DOM Firewall: Client-side Cross-site Scripting Detection

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 DOM Firewall, a fully client-side 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, and uses a database of exploit descriptions to prevent them, effectively singling out injection points in the HTML, and preventing malicious code from executing in these. This tool 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 most of these exploits. Initial peformance evaluation has resulted in an overhead of at most 2x on webpage load times.

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 [17]. 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 [15, 18]; or require additional developer effort [14]. Still, others disable client-side functionality [1, 21], 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, and they are able to fix these vulnerabilities, 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 [2], and another found that 30.95% of Alexa's top 1 Million sites run a vulnerable version of WordPress [6] TODO: how many of these can run without JS?. 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 [22]), 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. Furthermore, 30.3% of the Alexa top 1000 sites use WordPress [10]. Thus, we can be confident that our study results will hold true to for the average user.
- Due to its user popularity, it is also heavily analysed by security experts, as has been previously stated. There are currently 286 CVEs related to WordPress in the CVE Details database [11]. Plugins, specifically, are an important part of this issue, 52% of the vulnerabilities reported by WPScan are caused by WordPress plugins [12].
- 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, as long as 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 (e.g. blacklisting in the NoScript browser extension). 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. TODO: expand on this, explain why DOM is the right place.

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 THE DOM FIREWALL

We now present the DOM Firewall's components and how they interact with each other. First, we give a overview of current defence solutions and how they 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 existing 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, there is a separate networking component. At this point, 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.

It should be noted that 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" [3].

Our approach, instead, focuses on application-specific detection at the client-side layer, and thus, doesn't rely on any server-side infrastructure (or its operators) and is more accurate than many client-side solutions. 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.

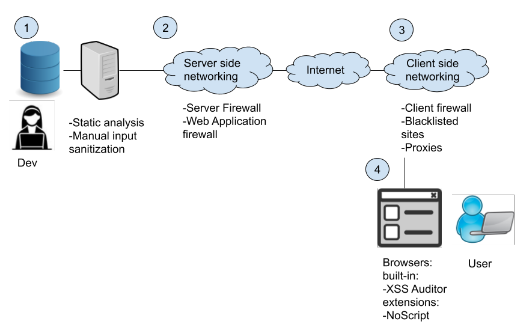


Figure 1: Architecture of typical web applications. Different security solutions apply at distinct layers.

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 2 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 the potentially malicious document, doing so by loading signatures which a developer (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.

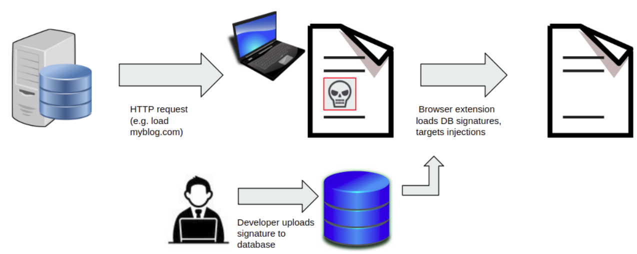


Figure 2: The DOM Firewall approach for protection against XSS.

In order to further illustrate this approach, we present a small example of how DOM context can be used to defend against XSS. This is reproducible in an off-the-box WordPress installation running the Responsive Cookie Consent plugin, v1.8, and Chrome's XSS auditor does not protect against this. 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. TODO: add PHP code and compare, also talk about template vs injected content

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

Under normal circumstances, it might display something such as "0". However, if the user is malicious, it could inject a script through the border-size variable, 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.

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.

2.3 Firewall Signatures

The firewall signatures are at the core of our defense strategy. These must be precise enough for our system to single out 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 change its location without any JavaScript execution. 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. 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. TODO: feel like this could be expanded but not sure what else to say. Could talk about signature sharing for power-users to get back to the user-centred argument, although we haven't really gotten this far into deployment talk yet.

