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 most of these exploits can be defended against with our tool. Initial performance evaluation has resulted in an overhead of at most 2x on webpage load times.

1 INTRODUCTION

Cross-site scripting (XSS) has long been one of the most dominant web vulnerabilites. In 2017, a report showed that 50% of websites were vulnerable to XSS attacks [8]. Even though many countermeasures have been developed 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 [6, 9]; or require additional developer effort [5]. Still others disable client-side functionality [1, 12], 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 benefit from these immediately: according to WordPress, only 65.9% of websites running WordPress have currently updated to the latest version [3]. 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 [13]), users are left more exposed than ever to client-side vulnerabilities.

To provide users with the means to protect themselves, a client-side solution must be delivered. The aforementioned solutions also suffer from an increased rate of false-positives and false-negatives, due to the lack of information available at these layers. In contrast, 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. Our system consists of three main components: a trusted Firefox extension for interposing between the application and the DOM, an automatically updating 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 CLIENT-SIDE FIREWALL APPROACH

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 1 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 proceeds to render.

In order to further illustrate this approach, we present a small example of how DOM context can be used to defend against XSS. 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 value 'border-size'.

```
<div class="rcc-panel group"
border-bottom:<?php rcc_value('border-size');>
```

This div element will display the border-bottom attribute that was input by the user, under normal circumstances, it might display something such as:

```
<div class="rcc-panel group"
border-bottom: 0px solid rgb(85,85,85);>
```

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:

```
<div class="rcc-panel group"
border-bottom:""
<script>alert('XSS')</script>
```

Assuming we have knowledge of how the full HTML should render as without any injections, we can single out the point of injection and get rid of the malicious script entirely. However, this might not be the case: different webpages might be running the same or similar backend, and thus the client-side rendering would differ

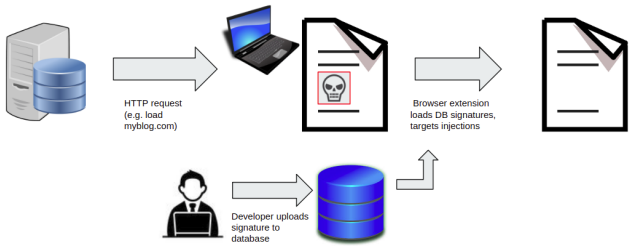

Figure 1: The DOM Firewall approach to protection against XSS.

across these. In this case, we can't be sure of where a script should be or not, potentially limiting the approach.

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.

2.1 Firewall Signatures

The firewall signatures are at the core of our defense strategy. These must be precise enough for the extension to single out the intended injection, and not an otherwise vital element of the website. 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 HTML tree; second, in a well-formed DOM, the injected element 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 it's end, and the extension will stop any JavaScript from executing in this section of the HTML. Of course, it's up to the signature developer to decide whether there should be any JavaScript executing in this location. However, it is often easy to make this decision, as there are many input parameters which clearly shouldn't execute any JavaScript.

An immediate concern that can come up is who writes these signatures. We believe CVEs to be an ideal source for these: 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.

2.2 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 should suffice to express precise sections of the HTML. Furthermore, a regex language allows us to identify malformed HTML before it renders on the browser, as described below.

2.3 Firefox Extension

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. Firefox in particular allows the interception of any script elements in the HTML, via the **beforescriptexecute** event. Thus, whenever any malicious script wants to run, we can first audit it and determine whether it should execute or not, according to rules defined by the firewall. While not all XSS is triggered by script elements, they do require JavaScript. Thus, we can modify any event listeners as well, effectively interposing on any JavaScript execution. Any required modifications of the HTML can be done in-place, as the website loads.

We have assumed that we know how the HTML will be rendered on the client-side, but even with this knowledge, there are injections which might alter the expected placement of different elements. 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. To account for this, we make use of the webRequest API provided by Firefox to filter incoming responses, tagging all elements with special identifiers, allowing us to have a similar representation of the HTML as the signature developer had in mind. This technique is susceptible to a knowledgeable attacker, who can identify all attributes of the starting and end points of the injection sections, to trick our extension into thinking that the section is already over by injecting a duplicate of the end node. To combat this, we use a top-down, bottom-up approach when looking for attacks. Our main observation in this case is that the injected code cannot be infinitely long. Thus, the real end node will eventually follow the injection, and we can backtrack the HTML tree from the end of the document to find it. Thus, an attacker cannot trick the extension into analysing a smaller section of the document.

3 APPROACH VIABILITY

Because of the binding between CVEs and firewall signatures, ideally, our approach should hold at application level, for example, a webpage running WordPress v. 4.9.8, with plugin WooCommerce v. 3.4.6. Due to the nature of websites and their complex interactions between different parts of their back-end, the assumption that we always have full knowledge of the rendering of the client-side HTML might not hold in practice. Even if this assumption held for every individual website, it might not hold true across different websites using the same services, as the aforementioned example. Both sites might be running WordPress with the same plugins but each of them might have different client-side modifications that would render our starting-node, ending-node method useless. Figure 2 gives an example of the admin panel of a WordPress site, the part enclosed in the red box is constant across any website. In contrast, Figure 3 shows the user side of this webpage, many elements of this view can be modified directly by the admin as they please. In light of this, we have conducted a study that demonstrates the applicability of our approach.

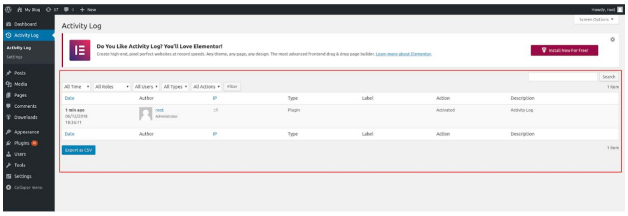

Figure 2: Admin-view of a WordPress website, the section enclosed in red is dictated by the plugin code.

