XSnare: Client-side Cross-site Scripting Protection

José Carlos Pazos
University of British Columbia
jpazos@cs.ubc.ca

Jean-Sébastien Légaré
University of British Columbia
jslegare@cs.ubc.ca

William Aiello
University of British Columbia
aiello@cs.ubc.ca

ABSTRACT

We present XSnare, a fully client-side cross-site scripting (XSS) solution, implemented as a Firefox extension. Our approach takes advantage of available previous knowledge of a webpage's HTML, as well as the rich context available in the DOM to interpose on these attacks. It uses a database of exploit descriptions, which are written with the help of previously recorded CVEs, to prevent them. CVEs for XSS are widely available and are one of the main ways to tackle zero-day exploits. We effectively single out injection points in the HTML, preventing malicious code from executing in these vulnerable points. XSnare allows users to protect themselves without having to wait for developers to patch their code once a vulnerability has been released.

We evaluate the applicability of our approach by studying the latest 100 CVEs related to XSS attacks in WordPress, and find that our tool defends against 93.4% of these exploits. To the best of our knowledge, our work is the first to systematically study an XSS protection tool on real-world exploits in the form of CVEs. Our peformance evaluation shows that our extension's overhead on web page loading time is less than 10% for 72.6% of sites.

1 INTRODUCTION

Cross-site scripting (XSS) is still one of the most dominant web vulnerabilites. In 2017, a report showed that 50% of websites contained at least one XSS vulnerability [30]. Even though many countermeasures have been researched to combat these issues, many of them lack widespread deployment, and so have been unable to protect users. Many of these defenses leverage server-side techniques, along with browser modifications [27, 31, 38]; or require additional developer effort [26]. Still, others disable client-side functionality [2, 34], sometimes rendering websites unusable. We believe many of these solutions have not seen widespread adoption because they simply are not practical: developers might not be willing, or might not have the resources or expertise available to implement them. Furthermore, even when enough information is available for a developer, many website administrators won't deploy fixes immediately: a 2016 study found that 61% of WordPress websites were running a version with known security vulnerabilities [4], and another found that 30.95% of Alexa's top 1 Million sites run a vulnerable version of WordPress [16]. As the number of websites using client-side technologies continues to increase (a study showed that as of 2012, almost 100% of the Alex top 500 sites were using JavaScript [36]), users are effectively left at the mercy of developers, without tools that both allow them to protect themselves and browse the web worry-free.

Our work focuses on WordPress as a study platform: we look at recent CVEs related to WordPress plugins. While this may seem restrictive, there are several reasons why this is a worthwhile endeavour:

- WordPress powers 25% of all websites according to a recent survey [20]. 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.
- Due to its user popularity, it is also heavily analysed by security experts. There are currently 286 CVEs related to WordPress in the CVE Details database [21]. Plugins, specifically, are an important part of this issue, 52% of the vulnerabilities reported by WPScan are caused by WordPress plugins [22].
- Due to the open source nature of WordPress plugins, we can easily analyze both the client-side HTML, as well as the server-side code that generated it, and use this to reach conclusions about the design of our solution.

Even though our study has not focused on other sites, our approach is not limited to a specific framework, and we believe it should generalize to arbitrary webpages, under the assumption that we have a pre-existing notion of a webpage's contents.

To provide users with the means to protect themselves in the absence of control over servers, we strongly believe a client-side solution must be delivered. A number of existing solutions also suffer from a high rate of false-positives and false-negatives, due to the lack of information available at the layers they operate at. For example, the popular browser extension, NoScript [2], works via domain whitelisting, thus by default, JavaScript scripts and other code will not execute. However, not all scripts outside of the whitelist should be assumed to be malicious. Browser-level filters like Chrome's XSS auditor [23] work based on general policies and can therefore incorrectly sanitize non-malicious scripts.

In contrast, we posit that the DOM is the right place to interpose for the purpose of mitigating against these attacks, since we have the full picture at that point. While most of the functionality we provide could be done by a network filter on top of the browser, we require some additional context to perform effective XSS identification. In particular, when an exploit manifests itself through dynamic behaviour, like a network request initiated by an user's click, we require knowledge of the tab which initiated the request to filter the response.

Our system consists of three main components: a trusted Firefox extension for interposing between the application and the DOM, a periodically updated local database which maintains exploit definitions and descriptions of the steps needed to be followed by the extension, and finally, a declarative language for defining exploits, expressive enough for an user to be precise about which parts of the HTML are vulnerable.

2 XSNARE

We now present XSnare's components and how they interact with each other. First, we give a overview of current defence solutions and how they fit into our model. Further details of these different approaches are described in Section 5.

2.1 Web application architecture

Figure 1 shows a typical architecture for web applications. There are several different places where vulnerability defences can be integrated. We give a brief description of what can be done at each point:

(1) At the application's server-side, the developer is trying to defend itself against malicious users. The first line of defence from these vulnerabilities lies in the application logic itself. The developer might choose to ensure safety of the code, either by using third-party solutions, or by securing the code themselves, for example, by applying static analysis on the server code to detect unsanitised input.
(2) Inside the hosting environment, developers deploy defences including generic firewalls and more specific Web Application firewalls (WAFs), which defend against attacks such as DDoS, SQL injections and XSS.
(3) At the client side layer the user is defending from malicious websites, and may have their own generic firewall, blacklisted websites, and proxies.
(4) Finally, the information gets to the user's browser. This will usually have built-in defences, such as Chrome's XSS auditor. The user might also install browser-dependent functionality, such as extensions like NoScript.

Many of these approaches are either unfit for widespread deployment or do not benefit from an application's contextual knowledge. For example, a WAF might be enough to defend against most XSS attacks on one deployment, but it would require each individual developer to have the necessary knowledge and resources to integrate one. A network proxy at the client-side will usually have a generic set of rules to apply on incoming network traffic, and this will often lead to an elevated rate of false positives. Browser built-in defences are very coarse, and will only work on a subset of exploits. Chrome's XSS auditor, for example, only attempts to defend against reflected XSS. In fact, Google has recently announced its intention to deprecate XSS auditor, with reasons including "Bypasses abound", "It prevents some legit sites from working", and "Once detected, there's nothing good to do" [9].

We instead focus on application-specific detection at the client-side layer, and thus, don't rely on any server-side infrastructure (or its operators). Furthermore, it is complementary to the aforementioned techniques: a WAF will not reduce the security of our approach by any means, and having these two work in tandem is beneficial to the user's experience.

2.2 Operation, at a glance

Commonly, bug bounty hunters and penetration testers will scour websites to find vulnerabilities and alert developers of issues in these, as well as potential fixes. Developers will then fix the bugs accordingly so that users are not subject to vulnerabilities. Inspired by this workflow, we believe this process can be partly automated using a firewall-based approach, so that users don't have to wait for developers to update their code. Figure **??** illustrates how the firewall can be used to guarantee full client-side protection: A user loads a request, such as www.myblog.com, this request might come back with malicious code in the form of an XSS attack. Before rendering the webpage in the browser, an extension can analyze

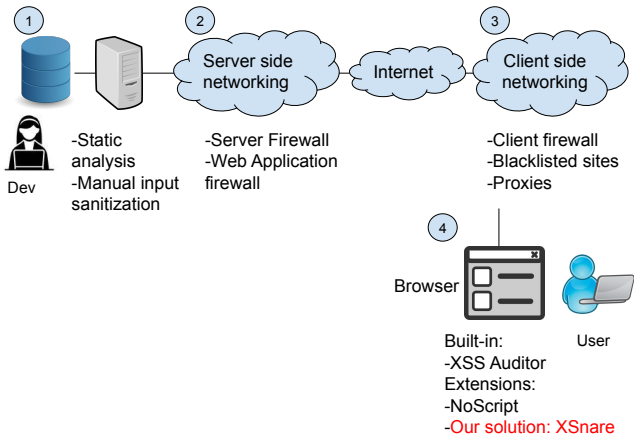


Figure 1: Architecture of typical web applications. Different security solutions apply at distinct layers.

the potentially malicious document, doing so by loading signatures which a security analyst (a bug bounty hunter, for example) has uploaded to a database, and completely eliminating the injected code. Finally, the extension returns a clean HTML document, which the browser then renders.

