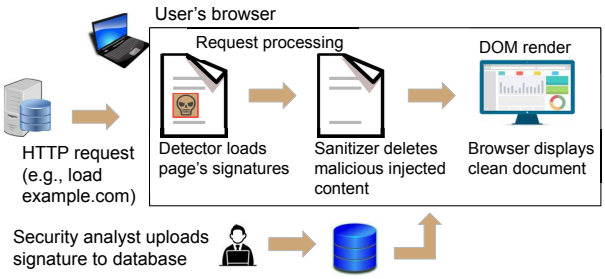

Fig. 2: XSnare's approach to protect against XSS.

We trust the browser and the browser's extension mechanism to correctly execute XSnare. We also depend on the browser to disallow malicious tampering with the client-side signature database.

We trust the analyst who writes the signature definitions used by XSnare. For XSnare to be effective the signatures must be correct. However, a signature that fails to match a vulnerability, will only impact the page with longer load times.

B. Overview

We now review the high-level operation of XSnare with Figure 2. A user requests a page, *example.com*, on a browser with the XSnare extension installed. The response may or may not contain malicious XSS payloads. Before the browser renders the document, XSnare analyzes the potentially malicious document. The extension loads signatures from its local database into its detector. The detector analyzes the HTML string arriving from the network, and identifies the signatures which apply to the document. These signatures specify one or more "injection spots" in the document, which correspond, roughly speaking, to regions of the DOM where improperly sanitized content could be injected. The extension's sanitizer eliminates any malicious content and outputs a clean HTML document to the browser for rendering.

C. An example application of XSnare

To further explain our approach, we present a small example of how HTML context can be used to defend against XSS, taken from CVE 2018-10309 [24]. This is reproducible in an off-the-shelf WordPress installation running the Responsive Cookie Consent plugin, v1.7. This is a stored XSS vulnerability, and as such is not caught by some generic client-side XSS filters, including Chrome's XSS auditor.

Consider a website running PHP on the backend which stores user input from one user, and displays it later to another user, inside an **input** element.

The PHP code defines the static HTML template (in black), as well as the dynamic input (in red):

```
<input id="rcc_settings[border-size]"
name="rcc-settings[border-size]"
type="text" value="<?php rcc_value('border-size');
    ?>"/>
<label class="description"
for="rcc_settings[border-size]">
```

Normally, the **input** might have a value of "0":

```
<input id="rcc_settings[border-size]"
 name="rcc-settings[border-size]"
 type="text" value="0">
<label class="description"
 for="rcc_settings[border-size]">
```

However, the php code is vulnerable to an injection attack:

```
border-size = ""><script>alert('XSS')</script>
```

The browser will render this, executing the injected script:

```
<input id="rcc_settings[border-size]"
name="rcc-settings[border-size]"
type="text" value=""><script>alert('XSS')</script>
<label class="description"
for="rcc_settings[border-size]">
```

Note that the resulting HTML is well-formed, so a mere syntactic check will not detect the malicious injection. Let us assume a security analyst knows the original template, i.e., without injected content. If the analyst were given a filled-in document, they could (in most cases) separate the injected content from the server-side template, and get rid of the malicious script entirely, using proper sanitization.

The injected script is bounded by template elements with identifiable attributes. Assuming (for now) that there is only one such vulnerable injection point, we can search for the **input** element from the top of the document, and the **label** from the bottom to ensnare the injection points in the HTML.

This shares goals with the client/server hybrid approach of Nadji et al. [4]. They automatically tag injected DOM elements on the server-side using a taint-tracking, so that a modified browser can reliably separate template vs injected content. We do not require any server-side modifications, but rather opt for a client-side tagging solution based on exploit definitions.

The injected content, once identified, must be sanitized appropriately. The appropriate action will depend on the application setting, but assuming a patch has been written, it suffices to translate the intention in the server code's path to the client-side. This is straightforward once the fix is understood.

The developer incorrectly claimed the bug had been fixed in version 1.8 of the plugin. Other similar vulnerabilities had indeed been fixed, but not this one [25]. The built-in WordPress function sanitize_text_field needed to be applied.

XSnare does not automatically determine the actions to implement from a patch. We assign this task to a security analyst, who acts as the signature developer for an exploit. The system automates signature matching and sanitization.

D. XSnare Signatures

Our signature definitions make two assumptions: first, **an injection must have a start point and end point**, that is, an element can only be injected between a specific HTML node and its immediate sibling in the DOM tree; second, in a well-formed DOM, **the dynamic content will not be able to rearrange its location in the document without JavaScript execution** (e.g., removing and adding elements), allowing us to isolate it from the template.

Pages commonly contain more than one vulnerable injection point. We discuss the difficulty of supporting these pages in Section II-G.

We believe CVEs are an ideal source of signature definitions. Previous client-side work does not benefit from our level of specificity; these tools often use less accurate heuristics to detect exploits. XSnare signatures are written specifically to match a known software vulnerability. Even though they cannot be generated automatically, any trusted developer can write and compose these signatures. This does not require participation from application developers.

In general, we do not require the existence of a publicly disclosed CVE to be able to write a signature for an exploit. CVEs have been useful to us as we did not discover the exploits. A knowledgeable analyst can write a signature without a public CVE. In fact, for security measures, many CVEs are not publicly available until the application developer has patched its software. Our system can help reduce the time between zero day attacks and patch deployment: an analyst can write a signature for a vulnerability as soon as they know the issue.

Long term, we imagine that volunteers (or entrepreneurs) would cultivate and maintain the signature database. New signatures could be contributed by a community of amateur or professional security analysts, in a manner not so different from how antispam or antivirus software is managed. The popular ad blocking extension AdBlock, for example, relies on filter rules taken from open-source filter lists [26].