2.4 Firewall Signature Language

Our signature language needs to be such that it has enough power of expression for the developer to be as precise as they need. 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-unique',
endPoints:
['<input id="rcc_settings[border-size]"
name="rcc_settings[border-size]" type="text"',
'<label class="description"
for="rcc_settings[border-size]">']
```

Along with the previously mentioned endpoints, the signature also defines the specific url (if any) 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 plugin where the exploit occurs, as well as its version. Since we are looking at injection points in the HTML, there are different scenarios with regards to the number of places where an injection can occur. This is encoded in the "typeDet" and has the following format: **occurrence-uniqueness**. The occurrence refers to the number of CVEs in one specific page and can be either "single" or "multiple"; and the uniqueness refers to the identifiability of the endpoints, and can be either "unique" or "general". To understand this better, consider our example: we stated that the endpoints were easily identifiable and as such, the injection could only happen in one specific spot, therefore the type is single-unique. However, if the injection were to happen in an arbitrary row in a table, it's hard to determine which row is the one affected, but we know the injection is still bound to that specific table, so the type is single-unique. If a page has, for example, two different variables that are set by user input and are not sanitized, then the occurrence will be 'multiple'. These different scenarios complicate the detection mechanism and are further discussed in the implementation section.

2.5 Firefox Extension

Our extension's main purpose is to detect vulnerabilities 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 [7] . This allows the extension's background page to analyse and modify incoming network traffic. The extension therefore 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.

A network filter is particularly useful in the case of client-side XSS (e.g. DOM Based XSS), and to detect malformed HTML or detect XSS before it can rearrange itself in the HTML. 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 above the <tr> in the DOM. This defies one of our key observations with regards to injection placements. Therefore, we can't wait until the website is rendered client-side to start interposing on code execution.

This approach guarantees safety even in the face of a knowledgeable attacker: if they know what the signatures look like, they can't take advantage of this knowledge because 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. Since the injection can't be infinitely long, it can be easily distinguished from the HTML template.

While we haven't seen any examples that warrant this functionality, it is possible for an exploit to only manifest itself through dynamic behaviour, i.e. after an user clicks on the page. The network filter might not be able to defend against this, but the extension's content script can safely interpose on it through the user of event listeners and in particular Firefox's **beforescriptexecute** event, which occurs before a script element executes. Signatures can also be defined for these scenarios, to be ran in the content script. However, we believe this to be less ideal due to the added performance costs, as the extension now has to install the content script's code on all browser tabs.

3 IMPLEMENTATION

We have implemented our browser extension in Firefox 67.0.2. Our signatures are currently stored in a local JavaScript file in the extension package. Injection point sanitization is done with DOMPurify [14]. 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" [5]. 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, in some cases, it is more desirable to use a different sanitization approach, specially 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.

As previously mentioned, our 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. 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 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 [4] 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.

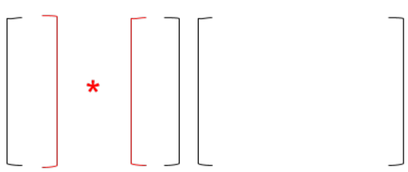
Finally, we apply version identification. Our objective for versioning is that our signatures don't trigger any false positives on websites running patched software. We found this to be one of the harder aspects of signature loading. In WordPress, for example, many of the plugins do not update their 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 more 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.

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]']
}
```

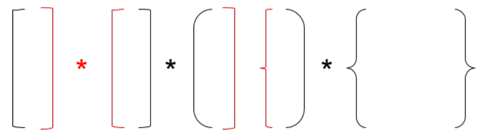
This signature describes an exploit on a WordPress site running the Caldera Forms plugin. The XSS occurs in the specified url. The listenerData attribute defines an extra listener to attach in the background page of the extension. In this case, the page listens for an XHR, specifically done as a POST 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.

In the previous example, the endPoints were listed as table rows. In this case, these row ids were not arbitrarily generated, and so we could use the "entry_row_#" id attribute to identify them. 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, as shown in the following diagram:



The content in between black brackets is an injection point. The HTML originally has two pairs of endPoint patterns. 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. Due to the difficulty of applying a general solution to this scenario, we defer to the signature developer's judgement, who can specify whether the page should not load at all as part of the signature. We believe this to be preferable to allowing malicious content to be displayed in the page.

A similar case occurs when there are several injection points in one page, but each of them is unique:



In this case, 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 stars (*). We can detect this behaviour by nothing 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 currently does not have a centralized database for signatures, 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 realted 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 test our detector against existing exploits. In this manner, we could iteratively enhance our system implementation by adding necessary features, as well as pruning unnecessary ones.