2.3 An example application of XSnare

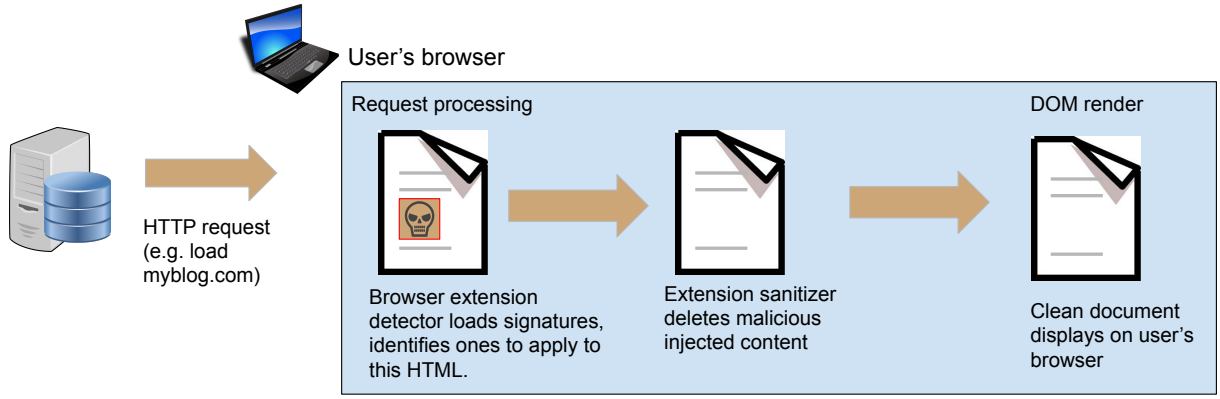
In order to further illustrate this approach, we present a small example of how DOM context can be used to defend against XSS, taken from CVE 2018-10309 [7]. This is reproducible in an off-the-box WordPress installation running the Responsive Cookie Consent plugin, v1.7. This is also an example exploit which Chrome's XSS auditor does not protect against. Consider a website running PHP on the backend that takes user input and stores it to later display it to another user; in this case, the input element's value attribute is set by the webpage admin using the plugin's UI.

The PHP code defines the static HTML template (in black) in the "admin-page.php" file, as well as the dynamic input (in red):

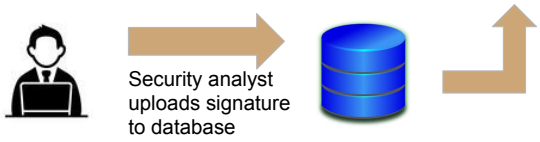
```
<input id="rcc_settings[border-size]"
name="rcc-settings[border-size]"
type="text" value="<?php rcc_value('border-size'); ?>"/>
<label class="description"
for="rcc_settings[border-size]">
```

This HTML will be displayed in the admin's UI. Under normal circumstances, the input element might be used to insert a value of "0" as the dynamic content:

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



(a) XSnare's browser extension component



(b) XSnare's signature component

Figure 2: XSnare's approach for protection against XSS.

However, the php code is vulnerable to an injection attack, as it is not sanitized on the server side. If we have

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

then, the browser will render the following, 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 this HTML is well-formed, so it is hard to detect that a malicious injection has occurred without knowing the developer's intention. However, assuming an analyst has knowledge of how the full HTML should render without any injections, and the possible range of values of the injection, they can single out the point of injection, by separating user input from the server-side template, and get rid of the malicious script entirely. In the example, the injected script in red can be easily distinguished from the rest of the HTML template due to their identifiable attributes. By searching for this specific input element from the top of the document, and this label element from the bottom, the dynamic content can be identified from the static template.

We aim to reproduce the developer's intended server-side patch on the client-side, therefore, we need to be able to determine the separation between dynamic content and the static template, and pass the dynamic elements to a sanitization function. We provide various sanitization methods to reach this goal.

Ideally, we could apply sanitization functions that are equivalent to the server-side patch, but the developer's intention might not always be clear. After careful analysis of the code in the example, we noted that while the CVE describing this vulnerability states that the bug was fixed in version 1.8 of this plugin, this was not the case: the developer fixed other similar vulnerabilities but did not handle this specific parameter. However, we can infer the application's intended behaviour from the other patches [12]. In particular, the developer applied a built-in WordPress function "sanitize_text_field", which sanitizes the parameters by checking for common invalid characters like invalid UTF-8. The task of determining the intended behaviour falls upon a security analyst, who will act as the signature developer for a given exploit.

In the following sections we give a detailed description of each component of our system, the challenges that arise when trying to defend against XSS client-side, and the tools provided by the browser to facilitate our methods.

2.4 Firewall Signatures

The firewall signatures are at the core of our defense strategy. These must be precise enough for our system to get rid of the intended injection, and not an element of the website crucial to the user experience. Since we are only relying on DOM knowledge, these signatures must be related to HTML features, for example, specifying elements and element attributes that are unique to where the exploit might occur.

The basis for our signatures relies on two observations: first, an injection has a start 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 any JavaScript execution (e.g. removing and adding elements), allowing us to isolate it from the template code. Thus, our basic approach at signature definition is to specify an injection's start and its end, and any sanitization to be done between these two endpoints. Typically, one page will have several dynamic content injection points. The signature developer has to be able to identify all of these. Different scenarios might warrant different resolutions, ranging from stopping a webpage's rendering altogether, to performing some basic checks on the string. We discuss how different exploits might affect a signature definition and how our signature language gives an analyst enough expressibility to deal with these in later sections.

We believe CVEs to be an ideal source for signatures, and our system assumes these are written by a third-party: as discussed previously, bug bounty hunters and penetration testers will commonly identify issues in application code, inform developers and publish it for the benefit of the community in the form of CVEs. Our system adds an extra component to this workflow, where hackers and security enthusiasts also write the signatures to defend users. Thus, the signature database is maintained by a trusted entity which audits CVEs, and thus, a malicious analyst can not take advantage of this model to throttle the extension's performance. An analyst can write a signature in our language given their knowledge on the exploit, as they will often know both the source and the way it manifests in the HTML, as well as the fix.

2.5 Firewall Signature Language

Our signature language needs to be such that it has 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. Due to the nature of our signature definitions, a regex language suffices to express precise sections of the HTML. Furthermore, a regex language allows us to identify malformed HTML before it renders on the browser. The following is the signature that defends against the motivating example of Section 2.1:

```
url: 'wp-admin/options-general.php?page=rcc-settings',
software: 'WordPress',
softwareDetails: 'responsive-cookie-consent',
version: '1.5',
type: 'string',
typeDet: 'single',
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]">']
```

Along with the previously mentioned endpoints, the signature also defines the urls, if necessary, in which the exploit occurs, what kind of webpage it is (WordPress in this case), and any additional details of the software. In this case, since it's a WordPress site, the details include the description of the vulnerable plugin. The `typeDet` value specifies whether this signature includes a "single" pair of endPoints, or "multiple". This is useful for when one signature developer has identified several vulnerable points in the same document, and the writing can be streamlined into one signature. Having multiple injection points in one document complicates the detection mechanism: for example, if one document has two different injection points, a knowledgeable attacker might leverage this information to try to trick the extension into identifying the wrong spots. We further discuss measures to protect against these attacks in the implementation section.

Once the dynamic content is identified, the analyst can configure their signatures with a function chosen from a pre-defined static set of sanitization functions. These functions innoculate potential malicious injections based on the DOM context surrounding the injection. The goal of signatures is to provide such sanitization, while maintaining the core web page user experience. To this end, default injection point sanitization is done with DOMPurify [26]. This library is described by its creators as a "DOM-only, super-fast, uber-tolerant XSS sanitizer for HTML, MathML and SVG". The Mozilla community cites it as an useful tool for "safely inserting external content into a page" [13]. While it offers a lot of configurability and hooks, we have used the default functionality, with satisfying results, as described in Section 4, in our own signatures. However, there are cases where page functionality is lost due to a naive sanitization approach. Thus, it is sometimes more desirable to use a different sanitization approach, especially when heavier methods disrupt the look and feel of the web page.

We provide different types of sanitization: "DOMPurify", "escape", and "regex". Regex Pattern matching can be particularly effective when the expected value has a simple representation, e.g. a name field should only have a particular subset of characters available. Additionally, for each of these approaches, the signature can specify a corresponding config value, as described in Section 5.1. DOMPurify provides a rich API for additional configuration. When escaping, defining specific characters to escape via regex can be useful. For pattern matching via regex, the config value specifies the string the injection point content should match.

An alternative to this approach would have been to allow signatures to specify arbitrary code for the sanitization routines. While this would provide a more accurate sanitization technique, we have decided to impose a declarative spec for these routines for three main reasons:

(1) Security Concerns: We assume signatures come from a trusted source. However, partly due to the way they are currently stored, it is possible for an attacker to add malicious signatures. In general, this would only cause pages to trigger signatures that should not have been triggered, potentially harming the web site user experience. If we allowed arbitrary code to run from signatures, an attacker could take control of the victim's browser, as the code would execute in a high-privilege environment.
(2) Case Coverage: While our provided methods might be limited in some scenarios, we have applied them in our studied CVEs with positive results, and are confident they can cover most use cases.

(3) Adoption: A declarative language will help signature developers expedite the process of writing signatures, as they will find that our provided methods will most often suffice.