The challenge of automatically deriving signatures from detailed CVEs is challenging, albeit outside the scope of this paper.

E. Firewall Signature Language

Our signature language needs enough power of expression for the signature writer to be precise, both for determining the correct web application and to identify the affected areas in the HTML. For injection point isolation, a language based on regular expressions suffices to express precise sections of the HTML. The following is the signature that defends against the motivating example of Section II-C (For an in-depth description of the signature language, please see [27]):

Listing 1: An XSnare signature

```
url:
    'wp-admin/options-general.php?page=rcc-settings',
software: 'WordPress',
softwareDetails: 'responsive-cookie-consent',
version: '1.5',
type: 'string',
typeDet: 'single-unique',
sanitizer: 'regex',
config: '/^[0-9](\.[0-9]+)?$/',
endPoints:
['<input id="rcc_settings[border-size]"
    name="rcc_settings[border-size]" type="text"
  value="',
'<label class="description"
for="rcc_settings[border-size]">']
```

In summary, a signature will have the necessary information to determine whether a loaded page has a vulnerability, and specify appropriate actions for eliminating any malicious payloads.

Analysts configure their signatures with one function chosen from the static set of sanitization functions offered by XSnare (Section III-B). These functions inoculate potentially malicious injections based on the DOM context surrounding the injection. The goal of signatures is to provide such sanitization, ideally without "breaking" the user experience of the page. The default function preset is DOMPurify's [7] default configuration, which takes care of common sanitization needs [28]. However, DOMPurify's defaults can be unnecessarily restrictive, or not restrictive enough, in which case the other sanitization methods are preferable.

We considered allowing arbitrary sanitization code in signatures. While it would open complex sanitization possibilities, we have decided against it, principally for security reasons. The minimal set of functions we settled on also sufficed to express all of the signatures defined for this paper.

F. Browser Extension

Our system's main component is a browser extension which rewrites potentially infected HTML into a clean document. The extension detects exploits in the HTML by using signature definitions and maintains a local database of signatures. We leave the design of an update mechanism to future work, but in its current form, the database is bundled with each new installation of the extension.

The extension translates signature definitions into patches that rewrite incoming HTML on a per-URL basis, according to the top-down, bottom-up scan described in Section II-C.

The extension's detector acts as an in-network filter. We initially considered other designs but quickly found out that applying the patch at the network level was necessary for sanitization correctness: even before any JavaScript code runs, the browser's built-in DOM parser can rearrange elements into an unexpected order, making our extension sanitize the wrong spot. Consider the following example, where an element inside a <tr> tag is rearranged after parsing the string:

```
<table class="wp-list-table">
  <thead>
    <tr>
      <th></th>
      <img src="1" onerror="alert(1)">
      <th>
        <form method="GET" action=""> ...
```

In this HTML, the signature developer might identify the exploit as occurring inside the given table. However, if we wait until the string has been parsed into a DOM tree to sanitize, the elements are rearranged due to <tr> not allowing an <img> as its child:

```
<img src="1" onerror="alert(1)">
<table class="wp-list-table">
  <thead>
  <tr>
    <th></th>
```

```
<th>
  <form method="GET" action=""> ...
```

Note that the injected <img> tag is now outside of the table, simply by virtue of the DOM parsing. The extension will not find an injection in the expected place, creating a false negative (FN). Similarly, elements rearranged inside an injection point can create false positives. This example would generate a class of circumvention techniques for our detector, so we must analyze the DOM before the browser has a chance to modify it.

G. Handling multiple injections in one page

In Listing 1, the endPoints were listed as two strings in the incoming network response. However, there are cases where arbitrarily many injection points can be generated by the application code, such as a for loop generating table rows. For these, it is hard to correctly isolate each endPoint pair, as an attacker could easily inject fake endPoints in between the original ones.

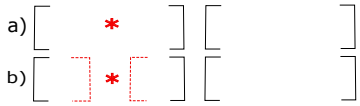

Fig. 3: Example attacker injection when multiple injection points exist in the page. a) a basic injection pattern. b) an attempt to fool the detector.

In Figure 3a, the brackets indicate a template. The content in between (i.e. the star) is an injection point , where dynamic content is injected into the template. In the case of a vulnerability, the injected content can expand to any arbitrary string. The signature separates the injection from the rest by matching for the start and end points (the endPoints), represented by the brackets. This HTML originally has two pairs of endPoint patterns.

In Figure 3b, the attacker knows these are being used as injection end points and decides to inject a fake ending point and a fake starting point (the dotted brackets), with some additional malicious content in between. If just searching for multiple pairs of end points, the detector cannot tell the difference between the solid and dotted patterns, and will not get rid of the content injected in the star. Therefore, we have to use the first starting point and the last ending point before a starting one (when searching from the bottom-up) and sanitize everything in between.

Fig. 4: Example attacker injection when multiple distinct injection points exist in the page.

Figure 4 illustrates a case when there are several injection points in one page, but each of them is distinct. Now, the filter is only looking for one pair of brackets, so the attacker can't fool the extension into leaving part of the injection unsanitized. However, they could, for example, inject an extra ending bracket after the opening parenthesis (or an extra starting brace). The extension will be tricked into sanitizing non-malicious content, the black pluses (+). Since we know the order in which the endPoints should appear, when the filter sees a closing endPoint before the next expected starting endPoint, or similarly, a starting endPoint before the next expected closing endPoint, this attack can be identified. In the diagram, the order of the solid elements characterizes the possible malformations in the end points.

