Client-Side XSS Filtering in Firefox

Andreas Vikne and Pål Ellingsen
Department of Computing, Mathematics and Physics
Western Norway University of Applied Sciences
Bergen, Norway
Email: andreas.svardal.vikne@stud.hvl.no, pal.ellingsen@hvl.no

Abstract—**One of the most dominant threats against Web applications is the class of script injection attacks, also called cross-site scripting. This class of attacks affects the client-side of a Web application, and is a critical vulnerability that is difficult to both detect and remediate for website owners, often leading to insufficient server-side protection, which is why the end-users need an extra layer of protection at the client-side, utilizing the defense in depth principle. In this paper, a client-side filter for Mozilla Firefox is presented, with the goal of protecting against reflected cross-site scripting attacks while maintaining high performance. By conducting tests on our implemented solution, although still in an early phase, we can conclude that our filter does provide more protection than the original Firefox browser, at the same time achieving high performance, which with further development would become an effective option for end-users of Web applications to protect themselves against reflected cross-site scripting attacks.**

Keywords–cross-site scripting; client-side filtering; Web browser protection.

I. Introduction

Cross-site scripting has for long been among the top threats against Internet security as defined in the Open Web Application Security Project (OWASP) Top 10 report, which presents the 10 most common security vulnerabilities found in Web applications [1]. Even if cross-site scripting has fallen to 7th place in the OWASP Top 10 2017 report [1], cross-site scripting remains one of the most serious attack forms. Another report being published annually for the past 12 years by WhiteHat Security, WhiteHat Security Application Security Statistics Report [2], also identifies that cross-site scripting is among the top two most critical Web vulnerabilities. An interesting observation made in this report is that even though cross-site scripting is considered one of the most critical vulnerabilities, it is not being prioritized for remediation by websites. The statistics being presented suggest that the vulnerabilities receiving most remediation are vulnerabilities that are easy to fix, which is not the case for cross-site scripting. It is suggested organizations must adopt a risk-based remediation process, to prioritize the most critical vulnerabilities first, like cross-site scripting. A report [3] published by Bugcrowd Inc., a Web-based platform that uses crowdsourced security for companies to identify vulnerabilities in their applications, analyze data from their platform, including information about the most common vulnerabilities found. The data in this report is based on all Bugcrowd data from January 2013 through March 2017, which contains of over 96 000 submissions, where the most reported vulnerability is cross-site scripting with a submission rate of 25%. They also have data on the most critical vulnerabilities by type, where cross-site scripting is considered the second most critical, which is the same result found in WhiteHat Security's report. These are some of the most recent numbers regarding cross-site scripting, but there have been published numerous of studies done on XSS vulnerabilities and attacks. One study [4] from 2014 conducted a systematic literature review were they reviewed a total of 115 studies related to cross-site-scripting. They concluded that XSS still remains a big problem for Web applications, despite all the proposed research and solutions provided so far. As seen from the recent numbers from OWASP, WhiteHat and BugCrowd, this conclusion still holds true, that XSS vulnerabilities remains to be at large. With the observation about how prevalent this type of attack is, and the fact that it is not prioritized nor easy for websites to fix and remediate it, it becomes clear that the user needs some means of protecting themselves at the client-side, since it is mainly the end-users of vulnerable Web applications that are affected by potential attacks.

A. Cross-Site Scripting Attacks

Cross-site scripting vulnerabilities are caused by insufficient validation/sanitation of user submitted data that is used and returned by the website in the response, which could compromise the user of the site. An attacker could potentially use this vulnerability to steal users' sensitive information, hijack user sessions or rewrite whole website contents displaying fake login forms. The end-users of websites are the main victims of these attacks, but the actual websites are also affected, as the attacks might negatively impact the reputation of the site, which again could lead to fewer visitors. There exist three main types of cross-site scripting attacks, which is one of the reasons why remediation for such vulnerabilities is not an easy task, as each of the different types operate differently and thus require small differences in how to properly handle and secure them. All three types rely on insecure handling of JavaScript code, and are called Reflected, Stored and Document Oject Model (DOM) Based Cross-Site Scripting (XSS) attacks [5]:

Stored XSS occurs when user input attack code is stored on a publicly accessible area of a website, typically in a comment section, message board post, visitor log or in chat rooms. When a user visits a page where such an attack is stored, the browser will retrieve the data and render it, which in turn will execute the stored XSS attack in the browser's context. This type of XSS is very difficult to protect against on the client-side, as the client have no means to identify whether the JavaScript code coming from a website is legitimate, or if it is malicious JavaScript code injected by an attacker. From the client's perspective, all JavaScript code coming from a website is legitimate and should be rendered accordingly.