2.6 Firefox Extension

Our system's main component is a browser extension which rewrites potentially infected HTML into a clean document. We have decided to implement this tool as an extension due to the following main benefits:

- **Ease of deployment and adoption**. Most modern browsers provide a convenient way to upload extensions to their respective official sites (Firefox Add-Ons in the case of Firefox). This eases deployment for the extension developer, and also eases installation for an user, as they can install the tool within the browser in one click, without having to install additional third-party software.
- **Privileged execution environment**. The extension's logic can lie in a separate environment from the web application code. This is important for security, as it guarantees that any malicious code in the application can not affect the extension's behaviour. If the extension needs to run code within the application, it can maintain security by installing the code before the application's code runs. A web application, on the other hand, can not achieve this level of security in general, as it can not guarantee that its own code will run before any malicious code.
- **Web application context**. Our solution requires knowledge of the application's context to perform several of its tasks. For example, it's useful to know which tab generated a network request to determine whether the response content is safe or not. The extension naturally retains this context.
- **Ability to interpose at the network level and the web application level**. As it lies within the browser, the extension can run both at the network level, e.g. rewrite an incoming response or outgoing request; and at the web application level, e.g. keeping track of click events, or interpose on the application's JavaScript execution. Other types of solutions often have to choose between one or the other.

Our extension's main purpose is to detect exploits in the HTML by using signature definitions and maintain a local database of signatures that is periodically updated from the main server. The extension model provided by several browsers allows us to interpose on any functionality of a website in a privileged execution environment, unavailable to any third-party. In particular, Firefox provides the filterResponseData method through the webRequest API [17] . This allows the extension's background page to analyse and modify incoming network traffic. The extension translates signature definitions into the logic needed to rewrite incoming HTML on a per-URL basis, according to the top-down, bottom-up scan described earlier.

The patch applied by the extension needs to take place in the raw HTML string. Even before any code runs, the parsing of the HTML into a DOM tree might cause elements to be re-arranged into an unexpected order, making our extension sanitize the wrong spot. For example, a <tr> element may only have direct children <th> or <td>. In our experiments, we found that an injection occurring as a direct child of the <tr> might cause the injected element to be rendered before the <tr> in the DOM. The following code shows the HTML as written in its file, with the injection (the img element) in red:

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

According to this HTML, the signature developer might identify the exploit as occurring inside the table with the specified class, and the extension would be able to sanitize the code correctly. However, if we wait until the string has been parsed into a DOM tree, the elements are rearranged due to the previously mentioned rule:

```
<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 element is now outside of the table, simply by virtue of the DOM parsing. When performing the top-down search, the extension will search past the injection, as it occurs before the table element, and will therefore miss it during sanitization, creating a false negative. Similarly, false positives might occur if elements which were outside of an injection point are rearranged inside one. Thus, we can't wait until the website is rendered client-side to start interposing on code execution.

A knowledgeable attacker can therefore not take advantage of such behaviour: even if they know what the signatures look like, the extension can't be tricked into looking for the element in the wrong spot, as the injection can only happen after a signature's start and before its end. Additionally, the injection must end at a certain point in the document. Since we look for the end point from the bottom-up, we will eventually reach the injection's end, and, regardless of the contents (for example, trying to spoof the end point early), the sanitization will be correctly applied.

3 IMPLEMENTATION

We have implemented our browser extension in Firefox 69.0. Our signatures are currently stored in a local JavaScript file in the extension package.

3.1 Loading signatures

Our network filter/detector loads signatures and finds injection points in the document. However, there might be a large number of signatures which don't need to be loaded for a specific website. For example, if several signatures are designed for pages running a WordPress plugin, then the extension need not check any of these for a site which is not running WordPress at all. On the other hand,

if a site is running WordPress, we might have to check all signatures meant for WordPress, but not others. Therefore, when loading signatures, we proceed in a manner similar to a decision tree. The detector first probes the page to identify the underlying framework (we call this the 'software' in our signature language). These are usually found by hints in the document HTML. These probes are framework-specific, and as such, need to be encoded in some way so that the detector can run them. There are two approaches for this: the detector completely takes care of this, and needs to be maintained for changes in frameworks and future technologies, or, the signature developer additionally specifies a more specialized version detector as part of a probe file. We chose to go with the first option, as this provides a greater ease of use for signature developers. In our prototype implementation, hard-coding probes into the detector did not imply a substantial amount of work. However, as more signatures are written and more applications are required to be included, this can become an arduous task. We believe the second option can be desirable and would not be a terrible burden for signature developers: for example, the widely popular network mapping tool Nmap [11] uses probes in a similar manner, and these are kept in a modifiable file so that advanced users can have more expressibility. After running these probes, the detector loads signatures for the specific software (e.g. all signatures for WordPress web sites).

At this point, we filter out the ones that do not apply to the current page by iterating through the list of signatures sequentially. The detector checks whether the identifying information of the signature matches the given HTML string and/or URL.

3.2 Version identification

Finally, we apply version identification. Our objective for versioning is that our signatures don't trigger any false positives on websites running patched software. We found this to be one of the harder aspects of signature loading. In WordPress, for example, many of the plugins do not update their file names with the latest versions, or do not include them at all, and thus, this information is often hard to come by from the client-side perspective. In the case of WordPress, the wp-admin/plugins.php subpath contains information about all currently active plugins on the site. Unfortunately, this information is only available to admins of the site. While this might not be the bulk of users, it is, on the other hand, the bulk of disclosed CVEs, as described in Section 4. Furthermore, we believe that even if we load a signature when the application has already been fixed at the server-side, it will often preserve the page's functionality, as many of the CVEs describe XSS which happens as a result of unsanitized input that was not meant to be JavaScript code regardless. Motivated by this observed behaviour, our mechanism follows a series of increasingly accurate but less applicable version identifiers: first, we apply general-purpose version probes, like the one described for WordPress (these are maintained in a similar way to software probes, hard-coded in the detector logic). If these are not successful, the signature language provides functionality for version identification in the HTML through regex. If the developer considers version information to be unavailable through the HTML, the version in the signature is left blank and the detector applies the signature patch regardless of version, as we can not be sure the page is running patched software.

3.3 Dynamic injections

Some of the exploits manifest themselves through dynamically loaded files. For example, CVE-2018-7747 had XSS triggered when the user loaded information stored in the plugin's database after clicking on an element of the page. Since this was not loaded with the original HTML, it came as a response to an Ajax request. Our signature language provides functionality to protect against these kinds of exploits, as shown in the example below:

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

This signature describes an exploit on a WordPress site running the Caldera Forms plugin. The XSS occurs in the specified url. The listenerData attribute defines an extra listener to attach in the background page of the extension. In this case, the page listens for an XHR, specifically done as a POST to the specified subdomain listenerUrl. The rest of the information is similar to a regular signature, as it will execute the filter and sanitize the response if necessary, according to the specified endpoints. The background page knows to only filter such requests originated from the correct web page. The type of resource to listen for is taken as specified by the webRequest API.

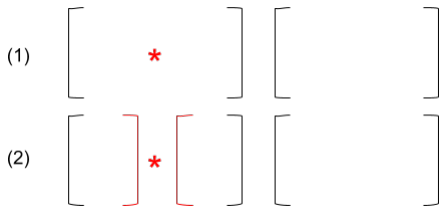


Figure 3: Example attacker injection when multiple injection points exist in the page. (1) shows a basic injection pattern. (2) shows an attempt to fool the detector.

3.4 Handling multiple injections in one page

In the previous example, 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 php 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.

We illustrate one case of this mechanism using Figure 3 as a visual aid: In (1), the content in black is a template. The content in between the brackets 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 (called the endPoints), represented by the brackets. This HTML originally has two pairs of endPoint patterns.

In (2), the attacker knows these are being used as injection endPoints and decides to inject a fake ending point and a fake starting point, with some additional malicious content in between (shown in red). If the detector were to look for several pairs of endPoints, it would not be able to tell the difference between the red and black patterns, even when using our top-down, bottom-up approach, and would not be able to get rid of the content injected in the red star (*). Therefore, we have to use first starting point and the last ending point and sanitize everything in between. Depending on the application, this might get rid of a substantial amount of valid HTML, in particular, the web page's functionality might be affected. Note that due to the difficulty of applying a general solution to this scenario, we defer to the signature developer's judgement, of what behaviour the detector should follow. We expand upon this further in Section 5.

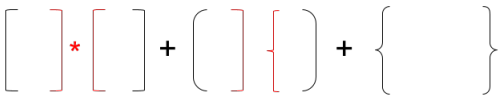


Figure 4: Example attacker injection when multiple distinct injection points exist in the page

Figure 4 illustrates a case when there are several injection points in one page, but each of them is distinct. Now, because the filter is only looking for one pair of brackets, the attacker can't fool the extension into leaving part of the injection unsanitized. However, they could inject an extra ending bracket after the opening parenthesis, or similarly, an extra opening brace before the ending parenthesis. In either case, the extension will be tricked into sanitizing non-malicious content, the black pluses (+). We can detect this behaviour by noting that we know the order in which the endPoints should appear, and so if 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 is brackets, parenthesis, braces. When it sees a closing bracket after an opening parenthesis, and similarly, when it sees an opening brace before a closing parenthesis, this represents the described attack. As with the previous scenario, we can not easily identify which endPoint is the real one, so sanitization will still potentially get rid of non-malicious content. As before, the signature developer specifies whether the page should be blocked or sanitization goes through as usual when this behaviour is detected.

While our prototype's signature database is currently encoded entirely in the extension software, we envision that a trusted entity can maintain this database, such as those that currently maintain CVE databases, and will have to be audited so that it is not filled with false signatures and existing ones are not compromised.

4 APPROACH VIABILITY

In order to verify the applicability of our detector and signature language, we tested the system by looking at several recent CVEs related to XSS. Our objective is two-fold: to verify that our signature language provides the necessary functionality to express an exploit and its patch, and to test our detector against existing exploits.

4.1 Test methodology

In order to achieve a comprehensive test suite, we looked at the 100 most recent CVEs, as of October 2018, related to XSS attacks, and in particular, to WordPress. We have chosen WordPress because it is widely used, and because it is relatively efficient and easy to reproduce many of the attacks related to it: WordPress plugins are very popular among developers (there are currently more than 55,000 plugins [19]) and there's many of these that have been found to be vulnerable to XSS [22]; using one framework, we can install many different plugins for the version we want, reproduce attacks, and investigate the conditions under which they happen, without having to install additional software. We have chosen to use a CVE database, CVE Details [21], as opposed to other databases that include vulnerabilities or exploits, mainly because of reliability. We have been able to find hundreds of verified attacks on WordPress and its plugins using a CVE database, which also usually contain information on how to reproduce them. This provides the perfect platform to analyze XSS attacks and decide whether they can be countered by our approach.