In both scenarios, we have to sanitize the outermost end points. This might get rid of a substantial amount of valid HTML, so we defer to the signature developer's judgment of what behavior the detector should follow. We expand upon this further in Section IV-A.

Note that these complex cases do not mean that our approach is not applicable, as the extension provides a choice for blocking the page entirely if the signature writer believes a given case is too complex for our signature language.

H. Dynamic injections

The top-level documents of web pages fetch additional dynamic content via fetch or AJAX APIs. Content fetched in this way is also vulnerable to XSS, and must be filtered. An example vulnerability is CVE-2018-7747 (in WordPress Caldera Forms, which allows malicious content retrieved from the plugin's database to be injected in response to a click.

XSnare allows XMLHTTPRequests (XHR) to be filtered with XHR-type signatures. To reduce the number of signatures that need to be considered when a browser issues a request, we require that signatures for XHR be nested inside a signature for a top-level document. If a page's main content matches an existing top-level signature description, XSnare will then enable all nested XHR listeners.

Listing 2 shows an example of such a signature. The idea is extensible to scripts and other objects loaded separately from the main document (e.g., images, stylesheets, etc.).

Listing 2: An example dynamic request signature. This patches CVE-2018-7747.

```
...
listenerData: [{
  listenerType: 'xhr', listenerMethod: 'POST',
  sanitizer: 'escape', type: 'string',
  listenerUrl: 'wp-admin/admin-ajax.php',
  typeDet: 'single-unique',
  endPoints: ['<p><strong>', '[AltBody]']
}]
```

III. Implementation

We implemented our system as an extension in Firefox 69.0. Our signatures are stored in a local JavaScript file in the extension package. We decided on an extension implementation for several reasons. (1) *Privileged execution environment*. The extension's logic lies in a separate environment from the web

Algorithm 1: Network filter algorithm

```
1  //global DBSignatures
2  procedure verifyResponse (responseString, url)
3      loadedProbes = runProbes(responseString, url)
4      signaturesToCheck ← []
5      for probe in loadedProbes do
6          signaturesToCheck.append(DBSignatures[probe])
7      end
8      filteredSignatures ← []
9      for signature in signaturesToCheck do
10         if responseString and url match signature then
11             filteredSignatures.push(signature)
12     end
13     versionInfo ← loadVersions(url, loadedProbes)
14     endPoints ← []
15     for signature in filteredSignatures do
16         if (signature,signature.version) ∈ versionInfo
             then
17             endPoints.push(signature.endPointPairs)
18     end
19     indices ← []
20     for endPointPair in endPoints do
21         indices.push(findIndices(responseString,
             endPointPair))
22     end
23     if discrepancies exist in indices then
24         Block page load and return
25     for endPointPair in endPoints do
26         sanitize(responseString,indices)
27     end
28 end
```

application code. This guarantees that malicious code in the application cannot affect the extension. (2) *Web application context*. Our solution requires knowledge of the application's context. The extension naturally retains this context. (3) *Interposition abilities*. As it lies within the browser, the extension can run both at the network level, e.g., rewrite an incoming response; and at the web application level, e.g., interpose on the application's JavaScript execution.

A. Filtering process

Algorithm 1 describes our network filtering process: once a request's response comes in through the network, we process it and sanitize it if necessary.

Loading signatures. Our detector loads signatures and finds injection points in the document. However, not all signatures need to be loaded for a specific website, since not all sites run the same frameworks. When loading signatures, we proceed in a manner similar to a decision tree. The detector first probes the page (line 3) to identify the underlying framework (the **software** in our signature language). We currently provide a number of static probes. However, as more applications are required to be included, we believe it would be better to cover this task in the signature definitions. The widely popular network mapping tool Nmap [29] uses probes in a similar manner, kept in a modifiable file. As mentioned in Section V, we currently only have signatures for Content Management Systems (CMSs) applications. Our probes use specific identifiers related to the application, as well as the particular site that is affected by the exploit. WordPress pages, for example, have several elements in the page that identify it as a WordPress page. While this might seem easier for CMS style pages, and we acknowledge that application fingerprinting is a hard problem in general, we believe other web apps will also have similar identifying information, like headers, element ID's, script/CSS sources, classes, etc. Previous work has shown that DOM element boundaries can be effectively identified given some previous knowledge of the DOM structure [30].

After running these probes, the detector loads corresponding frameworks' signatures and filters out checks whether the information of each loaded signature matches the page (lines 5-12).

Version identification. We then apply version identification (lines 13-16). Our objective for versioning is to prevent signatures from triggering false positives on websites running patched software. We found this to be one of the harder aspects of signature loading. In many CMSs, for example, file names are not updated with the latest version, and versioning information is often unavailable on the client-side.

We have observed that even if we load a signature when the application has already been patched on the server, it can preserve the page's functionality. Motivated by this observation, our mechanism follows a series of increasingly accurate but less precise version identifiers. If versioning is unavailable in the HTML, the patch is applied as we cannot be sure the page is running patched software.

Injection point search and sanitization. Once we have the correct signatures, we find the indices for the endpoints using our top-down, bottom-up scan, and need to check for potential malformations in the injection points (lines 19-24), as described in Section II-G. If this occurs, the page load is blocked and a message is returned to the user, or if the signature developer specifies so, sanitization proceeds on the new endpoints. Finally, if all **endPoint** pairs are in the expected order, we sanitize each injection point (lines 25-27).

B. Sanitization methods

We provide different types of sanitization: "DOMPurify", "escape", and "regex". DOMPurify works well as an out-of-the-box solution. Escaping can be useful when only a few characters need to be filtered. Regex Pattern matching can be particularly effective when the expected value has a simple representation (e.g., a field for only numbers).