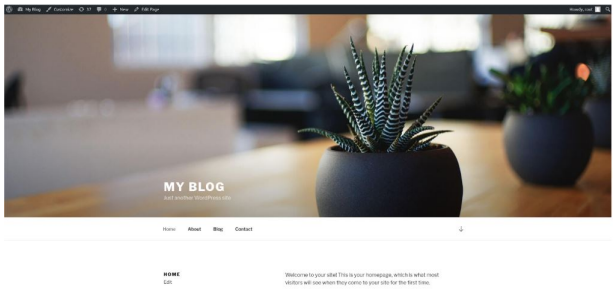

Figure 3: User view of a WordPress website, much of the layout can be modified by an admin.

3.1 Study Methodology

In order to achieve a comprehensive study, we have analyzed the 100 most recent CVEs related to XSS attacks, and in particular, to WordPress. We have chosen WordPress because it is widely used (WordPress powers 30% of the Internet according to a recent survey [2]), and because it is relatively efficient and easy to reproduce many of the attacks related to it: WordPress plugins are very popular among developers and there's many of these that have been found to be vulnerable to XSS; 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, 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.

We tagged the analyzed CVEs with four main pieces of information in a yes/no format:

- Whether they can be exploited without privileged user authentication: most WordPress sites will allow users to register, and admins of these sites can give increasingly more important privileges to them. Many exploits rely on the fact that the attacker has enough privileges to, for example, submit a post or change some website configurations. As such, we have decided to mark whether an attack requires the user to have high privileges or could just be an unauthenticated or have the most basic level of registration.
- Whether the attack manifests itself on an admin-only view: As discussed earlier, our approach relies on very specific knowledge of what the HTML document should look like without any code injections. Many WordPress plugins have an admin-only view for configuration setup where XSS manifests itself. These are of particular interest because these views do not change across different installations of the plugin, that is, different websites using the same plugin will display the same HTML in these views, and so our approach will be easier to apply.
- Whether the attack manifests itself on a user-only view: Similarly to 2, many XSS attacks on WordPress only manifest themselves on the user side of the website, thus, these are likely to vary across different webpages, even if the attack is caused by a WordPress plugin. However, it's worth mentioning that even if this is the case, all might not be lost, as the attack might happen in a very specific HTML context related to the WordPress plugin.
- Whether the attack manifests itself through some HTML element: Sometimes an XSS attack is not tightly linked to the HTML document, it might just be triggered through a POST request by clicking on a link. Attacks that are not related to HTML are not usually covered by our approach. We might be able to use some heuristics for simpler attacks, like URL filtering, but in general we do not expect to be able to guard against these.

3.2 Study Results

We classified 79 CVEs from our original 100. We dropped 21 of them due to insufficient information in the CVE descriptions to get any meaningful information, or because the plugin code was no longer available. In some cases, the plugin had been removed from the WordPress repository due to "security issues", which exacerbates the importance of being able to defend against these attacks.

The plugins we studied averaged 489,927 installations (min: 10, max: 5 million). Table 1 reports our main results according to our discussed methodology. It is clear from the results that our approach applies to a majority of the studied CVEs. Of particular importance are the first three reported figures.

A 72% in "No Authentication" implies that most attacks can be realized by an outsider, and is an alarming reflection of the current state of browser and application defenses against XSS.

The "User View Only" represents the hardest scenario for us to defend against XSS, as many of the elements in this view won't persist across different websites. A 14% here means that only a small minority of plugins won't be able to benefit from our strategy. It is worth noting that we have tagged webpages according to this criterion by barely being displayed in the user view, however, there might still be enough identifying information for the injection to be defended against, and this number might therefore not be fully representative of the limitations of our approach.

While we report a 66% in "Admin View Only", in our studies we noted that some of the CVE descriptions were somewhat limited and in some cases weren't accurate in describing where the injection points were rendered. As an example, one CVE explained how an unsanitized parameter could trigger XSS in a specific URL of the

No Authentication	Admin View Only	User View Only	HTML Context
57 (72%)	52 (66%)	11 (14%)	71 (89%)

Table 1: Study results

admin panel, but we found that there were other parts of the admin panel where the same XSS was being triggered. While we believe that this doesn't pose a big issue since it just counts as a failure of the signature description in the firewall, we note that the actual number of websites that can benefit from our approach might be slightly different.

Finally, even for plugins where we didn't consider it to be able to be defended against in the admin view, we still report the ones where we believe there is some HTML Context that identifies the injection point and can be used to defend against XSS; however, these might be more prone to false positives and therefore only consider them as a "last resort".

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.

4 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.

5 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.

5.1 Server-side techniques

In addition to existing parameter sanitization techniques, taint-tracking has been proposed as a means to consolidate sanitization of vulnerable parameters [4, 10, 11, 14]. 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.

5.2 Client-side techniques

There has been previous work in client-side defenses against XSS, our work is not novel in this respect. Noxes [7] 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 [5] 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 the

DOMPurify filter might be a good filter for injection points where no code should be running at all as expressed by our signatures.

Jim et al. [6] 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 at al. [12] 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.

5.3 Client and server hybrids

Nadji et al. [9] 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.

6 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] 2018. Usage of content management systems for websites. (2018). Retrieved October 26, 2018 from https://w3techs.com/technologies/overview/content_management/all

[3] 2018. WordPress Statistics. (2018). Retrieved October 26, 2018 from https://wordpress.org/about/stats/

[4] 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

[5] 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.

[6] 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

[7] 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

[8] 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/

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

[10] 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.

[11] 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

[12] 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

[13] 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

[14] 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