For each CVE, we set up a Docker container with a clean installation of WordPress 5.2 and installed the vulnerable plugin's version. A few of the CVEs depended on the WordPress version as well, and so we used the required WordPress version for those. We then tried to reproduce the exploit as described by the CVE author. Finally, we analyzed the vulnerable page and wrote a signature to patch the exploit.

4.2 Results

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

Table 1: Most popular studied WordPress plugins

Of the initial 100 CVEs, we were able to analyze 76 across 40 affected pages. We dropped 24 CVEs due to reproducibility issues: some of the descriptions did not include a proof of concept of the exploit, and as such, was 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 exacerbates the importance of being able to defend against

these attacks. This is not to say, however, that our detector would not work for such a CVE, as the author would have a better idea of how the exploit manifests itself, and would therefore be in a better position to write a signature. The plugins we studied averaged 489,927 installations, 1 shows the number of installations for the 5 most popular studied plugins. 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 users of all types.

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 reveals the existence of an attack e.g:

$http : //[pathtoWordPress]/wp-admin/admin.php?page = wps\_pages\_page\&page-uri =< script > alert(”XSS”) < /script >$

While Chrome's built-in XSS auditor blocked this request, Firefox did not, and so we still wrote signatures for such attacks. We wrote 59 WordPress signatures in total, which got rid of the PoC exploit when sanitized with one of our three methods. We were able to include several CVEs in some of them 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. For cases like these, the signature developer can weigh the trade-offs and decide whether the added cost is worth it.

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, modifying the page's visual aspect. One caused a small part of the page unusable, due to the sanitization method used (e.g. a table showing user information is now rendered as blank). As with the case of unsigned CVEs, most of the responsibility of maintaining functionality is left to the signature developer, being as precise as possible is key: A full sanitization of the whole HTML string will most likely get rid of any exploits, but will also make the page completely unusable in most scenarios.

While our goal with the signature language is to retain as much information of the webpage as possible after sanitization, we believe that even if a part of the page is now useless, this does not impact the user's experience as much, since most of these exploits manifest themselves in small sections of the HTML. A thorough study with regards to usability is out of scope for this work, but we provide a study on false positives and false negatives in later sections, which is related to this issue.

5 WRITING SIGNATURES

We envision the process of signature development to be part of the vulnerability discovery work flow and CVE creation. As such, we expect a signature developer to have a solid understanding of the principles behind XSS so that they can properly identify the minimal section of the DOM which acts as an injection point. Similarly, it is important that they can identify unique traits of elements (e.g. HTML id's, classes, etc.) and the page overall in order to reduce the rate of false positives. In this section, we aim to show that minor effort is required from a knowledgeable security analyst when writing a signature for our extension.

5.1 Signature language specification

We first provide a general description of a signature, in particular in the context of WordPress:

- url: If the exploit occurs in a specific URL or subdomain, this is defined as a string, e.g. /wp-admin/options-general.php?page=relevanssi%2Frelevanssi.php, otherwise null.
- software: The software framework the page is running if any, e.g. WordPress. A hand-crafted page might not have any identifiable software.
- softwareDetails: If running any software, this provides further information about when to load a signature. For WordPress, these are plugin names as depicted in the HTML of a page running such plugin.
- version: The version number of the software/plugin/page. This is used for versioning as described earlier.
- type: A string describing the signature type. A value of "string" describes a basic signature. A value of 'listener' describes a signature which requires an additional listener in the background page for network requests.
- sanitizer: A string with one of the following values: "DOMPurify", "escape", and "regex". This item is optional, the default is DOMPurify.
- config: The config parameters to go along with the chosen sanitizer, if necessary. For "DOMPurify", the accepted values are as defined by the DOMPurify API (i.e DOMPurify.sanitize(dirty, config). For "escape", an additional escaping pattern can be provided. For "regex", this should be the pattern to match with the injection point content.
- typeDet: A string with the following pattern: 'occurence-uniqueness'. As described in Section 2.4, this specifies whether there are several injection points in the HTML.
- endPoints: An array of startpoint and endpoint tuples
- endPointsPositions: An array of integer tuples. These are optional but useful when the one of the endPoints HTML are used throughout the whole page and appear a fixed number of times. For example: if an injection ending point happens on an element <h3 class='my-header'>, this element might have 10 appearances throughout the page. However, only the 4th is an injection ending point. The signature would specify the second element of the tuple to be 7, as it would be the 7th such item in a regex match array (using 1-based indexing), counting from the bottom up. For ending points, we have to count from the bottom up because the attacker can inject arbitrarily many of these elements before it, and vice versa for starting points.

Additionally, if the value of type is 'listener', the signature will have an additional field called listenerData. Similarly, to a regular signature, this consists of the following pieces of information:

- listenerType: The type of network listener as defined by the WebRequest API (e.g. 'script, 'XHR', etc.)
- listenerMethod: The request's HTTP method, for example "GET" or "POST".
- url: the URL of the request target.

5.2 Signature writing process

We now describe the process by which a signature is written after a vulnerability has been discovered:

(1) The signature developer crafts a proof of concept exploit for the given CVE. This step is not necessary but it will often help the developer correctly identify the affected areas of the DOM. For our own signatures, we heavily relied on this part because we often did not have the same information as the CVE writer. Conversely, a knowledgeable analyst will often be able to identify the vulnerable points in the application from the server-side code.
(2) Using information about where in the HTML the exploit will manifest itself, the developer identifies the start and end points of an injection, and creates regexes to match these. This step is particularly important because this is where the signature might end up covering a bigger part of the DOM than is required, potentially disabling desired functionality, to the detriment of the site user's experience. In particular, when multiple injection points occur in the same page, the developer might find it best to completely stop the page from loading if they think the sanitization would affect the user experience too greatly. While one of our main goals is to maintain the page's usability, there are cases where a large portion of the document would be affected by the sanitization, and we believe compromising usability for security is preferable in this case.
Furthermore, it is at this point where the developer identifies whether the exploit comes in from an external source (such as a response to an Ajax request or an external script) or is embedded in the document's mainframe HTML. This will result in a different signature layout.
(3) Signatures are loaded for specific pages, and the developer has to specify this information, either via an URL or a regex in the HTML. For example, for a WordPress plugin, the exploit might happen in localhost/plugin-name.php. However, for another exploit, the exploit might occur in a page where the plugin is loaded, which contains the string "wp-content/plugins/plugin-name" in the HTML. Additionally, if the webpage is running pre-defined software, such as WordPress, this has to be specified in the signature as well. Much of this information is already known beforehand, and so this step can be done in conjunction with Step 2.
(4) After the signature has been written, the developer should make sure it was correctly specified. This is most easily done via testing a PoC exploit and verifying the injection has been properly sanitized. For false positives, the developer should make sure that the specified endPoints are uniquely occurring in the HTML (or if not unique, their correct positions have been stated). The browser extension can be used for the purposes of debugging. Some of our most common mistakes when writing signatures were incorrect regexes for the endpoints, and not correctly identifying that the injection occurred as part of an additional network request. These two can be easily fixed by looking through the incoming HTML in the background page's filter.

5.3 Case Study: CVE-2018-10309

Going back to our example in Section 2.4, we describe the full process of writing a signature for one of the CVEs we studied. An entry in Exploit Database [18] describes a persistent XSS vulnerability in the WordPress plugin Responsive Cookie Consent for versions 1.7/1.6/1.5. This particular entry (as most do) comes with a proof of concept for the exploit:

(1) Access WordPress control panel.
(2) Navigate to the Responsive Cookie Consent plugin page.
(3) Select one of the input fields. For example, "Cookie Bar Border Bottom Size".
(4) Insert the script you wish to inject.
(5) Save the plugin settings.
(6) Injected script will run in the victim's browser. Depending on which input field you inserted the script, the script may also run everytime you load the Responsive Cookie Consent plugin page.

As described in Section 4.1, in order to test this vulnerability, we find a link to the affected plugin code, and launch a container a clean installation of WordPress 5.2 with the plugin downloaded. After this, we activate the plugin and proceed to reproduce the proof of concept as described in the Exploit Database entry, inserting the string '"><script>alert('XSS')</script>' in the rcc_settings[border-size] input field, resulting in the following HTML displayed on the page, as well as an alert box popping up in the page:

```
<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]">
```

In this case, it is clear that the input element is the injection starting point, and we use the label element as the end point, since it is the immediate element after the input. With this information, we are now ready to start writing the corresponding signature:

```
url: 'wp-admin/options-general.php?page=rcc-settings',
software: 'WordPress',
softwareDetails: 'responsive-cookie-consent',
version: '1.7',
type: 'string',
typeDet: 'single-unique',
endPoints:
['<input id="rcc_settings[border-size]"
name="rcc_settings[border-size]" type="text"',
'<label class="description"
for="rcc_settings[border-size]">']
```

The URL is acquired by noting that this exploit occurs on the plugin's settings page, which is in a specific subdomain of the web site. Of course, the software running is WordPress in this case. The settings page's HTML includes a link to a stylesheet with href "http://localhost:8080/wp-content/plugins/responsive-cookie-consent/includes/css/options-page.css?ver=5.2.2", in particular, "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". While the entry only lists versions 1.7, 1.6, and 1.5 as vulnerable, we apply the signature for all versions less

than or equal to 1.7. Since, the exploit only occurs in this specific spot in the HTML, the typeDet is listed as "single-unique". Finally, we list the endPoints as taken from the HTML.

Finally, we load up our extension and reload the web page. In this example, we expect to not have an alert box pop up, and we manually look at the HTML to verify correct sanitization. Note that there's nothing else in between the input and label elements now:

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

6 PERFORMANCE EVALUATION

The extension's performance goals are to provide our security guarantees without being a detriment to the end user's browsing experience. To this end, we take several timestamps throughout our code's execution. These were recorded using the Performance Web API. Note that while this API normally reports values as doubles, due to recent security threats, such as Spectre [29], several browser developers have implemented countermeasures by reducing the precision of the DOMHighResTimeStamp resolution [5, 15]. In particular, Firefox reports these values as integer milliseconds. For our tests, we re-enabled higher precision values.

While our extension's functionality only applies at the network level, there is potential slowdown at the DOM processing level due to the optimization techniques the browser applies throughout several levels of the web page load pipeline. Figure 5 shows the different timestamps provided by the Navigation Timing API [10], as well as a high-level description of the browser processing model. Since our filter listens on the onBeforeRequest event, none of the previous steps before Request are affected. In this section, we refer to the difference in time between responseEnd and requestStart as the "network filter time".

6.1 Top websites load times

We first report our extension's impact on top website load times, representing the expected behaviour of an user's average web browsing experience. For these tests, we first took the top 500 websites as reported by Moz.com [1]. For each website, we loaded it 25 times (with a 25 second timeout) and recorded the following values: requestStart, responseStart, responseEnd, domContentLoadedEventEnd, domComplete, duration, and decodedBodySize. From the initial 500, we only report values for 441 of them. The other 59 had consistent issues with timeouts, insecure certificates, and network errors. For our setup, we used a headless version of Firefox 6.90, and Selenium WebDriver for NodeJS, with GeckoDriver. All experiments were ran on one machine with an Intel Xeon CPU E5-2407 2.40GHz processor, 32 GB DRAM, and our university's 1GiB connection. 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: Firefox is loaded with the extension installed and the web driver is re-instantiated for every page load.
- 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: Firefox is loaded with the extension installed and the same web driver is used for the page's 25 loads.

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 of each website for each of the previous and the decodedBodySize.

We compare the load times with and without the extension running 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 shows the computed results. We have zoomed in on the $[-50\%, 50\%]$ range, as this is where the grand majority of values lie. The graph shows 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 the order of seconds, and in many cases tens or hundreds of milliseconds, especially for the network component. Such values should not alter the user's experience significantly. The slowdown increases by at most 5% when we take caching into account. We expect this to be the case because the network filter probably causes the browser to use less caching, especially for the DOM component, as it might have to process it from scratch every time.

Figure 7 shows a closer look at the distribution for the cold network filter slowdown on the top sites (same values as in Figure 6. We filtered out any values above $|100|\%$, as these were mostly due to timeouts and other network delays or errors. For this component, 87.6% of the slowdown values are less than 10%.

Figure 8 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. We loaded the same websites 25 times each and calculated the median times and lengths. The blue dots are the pages for which the WordPress probes tested negative, and the green dots are the pages for which the probes tested positive: 55 in total (our implementation currently only has probes for WordPress). We applied least squares regression to calculate the shown trend lines. The Spearman's rank correlation values for no WordPress, WordPress, and overall are 0.910, 0.91, and 0.72 respectively, demonstrating positive correlation. Since both our probes and signatures use regex matching, we expect both trend lines to be linear, as seen in the graph. Recall that once a probe for a certain software passes, we perform a linear scan over the signatures for that specific software and check whether it applies to the given HTML string

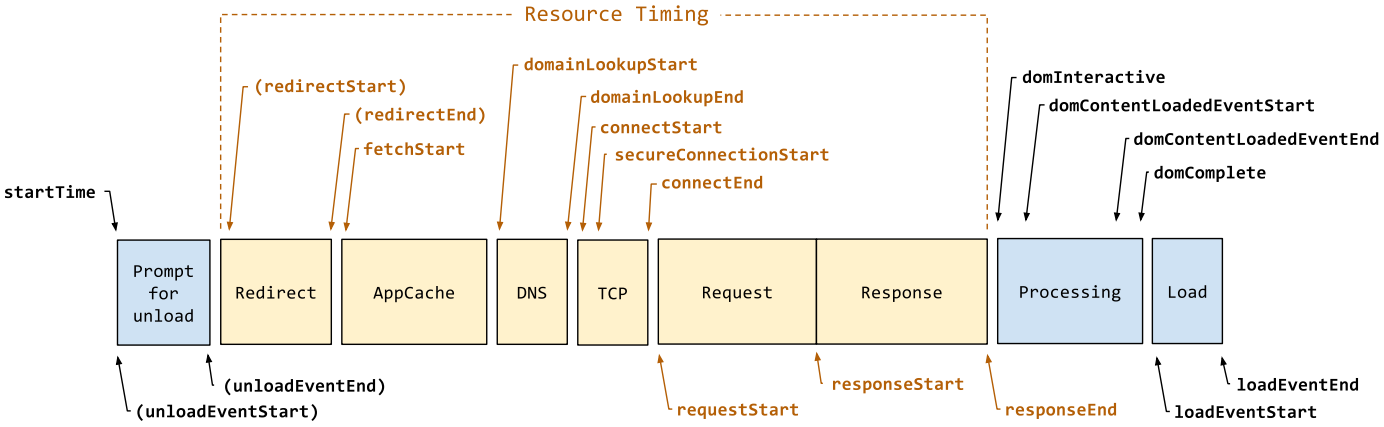


Figure 5: The Navigation Timing API's timestamps

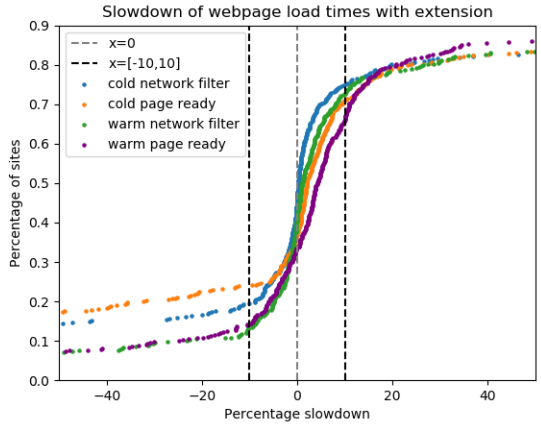


Figure 6: Cumulative distribution of relative percentage slowdown with extension installed for top sites

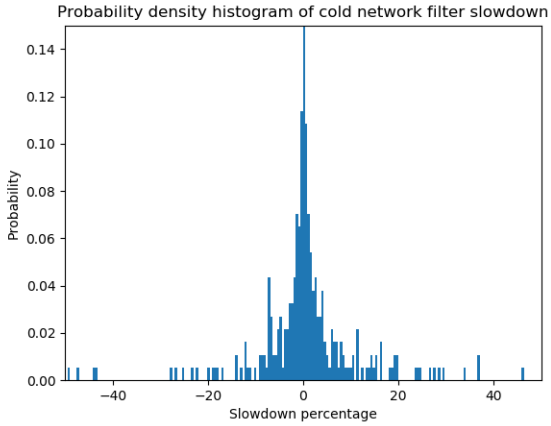


Figure 7: Density histogram of network filter slowdown for top sites

or not. Thus, we expect the slope of the line to be higher when the WordPress probe passes. Since only around 25% of sites use WordPress on average [20], we expect the impact of our network filter to be closer to the non-WordPress values, as corroborated by our overall trend line.

When factoring in the rest of the network component, additional noise is introduced in the measurements. Figure 9 shows the network filter time as a function of the page's decoded byte size. Applying least squares regression shows an upwards trend. However, the Spearman's rank correlation for this set of data is −0.054; we believe this to be due to the number of factors affecting this measurement. The data demonstrates that the size of the page being processed has a smaller impact than expected (see Figure 8) on the network filter time when the whole page load process is accounted for.

Additionally, for each website we recorded the number of loaded signatures (i.e. signatures whose endpoints were found in the HTML). We report a 0% false positive rate for loaded signatures. Thus, we can infer with confidence that the rate of false positives for loaded signatures during an average user's web browsing is similarly low. This rate could possibly go up as the number of signatures and covered frameworks increases. While we can not be sure that any of the sites is free of vulnerabilities described in our current signatures, this is very unlikely, as many of these websites are not running WordPress to begin with, and being some of the most popular, they would likely be updated relatively quickly if any vulnerability is found; thus, the rate of false negatives is likely extremely low as well.

6.2 Wordpress websites load times

We ran similar experiments as in Section 6.1, but with the WordPress sites described in Section 4.1. Thus, all of these have either one or multiple injection points in their HTML, and the network filter will spend an additional amount of time sanitizing these as defined by

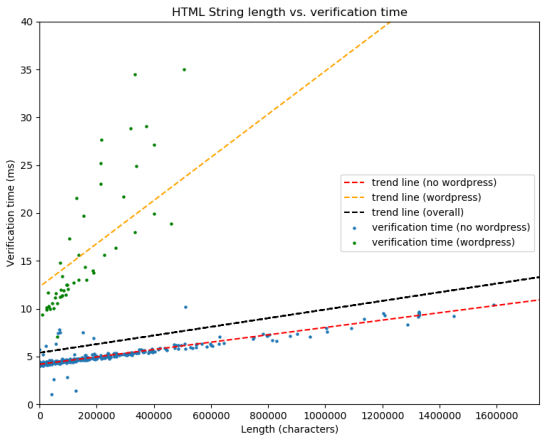


Figure 8: Scatter plot of network filter time as a function of decoded byte size for top sites

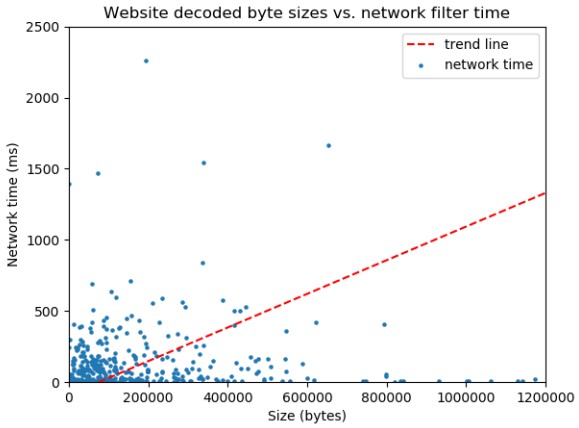


Figure 9: Scatter plot of network filter time as a function of decoded byte size for top sites

the signatures. Note that the data set is smaller here, and some of the trends might be harder to infer.

As before, Figure 10 shows the results for slowdown with the extension running. Recall that the only difference between a page which passes the WordPress probe and one that matches a signature is that the latter has to replace a portion of the original string by its sanitized version. This procedure is usually very fast, and can be faster depending on the method defined by the signature. Since even some of the top 500 sites ran WordPress, we expect a minimal difference between the two data sets. In this case we see a slowdown of less than 10% for 88.75% of sites, and less than 40% for 98.75% of them. The warm curves suffer from a smaller slowdown than the cold ones. We believe this to be the case because of the very small load times when caching is taken into account due to these pages being locally hosted, thus even minor changes in load times result in higher absolute slowdowns.

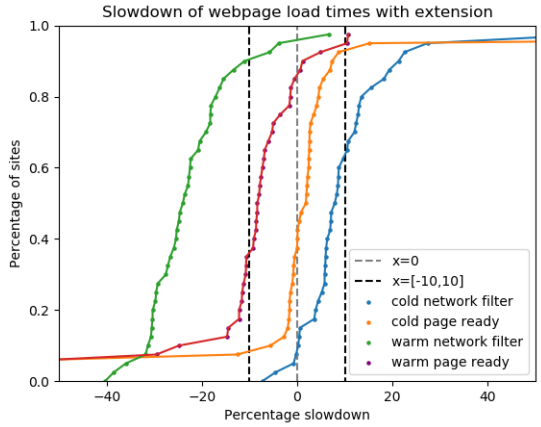


Figure 10: Cumulative distribution of relative percentage slowdown with extension installed for WordPress sites

Figure 11 shows the probability density of the cold network filter slowdown. In this case, we see that the distribution is skewed more towards a higher slowdown. As it is harder to discern the trend for this data set than its top site counterpart, we have also plotted the normal distribution of the data between 3 standard deviations. 65% of values are less than 10%.

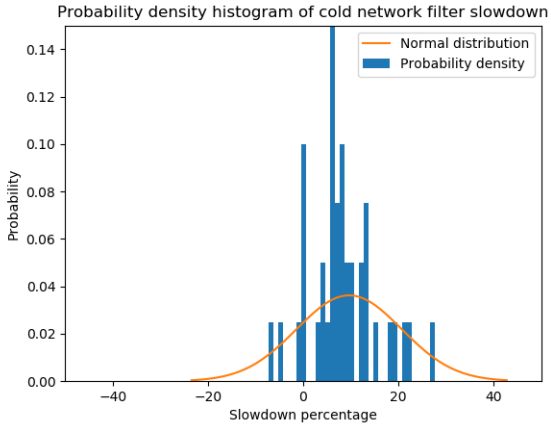


Figure 11: Density histogram of network filter slowdown for top sites

Finally, as in Figure 8, we report the string verification time as a function of its length, for the WordPress sites, shown in Figure 12. The Spearman's rank correlation for this set of data is 0.630.