IV. Writing Signatures

We expect a signature developer to have a solid understanding of the principles behind XSS, as well as web applications, HTML, CSS and JavaScript, so they can identify precise injection points. In this section, we aim to show that minor effort is required from an analyst when writing a signature.

A. Case Study: CVE-2018-10309

Going back to our example in Section II-C, we describe the process for writing a signature using one of our studied CVEs.

Identifying the exploit. An entry in Exploit Database [31] describes a persistent XSS vulnerability in the WordPress plugin Responsive Cookie Consent for versions 1.7/1.6/1.5. This entry describes the Cookie Bar Border Bottom Size parameter as vulnerable. We run a local WordPress installation with this plugin.

Establishing the separation between dynamic and static content. We insert the string ">script>alert('XSS')</script> in the Cookie Bar Border Bottom Size (rcc_settings[border-size] in the HTML) input field as a proof of concept (PoC). This results in an alert box popping up in the page.

In general, the analyst can find the vulnerable HTML from the server-side code without reproducing the exploit. Since we did not discover the exploit, we had to do this extra step.

In the example, the **input** element is the injection starting point, and the **label** tag is the end point. Identification of correct endpoints is extremely important, and in particular, when a page has multiple injection points, the signature developer must ensure the elements do not overlap with other innocuous ones. In some cases, the developer might think it best to stop the page from loading due to the complexity of the injection points. We believe that if sanitization is impractical, compromising usability for security is preferable.

Collecting other required page information and writing the signature. The next step is to gather the remaining information to determine whether the signature applies to the page loaded. The full signature for this example was previously shown in Listing 1. The page's HTML includes a link to a stylesheet with href "http://localhost:8080/wp-content/plugins/responsive-cookie-consent...", "wp-content/plugins/plugin-name" is the standard way of identifying that a WordPress page is running a certain plugin, in this case, "responsive-cookie-consent". Since the exploit only occurs in this specific spot in the HTML, the **typeDet** is listed as "single-unique". Since the vulnerable parameter is a border-size, the **sanitizer** applied is "regex", further restricting the pattern to only numbers in **config**. We list the **endPoints** as taken from the HTML.

Testing the signature. Finally, we run the extension. We expect to not have an alert box pop up, and we manually look at the HTML to verify correct sanitization. If the exploit is not properly sanitized, the developer is able to use the debugging tools provided by the browser to check the incoming network response information seen by the extension's background page and make sure it matches the signature values.

V. Approach evaluation

To verify the applicability of our detector and signature language, we tested the system by looking at several recent CVEs related to XSS. We have three objectives: to verify that our signature language provides the necessary functionality to express an exploit and its patch, to test our detector against existing exploits, and to show that composing signatures takes a reasonable amount of time.

A. Methodology

We study recent CVEs related to WordPress plugins. We focus on WordPress for two reasons:

1) WordPress powers 34.7% of all websites according to a recent survey [32] [33]. The same study states that 30.3% of the Alexa top 1000 sites use WordPress. Thus, we can be confident that our study results will hold true for the average user.
2) WordPress plugins are popular among developers (there are currently more than 55,000 plugins [34]). Due to its user popularity, WordPress is also heavily analyzed by security experts. A search for WordPress CVEs on the Mitre CVE database [35] gives 2310 results. Plugins, specifically, are an important part of this issue, 52% of the vulnerabilities reported by WPScan are caused by WordPress plugins [36].

We used a CVE database, CVE Details [37] to find the 100 most recent WordPress XSS CVEs, as of October 2018. For each CVE, we set up a Docker container with a clean installation of WordPress 5.2 and installed the vulnerable plugin's version. For CVEs that depended on a particular WordPress version, we installed the appropriate version. Of the CVEs we looked at, only one occurred in WordPress core. We believe it would be harder to precisely sanitize injection points in WordPress core, as many of the plugins have particular settings pages where the exploits occur, and the HTML is more identifiable. WordPress core, on the other hand, can be heavily altered by the use of themes and the user's own changes. However, as evidenced by our investigation, the vast majority of exploits occur in plugins.

Next, we reproduced the exploit in the CVE and we analyzed the vulnerable page and wrote a signature to patch the exploit.

B. Results

Plugin	Installations
WooCommerce	5+ million
Duplicator	1+ million
Loginizer	900,000+
WP Statistics	500,000+
Caldera Forms	200,000+

TABLE I: Top 5 most popular WordPress plugins included in our study.

Of the initial 100 CVEs, we were able to analyze 76 across 44 affected pages. We dropped 24 CVEs due to reproducibility issues: some of the descriptions did not include a PoC, making it difficult for us to reproduce; or, the plugin code was no longer available. In some cases, it had been removed from the WordPress repository due to "security issues", which emphasizes the importance of being able to defend against these attacks.

The plugins we studied averaged 489,927 installations: Table I shows the number of installations for the 5 most popular plugins studied. For the vulnerabilities, 27 (35.5%) could be exploited by an unauthenticated user; 56 (73.7%) targeted a high-privilege user as the victim, 7 (9.2%) had a low-privilege user as the victim; the rest affected all users.

Many of the studied CVEs included attacks for which there are known and widely deployed defenses. For example, many were cases of Reflected XSS, where the URL revealed the existence of an attack, e.g.,: http://⟨target⟩&page-uri=⟨script⟩ alert("XSS")⟨/script⟩ While Chrome's built-in XSS auditor blocked this request, Firefox did not, and so we still wrote signatures for such attacks[2].