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 [9]) and there's many of these that have been found to be vulnerable to XSS [12]; 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 [11], 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

Of the initial 100, were able to write TODO: x CVEs. We dropped TODO: 100-x 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 (min: 10, max: 5 million). TODO: y of these could be exploited by an unauthenticated user; TODO: z had a high-privilege user as the victim, and TODO: w had a low-privilege user as the victim.

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 Firefox didn't block this request, Chrome's built-in XSS auditor did block it. We believe such solutions are important and are complementary to our work, and so we still tagged such attacks as identifiable by our approach, as well as having the ability to detecting them in an extension.

The majority of these signatures maintained the same layout and core functionality of the webpage. However, a small number of signatures rendered parts of the page unusable, due to the sanitization method used (e.g. a table showing user information is now rendered as blank). 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 DEVELOPING SIGNATURES

As previously stated, 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.

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 didn't have the same information as the CVE writer.
(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. 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. 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 [8] 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 (in fact, the CVE description states that this occurs "before 1.8"). 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

this is performance. and it's good.

7 LIMITATIONS AND FUTURE WORK

Generalizability. Our study has only covered WordPress websites. While many websites, particular ones that use any kind of CMS might share similar structures to the ones we studied, it is clear that the open source nature of 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 study has only covered 100 CVEs, 21 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. The DOM Firewall has only been manually tested by specially crafted signatures. We are still in the process of refining the signature language and releasing a general framework for signature descriptions and the process of uploading and downloading to the firewall database.

False positives and false negatives. Due to the nature of our approach, it is nigh impossible to completely get rid of false positives. We have previously discussed the ability 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, and thus, our system will have false positives. Furthermore, since we rely on handwritten signatures to defend against attacks, it is possible that not every single injection point of every website will be covered, and so we will also have 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.

Performance. We haven't been able to evaluate our extension's performance. The added filtering and auditing of the network responses and the DOM will incur some overhead in the a website load times, but we don't expect this to be too detrimental, as the browser APIs provide fast methods to filter requests and interpose on event loads.

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.

8 RELATED WORK

In the following sections, we discuss a number of related works and how they compare with our own. We are primarily interested in the distinction between different techniques: client-side, server-side, and a combination of these; and how they can be used in tandem with our approach.

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 [13, 19, 20, 23]. 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.

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 [16] 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 [14] 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. [15] 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.

Although not solely related to XSS, Snyder et al. [21] 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.

TODO: mention built-in browser techniques? not sure if there's a lot of research but might be worth mentioning industry solutions, i.e. chrome's XSS auditor, CSP, etc.

8.3 Client and server hybrids

Nadji et al. [18] 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 identified 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, but we do not rely on the server passing this information along, and thus is less effective. However, as previously mentioned, the adoption of server-side techniques might not be feasible for many developers.

9 CONCLUSIONS

We have presented a fully client-side solution to the XSS problem. 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 very little additional user effort required. The study we conducted shows that a majority of websites can benefit from our defence strategy and, thus, we conclude that this is a viable system. Our implementation is still a work in progress and will continue 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 database signatures.

REFERENCES

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

[2] 2016. Hacked Website Report – 2016/Q3. (2016). Retrieved April 28, 2019 from https://blog.sucuri.net/2017/01/hacked-website-report-2016q3.html

[3] 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

[4] 2019. nMap Network Mapper. (2019). Retrieved June 17, 2019 from https://nmap.org/

[5] 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

[6] 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/

[7] 2019. webRequest. (2019). Retrieved June 17, 2019 from https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API/webRequest

[8] 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

[9] 2019. Wordpress: Plugins. (2019). Retrieved June 25, 2019 from https://wordpress.org/plugins/

[10] 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

[11] 2019. Wordpress: Vulnerability Statistics. (2019). Retrieved April 28, 2019 from https://www.cvedetails.com/product/4096/Wordpress-Wordpress.html?vendor_id=2337

[12] 2019. WPScan. (2019). Retrieved April 28, 2019 from https://wpscan.org/

[13] 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

[14] 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.

[15] 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

[16] 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

[17] 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/

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

[19] 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.

[20] 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

[21] 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

[22] 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

[23] 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