7 LIMITATIONS AND FUTURE WORK

Generalizability. Our study has only covered WordPress websites. While many websites, in particular ones that use any kind of CMS

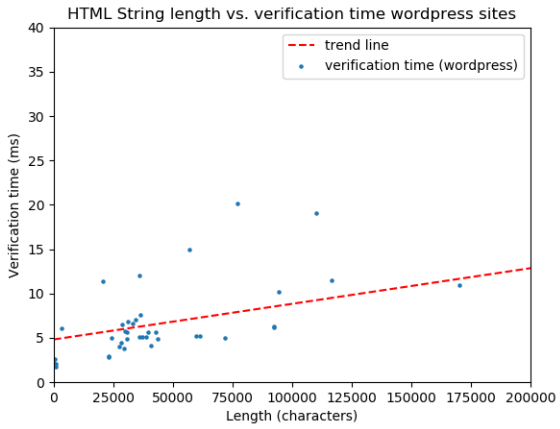


Figure 12: Scatter plot of network filter time as a function of decoded byte size for top sites

might share similar structures to the ones we studied, it is clear that the open source nature and availability of WordPress code and its plugins might have made our assumption of full knowledge of the HTML too strong. We acknowledge that this assumption will not always hold true, however, many websites will still be able to benefit from our approach.

Scope of study. Our current CVE study has only covered 100 CVEs, 23 of which had to be discarded in our result analysis. We intend to cover more in the future to have a better representation of WordPress websites and plugins and the web as a whole.

Current implementation. There are several aspects of injections that we have not been able to test as a result of the CVEs not exploiting these, like attacks through images. Our prototype implementation lacks several performance enhancements which would greatly benefit the overhead caused by the extension, like parallelizing signature loading and sanitization, as well as using more efficient data structures to store signatures.

False positives and false negatives. Due to the nature of our approach, it is nigh impossible to completely get rid of false positives. While we concluded that the rate of false positives is very low for loaded signatures, these can still occur when a signature is correctly loaded, since the sanitization might erroneously delete valid elements. We aim to reproduce the developer's intention with regards to when scripts should be able to run, however, this won't always be possible, as there will be cases where there are injection points where non-malicious JavaScript is allowed. If the sanitization applied targets JavaScript code, for example, a false positive will likely be triggered.

Furthermore, since we rely on handwritten signatures to defend against attacks, exploits for which no signature has been written are not defended against, so it is possible that not every single injection point of every website will be covered, and there might also be false negatives. In the future, we intend to study the rate of false positives and negatives in our approach and compare it to previous work. Faulty sanitization could be reduced by implementing our sanitization methods as a lexer instead of the declarative version we currently have: The signature developer would ideally provide the allowed behaviour in a given injection point, and the detector would check the injection content against the signature-specified behaviour, providing a proof of whether the content could have been generated by following the rules. If the sanitizer determines that the content could not have been produced via the allowed set of rules, it would proceed to eliminate either the whole content itself, or the part of the content which is invalid.

Usability. A main aspect of our work is its increased potential for usability and adoption from both an user's perspective that installs the extension to defend themselves against XSS, and a signature developer that has to write the database descriptions according to a known CVE. Future work could focus on usability studies related to both of these components.

Protection against CSRF. While our work has only focused on XSS, we believe we can easily adapt our network filter to defend against CSRF exploits as well. Using a similar signature language as the one for XSS, a signature developer could specify pages with potential vulnerabilities and enforce network requests that can not exploit such vulnerabilities. In some scenarios, it might be useful to track DOM events, e.g. user clicks to protect against clickjacking attacks. This kind of information is not available to the network; however, the extension could install a script on the page to monitor these actions.

Dealing with an increasing amount of signatures. Our naive approach at signature filtering, once a probe for a framework passes, has been effective with our current amount of signatures. However, as the number of signatures increases, and more types of sites are covered, the performance impact may increase drastically. As of September 2019, CVE Details had 14894 XSS vulnerabilities in its database [3]. This means that our current signatures only cover 0.5% of all existing XSS vulnerabilities. More efficient approaches at searching and filtering, as well as for the data structures used in the signature database could be applied to maintain a low performance overhead. The task of signature filtering, for example, is embarrassingly parallel in the number of signatures. Most users will visit the same pages numerous times. Caching certain aspects of the filtering process can result in a improved performance over several loads: for example, if a page does not run WordPress, it is unlikely to change this in the near future. In this case, we could cache the probing component and skip it entirely for the next load.

Design considerations. Our current design allows a single filter to be used to protect multiple users. In our current implementation, each browser user has to install our extension. However, the same functionality could be offloaded to a single processing unit in the form of a proxy, which handles the filtering for all machines in a network. This deployment model might be more feasible for certain environments.

While any user could alter their local signature database, most will not have the technical expertise to do so. However, a trusted community of users could flag websites as being potentially malicious as part of a separate database, saving time for others.

We continue working on the system to become more robust both in terms of being able to defend against a myriad of attacks, as well as providing ease of development for signatures.

8 RELATED WORK

In the following sections, we discuss related work and how it compares with our own. We highlight the similarities and how our work improves upon these previous ideas. We classify existing work into multiple categories: client-side, server-side, browser built-in, and hybrid: a combination of these.

8.1 Server-side techniques

In addition to existing parameter sanitization techniques, taint-tracking has been proposed as a means to consolidate sanitization of vulnerable parameters [24, 32, 33, 39]. These techniques are complementary to ours and, provide an additional line of defence against XSS. However, many of them rely on the client-side rendering to maintain the server-side properties, which will not always be the case.

Furthermore, these defences do not protect against purely client-side XSS, such as reflected XSS, or persistent client-side XSS, which uses a browser's local storage or cookies as an attack vector. Steffens et al. [35] 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. While some of these exploits are harder to carry out in practice, as many as 6% of these sites are vulnerable to an attack by a realistic attacker.

Our framework is not particular to client or server-side XSS as long as the exploit string appears in the HTML of the web site.

8.2 Client-side techniques

There has been previous work in client-side defenses against XSS, our work is not novel in this respect. Noxes [28] presents a similar approach as a client-side firewall-based network proxy. Rules dictate the links which can be accessed by a website when generating requests, and can be created both automatically and manually by an user. This technique does not protect against same-service attacks, such as code deleting local files. Furthermore, they rely on websites having a small amount of external dynamic links to third-parties. This likely does not hold true anymore, as websites require an ever-increasing amount of dynamic content, with several interconnections with third-parties, such as advertisement, analytics, and other user interactions.

DOMPurify [26] presents a robust XSS filter that works purely on the client-side. The authors argue that the DOM is the ideal place for sanitization to occur. While we agree with this view, their work relies on application developers to adopt their filter and modify their code to use it. This is a problem because developers might not be aware of vulnerable points in their application beforehand. In our study, we saw many instances of input parameters lacking basic sanitization. Thus, this technique is complementary to ours, and we have decided to use the DOMPurify filter for our injection points. We also believe the API is straightforward and simple to use, and won't require much signature developer effort to use effectively.

Jim et al. [27] present a method to defend against injection attacks through Browser-Enforced Embedded Policies. This approach is similar to ours, as the policies specify places where script execution should not occur. However, policies are defined by the application developers, and this again relies on them to know where their code might be vulnerable. Furthermore, browser modifications are required to benefit from it, and issues of cross-portability and backwards compatibility arise.

Hallaraker and Vigna [25] use a policy language to detect malicious code on the client-side. Similarly to us, they make use of signatures to protect against known types of exploits. However, unlike our approach, their signatures are not application-level, and there is no model for signature maintenance. Furthermore, there is no evaluation on the efficacy of their signatures.

Although not solely related to XSS, Snyder et al. [34] report a study in which they disable several JavaScript APIs and tests the amount of websites that are clearly non-functional without the full functionality of the APIs. They present a novel technique to interpose on JavaScript execution via the use of ES6 Proxies, allowing for efficient trapping of function calls. This approach increases security due to the number of vulnerabilities present in several JavaScript APIs, however, we believe disabling whole aspects of API functionality should only be used as a last resort.

8.3 Browser built-in defences

Browsers are equipped with several built-in defences:

- URL filters, such as Chrome's XSS auditor attempt to protect from certain attacks like reflected XSS. Unfortunately, these filters are often plagued by false positives and bypasses.