We wrote 59 WordPress signatures in total, which got rid of the PoC exploit when sanitized with one of our three methods. Note that while a PoC is often the most simple form of an attack, our sanitization methods can get rid of complex injections as well. We were able to include several CVEs in some PoCs because they occurred in the same page and affected the same plugin. Overall, these signatures represent 71 (93.4%) signed CVEs. The 5 we were not able to sign were due to lack of identifiers in the HTML, which would result in potentially large chunks of the document being replaced[3].

After manual testing, the majority of the 71 signatures maintained the same layout and core functionality of the webpage. However, 12 signatures caused some elements to be rearranged. One caused a table showing user information to render as blank. Most of the responsibility of maintaining functionality is left to the signature developer. We found that being precise is key to retaining functionality. Furthermore, even if the layout of the page is affected, we believe that applying the signature is preferable to allowing an exploit. And, unlike the complete blocking approach commonly used by malware detection software, our approach opens the option to let the user access the page.

While our goal is to retain as much information of the page as possible after sanitization, we believe that even if a part of the page becomes unusable, this does not impact the user's experience as much, since many of the exploits occur in small sections of the HTML. A usability study is out of scope for this paper and we leave it to future work.

C. Generalizability beyond WordPress

To test the generalizability of our approach to other frameworks, we analyzed 5 additional CVEs, 2 related to Joomla!, 2 for LimeSurvey, and 1 for Bolt CMS. We chose Joomla! because it is another popular CMS. Unfortunately, we only found 2 CVEs that we were able to reproduce, as the software for its extensions is often not available. For fairness, we looked for the most recent CVEs we could reproduce listed in the Exploit Database [38], since these have recorded PoCs. We carried out the same procedure as with the WordPress CVEs, and were able to patch all of the 5 exploits. This brought our CVE coverage rate up to 93.8%.

[2]In practice, we found several cases where even XSS auditor did not block a reflected XSS.

[3]In these cases, the signature developer can weigh the trade-offs and decide whether the added cost is worth it.

D. Signature writing times

Figure 5 plots a histogram of the times it took one of the authors to compose each of the signatures. Each time measurement includes the time it took to check the HTML injection points, write the signature and to debug it. We do not include the time taken to discover and carry out an exploit, as we assume a vulnerability has been discovered already. The median time is 3.89 minutes, and the standard deviation is 4.18 minutes. 72% of signatures were written in under 5 minutes. We believe this to be a reasonable amount of time considering the security granted by our extension.

The signature which took the longest time to write (25 minutes) corresponds to an exploit with 12 HTML injection points. Additionally, testing this signature proved difficult, as some of the injections were a result of a script inserting elements in the DOM after the page had loaded. This caused the initial HTML to look innocuous, but with exploits still occurring after sanitization. As this script was part of the initial request, we eventually got to the root of the problem. We believe a more experienced exploit analyst might be able to detect this kind of behaviour more easily.

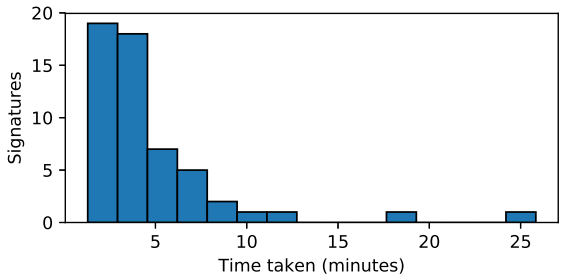

Fig. 5: Histogram of time taken to write signatures.

VI. Load time performance on top websites

XSnare's performance goal is to provide its security guarantees without impacting the user's browsing experience. We now briefly report XSnare's impact on top website load times, representing the expected behaviour of a user's average web browsing experience.

In our setup, we used a headless version of Firefox 69.0, and Selenium WebDriver for NodeJS, with GeckoDriver. All experiments were run on one machine with an Intel Xeon CPU E5-2407 2.40GHz processor, 32 GB DRAM, and our university's 1GiB connection.

In our tests we used the top 500 websites as reported by Moz.com [39]. For each website, we loaded it 25 times (with a 25 second timeout) and recorded the following values: requestStart, responseEnd, domComplete, and decodedBodySize. From the initial set of 500, we only report values for 441: the other 59 had consistent issues with timeouts, insecure

certificates, and network errors. We believe these to have been caused by the Selenium web driver, as our extension runs after a response has been delivered to the browser. We manually loaded each page on a personal computer with our extension running successfully and were not able to reproduce the issues.

We ran four test suites: **No extension cold cache**: Firefox is loaded without the extension installed and the web driver is re-instantiated for every page load. **Extension cold cache**: As before, but Firefox is loaded with the extension installed. **No extension warm cache**: Firefox is loaded without the extension installed and the same web driver is used for the page's 25 loads. **Extension warm cache**: As before, but Firefox is launched with the extension installed.

For each set of tests, we reduced the recorded values to two comparisons: network filter (responseEnd - requestStart), and page ready (domComplete - responseStart). The first analyzes the time spent by the network filter, while the second determines the time spent until the whole document has loaded. We calculate the medians for each website for each of these measures as well as the decodedByteSize.

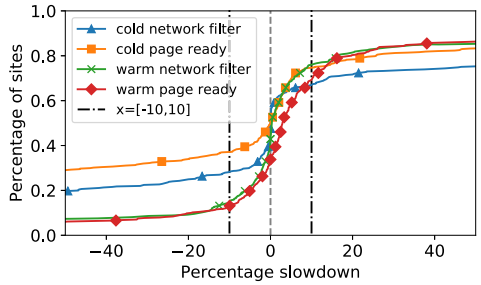

Fig. 6: Cumulative distribution of relative percentage slowdown with extension installed for top sites.

We compare the load times with/without the extension by calculating the relative slowdown with the extension installed according to the following formula:

$$100 * \frac{\tilde{x}_{with} - \tilde{x}_{without}}{\tilde{x}_{without}}$$

where $\tilde{x}$ is the median with/without the extension running.

Figure 6 plots the results. We can see a slowdown of less than 10% for 72.6% of sites, and less than 50% for 82% of sites when the extension is running. Note that these values are recorded as percentages, and while some are as high as 50%, the absolute values are in 77% of cases less than a second. This overhead should not alter the user's experience significantly.

The slowdown increases by at most 5% when we take caching into account. This is likely because the network filter causes the browser to use less caching, especially for the DOM component, as it might have to process it from scratch every time. While it may seem counter-intuitive that some pages have shorter loading times with the extension, there are several variables at play that can affect these measurements (local network, server-side load, internal scheduling, etc). We manually checked the websites for which values were higher than $|40\%|$ and verified that our extension did not change the page's contents, a possible cause of faster load times. We also checked the timings for the page as reported by the browser and noted a high variance even within small time windows. The time spent by our verification function was less than 10ms for 87.6% of sites (Figure 7). This corroborates our findings that the slowdown is mostly negligible.

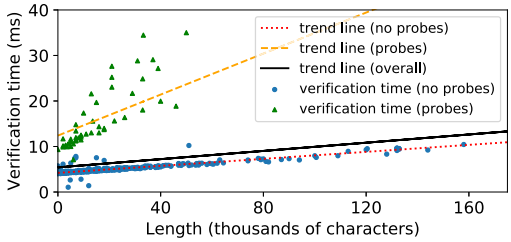

Fig. 7: Scatter plot of network filter time as a function of character length for top sites.

Figure 7 shows the time spent by the call to our string verification function in the network filter as a function of the length of the string to be verified, differentiating between websites for which some probes tested positive and ones which no probes did. We applied least squares regression to calculate the shown trend lines. Since both our probes and signatures use regex matching, we expect both trend lines to be linear, as seen in the graph. We expect the slope of the line to be higher when a probe passes, as it performs additional string verification. Around 37.4% of all web sites use frameworks covered by our probes [32], thus, we expect the impact of our network filter to be closer to the non-probe values, as corroborated by our overall trend line.

False positives on the Web. For each website, we recorded the number of loaded signatures. We report a 0% FP rate for these. Thus, we can infer with confidence that the rate of falsely loaded signatures during an average user's web browsing is similarly low. This rate could possibly go up as we cover more frameworks. Since many of these pages are not running WordPress and are very popular and more prone to fixing their vulnerabilities, the rate of false negatives is likely extremely low as well.

Scalability with signatures. We tested our system with a large number of signatures. We added 15,500 signatures to our database and recorded the time spent by the network filter to process these sites[4]. These were crafted so that the extension would check each one against the loaded sites, without triggering the injection search and sanitization. Thus, we effectively forced our extension to test each site against 15,500 signatures. The mean time spent by the filtering process was 1,930ms, with less than 2,000ms for 88% of the sites. In practice, we expect a smaller filter time, as many frameworks

[4]There are 15,303 CVEs related to XSS in CVE Details [40].

would have many signatures. For example, there are currently 200+ CVEs listed for WordPress core and its plugins.

VII. Limitations and Future Work

Generalizability and scope of study. As discussed in Section V-A, while many websites share similar structures to the ones we covered, our study only considered 4 other sites not running WordPress, and our signatures only cover CMS content. Not all websites might be identified as easily. Furthermore, we only studied 81 CVEs. In the future we intend to study a more diverse set of CVEs and websites.

False positives and false negatives. It is extremely hard to get completely rid of FPs. If the sanitization targets JavaScript code, for example, a FP will likely be triggered. Furthermore, since we rely on handwritten signatures, vulnerable sites for which no signature has been written will be subject to FNs. In the future, we intend to study the rate of FPs and FNs in our approach and compare it to previous work.

Protection against CSRF. We believe that we can adapt our work to defend against Cross-Site Request Forgery (CSRF) exploits, as well. Using a similar signature language as the one for XSS, a signature developer could specify pages with potential vulnerabilities to only allow network requests that cannot exploit such vulnerabilities.

Filtering network data. Our filter's design depends on Firefox's implementation of the WebRequest API. Firefox's filterResponseData method allows the extension to modify an incoming HTTP request[5]. This feature has been requested in other browsers like Chrome, but it has not been implemented. This design limits our deployability to Firefox users.

Design considerations. Currently, each browser user has to install our extension. However, the same functionality could be offloaded to a single processing unit similar to a proxy, which can handle the filtering for all machines in a network. This deployment model might be more appropriate in certain environments, such as in an enterprise setting.

VIII. Related Work

We classify existing work into several categories: client-side, server-side, browser built-in, and hybrid approaches.

Server-side techniques. In addition to existing parameter sanitization techniques, taint-tracking has been proposed as a means to consolidate sanitization of vulnerable parameters, and identify vulnerabilities automatically. [2], [16], [17], [18], [19], [41]. These techniques are complementary to ours, and provide an additional line of defence against XSS.

There has also been work on other server-side analysis approaches to find bugs security vulnerabilities in web applications. [42], [43], [44]. However, these do not target XSS specifically.

Client-side techniques. DOMPurify [7] presents a robust XSS client-side filter. The authors argue that the DOM is the ideal place for sanitization to occur. While we agree with this view, this work relies on application developers to adopt the filter and modify their code to use it. We have partly automated this step by including it as our default sanitization function.