- Content Security Policy (CSP) [14] has been widely adopted and in many cases provides developers with a reliable way to protect against XSS and Cross-site request forgery (CSRF) attacks. However, CSP requires the developer to know which scripts might be malicious. This is particularly hard in the case of inline scripts, like the ones used in many XSS exploits.
- Same-origin policy [8] is another useful security mechanism for protection against XSS and CSRF. This policy restricts how a document or script loaded from one origin can interact with a resource from another origin. This is useful in many attack scenarios, particularly against CSRF in cross-origin attacks. As with CSP, if the attack is injected in the same website the attacker intends to compromise, this will not defend against it.

As with other approaches these browser defences are complementary to ours.

8.4 Client and server hybrids

Nadji et al. [31] make use of a hybrid approach to XSS defences. They use sever-specified policies that are enforced on the client-side. Unlike previous work, they do not rely on developers to identify untrusted sources, and tag elements server-side, such that the client-side has a clear distinction of untrusted code and can filter it accordingly. Our own tagging mechanism is partly inspired by this, as injection point markers are identified based on surrounding elements, but we do not rely on the server passing this information along, and thus is less precise.

XSS-Dec [37] also employs a hybrid solution, mainly via the use of a proxy, which keeps track of an encrypted version of the server's source files, and uses this information to derive exploits in a page visited by the user. This approach is similar to ours, in the sense that we assume previous knowledge of the clean HTML document. Furthermore, they use both anomaly-based detection

and signature-based detection to calculate the likelihood of an attack taking place, and prevent it from happening. However, there is no mention of signature maintenance. In a way, our system offloads all this functionality to the client-side, without the need of any server-side information. The adoption of server-side techniques might not be feasible for many developers.

Since we don't rely on server-side techniques, our system ideally reduces the turnaround time between a vulnerability and its patch. A 2018 study found that the average time to patch a CVE regardless of severity is 38 days, increasing to as much as 54 days for low severity vulnerabilities, and the oldest unpatched CVE was 340 days old [6]. Instead, we protect from the day of vulnerability discovery.

9 CONCLUSIONS

We have presented XSnare, a fully client-side protection mechanism against XSS. This approach has many benefits over currently existing systems, as well as being complementary to many of them. Our firewall architecture makes it so that users can protect themselves in the face of an ever-increasing number of potential attacks and attack vectors, with little additional effort required for a knowledgeable security analyst when taking into consideration the existing vulnerability detection work flow. The CVE study we conducted showed that the provided API for sanitization and injection point detection is effective, as 93.4% of these vulnerabilities could be defended against with our tool. Our prototype implementation meets the required performance goals to not be detrimental to an user's web browsing experience, as web page load times are affected by less than 10% overhead on 72.6% of sites.

REFERENCES

[1] [n. d.]. Moz Top 500 Websites. ([n. d.]). Retrieved August 17, 2019 from https://moz.com/top500

[2] [n. d.]. NoScript homepage. ([n. d.]). Retrieved October 26, 2018 from https://noscript.net/

[3] [n. d.]. Vulnerabilities By Type. ([n. d.]). Retrieved September 04, 2019 from https://www.cvedetails.com/vulnerabilities-by-types.php

[4] 2016. Hacked Website Report – 2016/Q3. (2016). Retrieved April 28, 2019 from https://sucuri.net/reports/Sucuri-Hacked-Website-Report-2016Q3.pdf

[5] 2018. Reducing the precision of the DOMHighResTimeStamp resolution. (2018). Retrieved September 04, 2019 from https://github.com/w3c/hr-time/issues/56

[6] 2018. Security Report for In-Production Web Applications. (2018). Retrieved September 14, 2019 from https://www.rapid7.com/resources/security-report-for-in-production-web-applications/

[7] 2018. WordPress Plugin Responsive Cookie Consent 1.7 / 1.6 / 1.5 - (Authenticated) Persistent Cross-Site Scripting. (2018). Retrieved September 11, 2019 from https://www.exploit-db.com/exploits/44563

[8] 2019. Content Security Policy (CSP). (2019). Retrieved September 14, 2019 from https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP

[9] 2019. Intent to Deprecate and Remove: XSSAuditor. (2019). Retrieved July 24, 2019 from https://groups.google.com/a/chromium.org/forum/#!msg/blink-dev/TuYw-EZhO9g/blGViehIAwAJ

[10] 2019. Navigation Timing Level 2. (2019). Retrieved October 03, 2019 from https://www.w3.org/TR/navigation-timing-2/

[11] 2019. nMap Network Mapper. (2019). Retrieved June 17, 2019 from https://nmap.org/

[12] 2019. Responsive Cookie Consent 1.8 patches. (2019). Retrieved August 1, 2019 from https://plugins.trac.wordpress.org/browser/responsive-cookie-consent/tags/1.8/includes/admin-page.php

[13] 2019. Safely inserting external content into a page. (2019). Retrieved June 17, 2019 from https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Safely_inserting_external_content_into_a_page

[14] 2019. Same-origin policy. (2019). Retrieved September 14, 2019 from https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy

[15] 2019. Security and privacy considerations for DOMHighResTimeStamp resolution. (2019). Retrieved September 04, 2019 from https://github.com/w3c/hr-time/issues/79

[16] 2019. Statistics Show Why WordPress is a Popular Hacker Target. (2019). Retrieved April 28, 2019 from https://www.wpwhitesecurity.com/statistics-70-percent-wordpress-installations-vulnerable/

[17] 2019. webRequest. (2019). Retrieved June 17, 2019 from https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API/webRequest

[18] 2019. WordPress Plugin Responsive Cookie Consent 1.7 / 1.6 / 1.5 - (Authenticated) Persistent Cross-Site Scripting. (2019). Retrieved July 8, 2019 from https://www.exploit-db.com/exploits/44563

[19] 2019. Wordpress: Plugins. (2019). Retrieved June 25, 2019 from https://wordpress.org/plugins/

[20] 2019. WordPress powers 25% of all websites. (2019). Retrieved April 28, 2019 from https://w3techs.com/blog/entry/wordpress-powers-25-percent-of-all-websites

[21] 2019. Wordpress: Vulnerability Statistics. (2019). Retrieved April 28, 2019 from https://www.cvedetails.com/product/4096/Wordpress-Wordpress.html?vendor_id=2337

[22] 2019. WPScan. (2019). Retrieved April 28, 2019 from https://wpscan.org/

[23] 2019. XSS Auditor. (2019). Retrieved September 11, 2019 from https://www.chromium.org/developers/design-documents/xss-auditor

[24] Prithvi Bisht and V. N. Venkatakrishnan. 2008. 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 (DIMVA '08)*. Springer-Verlag, Berlin, Heidelberg, 23–43. https://doi.org/10.1007/978-3-540-70542-0_2

[25] Oystein Hallaraker and Giovanni Vigna. 2005. Detecting Malicious JavaScript Code in Mozilla. In *Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '05)*. IEEE Computer Society, Washington, DC, USA, 85–94. https://doi.org/10.1109/ICECCS.2005.35

[26] Mario Heiderich, Christopher Späth, and Jörg Schwenk. 2017. DOMPurify: Client-Side Protection Against XSS and Markup Injection. In *Computer Security – ESORICS 2017*, Simon N. Foley, Dieter Gollmann, and Einar Snekkenes (Eds.). Springer International Publishing, Cham, 116–134.

[27] Trevor Jim, Nikhil Swamy, and Michael Hicks. 2007. Defeating Script Injection Attacks with Browser-enforced Embedded Policies. In *Proceedings of the 16th International Conference on World Wide Web (WWW '07)*. ACM, New York, NY, USA, 601–610. https://doi.org/10.1145/1242572.1242654

[28] Engin Kirda, Nenad Jovanovic, Christopher Kruegel, and Giovanni Vigna. 2009. Client-side Cross-site Scripting Protection. *Comput. Secur.* 28, 7 (Oct. 2009), 592–604. https://doi.org/10.1016/j.cose.2009.04.008

[29] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2018. Spectre Attacks: Exploiting Speculative Execution. *CoRR* abs/1801.01203 (2018). arXiv:1801.01203 http://arxiv.org/abs/1801.01203

[30] Ian Muscat. 2017. Acunetix Vulnerability Testing Report 2017. (jun 2017). Retrieved October 26, 2018 from https://www.acunetix.com/blog/articles/acunetix-vulnerability-testing-report-2017/

[31] Yacin Nadji, Prateek Saxena, and Dawn Song. 2009. Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense. (01 2009).

[32] Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. 2005. 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*. 295–308.

[33] Tadeusz Pietraszek and Chris Vanden Berghe. 2006. Defending Against Injection Attacks Through Context-sensitive String Evaluation. In *Proceedings of the 8th International Conference on Recent Advances in Intrusion Detection (RAID'05)*. Springer-Verlag, Berlin, Heidelberg, 124–145. https://doi.org/10.1007/11663812_7

[34] Peter Snyder, Cynthia Taylor, and Chris Kanich. 2017. Most Websites Don'T Need to Vibrate: A Cost-Benefit Approach to Improving Browser Security. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. ACM, New York, NY, USA, 179–194. https://doi.org/10.1145/3133956.3133966

[35] Marius Steffens, Christian Rossow, Martin Johns, and Ben Stock. 2019. Don't Trust The Locals: Investigating the Prevalence of Persistent Client-Side Cross-Site Scripting in the Wild. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*.

[36] Ben Stock, Martin Johns, Marius Steffens, and Michael Backes. 2017. How the Web Tangled Itself: Uncovering the History of Client-side Web (in)Security. In *Proceedings of the 26th USENIX Conference on Security Symposium (SEC'17)*. USENIX Association, Berkeley, CA, USA, 971–987. http://dl.acm.org/citation.cfm?id=3241189.3241265

[37] Smitha Sundareswaran and Anna Cinzia Squicciarini. 2012. XSS-Dec: A Hybrid Solution to Mitigate Cross-Site Scripting Attacks. In *Data and Applications Security and Privacy XXVI*, Nora Cuppens-Boulahia, Frédéric Cuppens, and Joaquin Garcia-Alfaro (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 223–238.

[38] P. Wurzinger, C. Platzer, C. Ludl, E. Kirda, and C. Kruegel. 2009. SWAP: Mitigating XSS Attacks Using a Reverse Proxy. In *Proceedings of the 2009 ICSE Workshop on Software Engineering for Secure Systems (IWSESS '09)*. IEEE Computer Society, Washington, DC, USA, 33–39. https://doi.org/10.1109/IWSESS.2009.5068456

[39] Wei Xu, Sandeep Bhatkar, and R. Sekar. 2006. 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 (USENIX-SS'06)*. USENIX Association, Berkeley, CA, USA, Article 9. http://dl.acm.org/citation.cfm?id=1267336.1267345