Jim et al. [3] present a method to defend against injection attacks through Browser-Enforced Embedded Policies. This approach is similar to ours, as the policies specify prohibited script execution points. Similarly, Hallaraker and Vigna [23] use a policy language to detect malicious code on the client-side. Like XSnare, they make use of signatures to protect against known types of exploits. However, unlike our approach, their signatures are not application-specific, and there is no model for signature maintenance.

Snyder et al. [10] report a study in which they disable several JavaScript APIs and test the number of websites that are do not work without the full functionality of the APIs. This approach increases security due to vulnerabilities present in several JavaScript APIs, however, we believe disabling API functionality should only be used as a last resort.

Additionally, client-side taint tracking, through the use of static and dynamic analysis, has also been applied as a means to detect XSS, either at the browser level or at the extension level [45], [46].

Browser built-in defences. Browsers are equipped with several built-in defences. We previously described XSS Auditor in Section I. Another important one is the Content Security Policy (CSP) [47]. It has been widely adopted and in many cases provides developers with a reliable way to protect against XSS and CSRF attacks. However, CSP requires the developer to identify which scripts might be malicious. Previous work has also highlighted the need for further built-in defences [48].

Client and server hybrids. XSS-Dec [6] uses a proxy which keeps track of an encrypted version of the server's source files, and applies this information to derive exploits in a page visited by the user. This approach is similar to ours, since we assume previous knowledge of the clean HTML document. Furthermore, they use anomaly-based and signature-based detection to prevent attacks. Our system offloads all this functionality to the client-side, without the need for any server-side information.

IX. Conclusion

Users cannot depend on administrators to patch vulnerable server-side software or for developers to adopt best practices to mitigate XSS vulnerabilities. Instead, users should protect themselves with a client-side solution. In this paper we described the design, implementation, and evaluation of XSnare, one such client-side approach. XSnare prevents XSS exploits by using a database of exploit signatures and by using a novel mechanism to detect XSS exploits in a browser extension. We evaluated XSnare through a study of 81 CVEs in which we showed that it defends against 93.8% of the exploits.

Acknowledgment

We would like to thank Dr. William Aiello, who provided crucial expert advice and insight in the early stages of the project. We miss him dearly.

[5]https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API/webRequest/filterResponseData

REFERENCES

[1] I. Muscat. (2017, jun) Acunetix vulnerability testing report 2017. https://www.acunetix.com/blog/articles/acunetix-vulnerability-testing-report-2017/.

[2] G. Wassermann and Z. Su, "Static detection of cross-site scripting vulnerabilities," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 171–180. [Online]. Available: https://doi.org/10.1145/1368088.1368112

[3] T. Jim, N. Swamy, and M. Hicks, "Defeating script injection attacks with browser-enforced embedded policies," in *Proceedings of the 16th International Conference on World Wide Web*, ser. WWW '07. New York, NY, USA: ACM, 2007, pp. 601–610. [Online]. Available: http://doi.acm.org/10.1145/1242572.1242654

[4] Y. Nadji, P. Saxena, and D. Song, "Document structure integrity: A robust basis for cross-site scripting defense." in *NDSS*, vol. 20, 2009.

[5] P. Wurzinger, C. Platzer, C. Ludl, E. Kirda, and C. Kruegel, "Swap: Mitigating XSS attacks using a reverse proxy," in *Proceedings of the 2009 ICSE Workshop on Software Engineering for Secure Systems*, ser. IWSESS '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 33–39. [Online]. Available: http://dx.doi.org/10.1109/IWSESS.2009.5068456

[6] S. Sundareswaran and A. C. Squicciarini, "XSS-Dec: A hybrid solution to mitigate cross-site scripting attacks," in *Proceedings of the 26th Annual IFIP WG 11.3 Conference on Data and Applications Security and Privacy*, ser. DBSec'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 223–238. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-31540-4_17

[7] M. Heiderich, C. Späth, and J. Schwenk, "Dompurify: Client-side protection against XSS and markup injection," in *Computer Security – ESORICS 2017*, S. N. Foley, D. Gollmann, and E. Snekkenes, Eds. Cham: Springer International Publishing, 2017, pp. 116–134.

[8] Noscript homepage. https://noscript.net/.

[9] B. Stock, M. Johns, M. Steffens, and M. Backes, "How the web tangled itself: Uncovering the history of client-side web (in)security," in *Proceedings of the 26th USENIX Conference on Security Symposium*, ser. SEC'17. Berkeley, CA, USA: USENIX Association, 2017, pp. 971–987. [Online]. Available: http://dl.acm.org/citation.cfm?id=3241189.3241265

[10] P. Snyder, C. Taylor, and C. Kanich, "Most websites don't need to vibrate: A cost-benefit approach to improving browser security," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: ACM, 2017, pp. 179–194. [Online]. Available: http://doi.acm.org/10.1145/3133956.3133966

[11] (2016) Hacked website report 2016/q3. https://sucuri.net/reports/Sucuri-Hacked-Website-Report-2016Q3.pdf.

[12] (2019) Statistics show why wordpress is a popular hacker target. https://www.wpwhitesecurity.com/statistics-70-percent-wordpress-installations-vulnerable/.

[13] (2019) XSS auditor. https://www.chromium.org/developers/design-documents/xss-auditor.

[14] (2019) Intent to deprecate and remove: XSSAuditor. https://groups.google.com/a/chromium.org/forum/#!msg/blink-dev/TuYw-EZhO9g/blGViehIAwAJ.

[15] B. Stock, S. Lekies, T. Mueller, P. Spiegel, and M. Johns, "Precise client-side protection against dom-based cross-site scripting," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, ser. SEC'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 655–670. [Online]. Available: http://dl.acm.org/citation.cfm?id=2671225.2671267

[16] W. Xu, S. Bhatkar, and R. Sekar, "Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks," in *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, ser. USENIX-SS'06. Berkeley, CA, USA: USENIX Association, 2006. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267336.1267345

[17] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans, "Automatically hardening web applications using precise tainting," in *Security and Privacy in the Age of Ubiquitous Computing, IFIP TC11 20th International Conference on Information Security (SEC 2005), May 30 - June 1, 2005, Chiba, Japan*, 2005, pp. 295–308.

[18] T. Pietraszek and C. V. Berghe, "Defending against injection attacks through context-sensitive string evaluation," in *Proceedings of the 8th International Conference on Recent Advances in Intrusion Detection*, ser. RAID'05. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 124–145. [Online]. Available: http://dx.doi.org/10.1007/11663812_7

[19] P. Bisht and V. N. Venkatakrishnan, "XSS-GUARD: Precise dynamic prevention of cross-site scripting attacks," in *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, ser. DIMVA '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 23–43. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-70542-0_2

[20] (2018) Security report for in-production web applications. https://www.rapid7.com/resources/security-report-for-in-production-web-applications/.

[21] M. Steffens, C. Rossow, M. Johns, and B. Stock, "Don't trust the locals: Investigating the prevalence of persistent client-side cross-site scripting in the wild," in *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*, 2019.

[22] E. Kirda, N. Jovanovic, C. Kruegel, and G. Vigna, "Client-side cross-site scripting protection," *Comput. Secur.*, vol. 28, no. 7, pp. 592–604, Oct. 2009. [Online]. Available: http://dx.doi.org/10.1016/j.cose.2009.04.008

[23] O. Hallaraker and G. Vigna, "Detecting malicious javascript code in mozilla," in *Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems*, ser. ICECCS '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 85–94. [Online]. Available: http://dx.doi.org/10.1109/ICECCS.2005.35

[24] (2018) Wordpress plugin responsive cookie consent 1.7 / 1.6 / 1.5 - (authenticated) persistent cross-site scripting. https://www.exploit-db.com/exploits/44563.

[25] (2019) Responsive cookie consent 1.8 patches. https://plugins.trac.wordpress.org/browser/responsive-cookie-consent/tags/1.8/includes/admin-page.php.

[26] (2018, aug) How does adblock work? https://help.getadblock.com/support/solutions/articles/6000087914-how-does-adblock-work-.

[27] J. C. Pazos, J.-S. Legare, I. Beschastnikh, and W. Aiello, "Precise XSS detection and mitigation with client-side templates," 2020. [Online]. Available: https://arxiv.org/abs/2005.07826

[28] (2019) Safely inserting external content into a page. https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Safely_inserting_external_content_into_a_page.

[29] (2019) nmap network mapper. https://nmap.org/.

[30] C.-P. Bezemer, A. Mesbah, and A. van Deursen, "Automated security testing of web widget interactions," in *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC/FSE '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 81–90. [Online]. Available: https://doi.org/10.1145/1595696.1595711

[31] (2019) Wordpress plugin responsive cookie consent 1.7 / 1.6 / 1.5 - (authenticated) persistent cross-site scripting. https://www.exploit-db.com/exploits/44563.

[32] (2019) Usage of content management systems for websites. https://w3techs.com/technologies/overview/content_management/all.

[33] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *CoRR*, vol. abs/1801.01203, 2018. [Online]. Available: http://arxiv.org/abs/1801.01203

[34] (2019) Wordpress: Plugins. https://wordpress.org/plugins/.

[35] (2019) Wordpress cves. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=wordpress.

[36] (2019) Wpscan. https://wpscan.org/.

[37] (2019) Wordpress: Vulnerability statistics. https://www.cvedetails.com/product/4096/Wordpress-Wordpress.html?vendor_id=2337.

[38] (2019) Exploit database. https://www.exploit-db.com/.

[39] Moz top 500 websites. https://moz.com/top500.

[40] (2020) Cve details vulnerabilities by type. https://www.cvedetails.com/vulnerabilities-by-types.php.

[41] A. Kieyzun, P. J. Guo, K. Jayaraman, and M. D. Ernst, "Automatic creation of sql injection and cross-site scripting attacks," in *2009 IEEE 31st International Conference on Software Engineering*, 2009, pp. 199–209.

[42] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su, "Dynamic test input generation for web applications," in *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ser. ISSTA '08. New York, NY, USA: Association

for Computing Machinery, 2008, p. 249–260. [Online]. Available: https://doi.org/10.1145/1390630.1390661

[43] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst, "Finding bugs in web applications using dynamic test generation and explicit-state model checking," *IEEE Transactions on Software Engineering*, vol. 36, no. 4, pp. 474–494, 2010.

[44] X. Xiao, A. Paradkar, S. Thummalapenta, and T. Xie, "Automated extraction of security policies from natural-language software documents," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: Association for Computing Machinery, 2012. [Online]. Available: https://doi.org/10.1145/2393596.2393608

[45] J. Pan and X. Mao, "Detecting dom-sourced cross-site scripting in browser extensions," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2017, pp. 24–34.

[46] F. Sun, L. Xu, and Z. Su, "Client-side detection of XSS worms by monitoring payload propagation," in *Computer Security – ESORICS 2009*, M. Backes and P. Ning, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 539–554.

[47] (2019) Same-origin policy. https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy.

[48] E. Abgrall, Y. L. Traon, S. Gombault, and M. Monperrus, "Empirical investigation of the web browser attack surface under cross-site scripting: An urgent need for systematic security regression testing," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*, 2014, pp. 34–41.