Reflected XSS occurs when the user input data is sent in a request to a website, which immediately returns data in the response to the browser, without the site first making the data safe. Reflected XSS attacks are performed by entering data into search fields, creating an error message or by other means where the response use data from the request. In a reflected XSS attack, the JavaScript attack code is not stored on the website itself. For this attack to work, a user needs to visit a specially crafted URL, containing the exploit code, for the attack to be successfully done, executing the attack in the user's browser. A Reflected XSS attack thus contains a request to and response from a website, where the code inserted in the request is being used in the response. Client-side filters can, therefore, compare the contents of the request with the response, to identify a potential attack. The proposed filter in this paper utilizes this technique, which means it focuses on primarily stopping Reflected XSS attacks.

DOM Based XSS is a type of XSS attack where the malicious data that exploits a flaw never leaves the browser. This means that from an attacker inputs malicious data to a website until the code is executed in the browser, the malicious data is not part of neither the request or the response of the website, but rather part of the DOM of the Web-page. This is because DOM based attacks rely solely on flaws using JavaScript code.

B. *Counter-Measures for XSS Attacks*

Counter-measures for XSS attacks can be achieved in several ways. The first step would be to properly identify and map the attack surface of the Web application, before implementing the desired option for protection, ideally a combination of several of the following methods:

Validation/Sanitization of all untrusted data input to a Web application makes sure that malicious input is either being rejected or manipulated into being safe for usage in the output. It might be difficult to implement this properly as it can be challenging to know what a malicious input looks like, considering all the possible attack vectors that use advanced obscuration techniques.

Output encoding is the most effective remediation for cross-site scripting attacks when done properly. It is important to implement the output encoding according to the context it is being used in, because different encodings are needed depending if HTML or JavaScript code is being used.

Content Security Policy (CSP) is another common way for preventing cross-site scripting attacks, which is a declarative policy that let Web application owners create rules for what sources the client is expecting the application to load resources from. As stated in the World Wide Web Consortium (W3C) Recommendation [6], CSP is not meant as a first line of defense mechanism, but rather an element in a defense-in-depth strategy.

Disabling JavaScript is also a possibility that would totally stop XSS, since these attacks rely on a JavaScript environment for execution. This solution can be effective for simple static websites, but most dynamic websites require some sort of JavaScript support for basic functionality, which means this remediation would not be suited as an overall solution.

In the following Section II, different filter techniques are being discussed before presenting a client-side filter implementation for the Mozilla Firefox browser in Section II-A. Then, in Section III, the presented filter is analyzed, and finally, we end the article in Section IV with the conclusion and further work.

II. Client-Side XSS Filtering

When a website is vulnerable to cross-site scripting attacks, an attacker could exploit this vulnerability and possibly steal sensitive information or hijack sessions of the users accessing the exploited website. Filters try to stop these attacks by utilizing a set of rules to detect potential malicious input data, before either blocking it or sanitizing it for safe usage. There exists many XSS filter implementations, with varying focus on the different areas such as security, performance, low false-positives and usability. All of these areas are in focus of most filters, but it is not common for a filter to be best in all categories, as they do not necessarily compensate each other. There is, however, one clear way to differentiate between filters, by dividing them into two groups, server-side and client-side filters:

Server-side filters are implemented on the server side of a website, which means it can only detect input data that are sent via the server. The DOM based XSS attack, as discussed in Section I, is an attack only relying on client-side code, which means a server-side filter would not be able to detect the attack at all, which implies it would not be able to stop the attack. This is one of the reasons why only relying on server-side protection is not enough, and why we need client-side filters.

Client-side filters are located in the client, which typically would be the Internet browser used to access the website. Client-side filtering would be able to detect DOM based XSS attacks, providing the extra protection server-side filters are missing. However, even though client-side filters could possibly detect all types of XSS attacks, it should not be used alone, without server-side filtering. By placing the filter on the client-side, it means that the user might be able to modify it to circumvent the filtering. It is, therefore, strongly recommended to utilize both server- and client-side filtering, to be able to detect all attack types and achieving defense in depth protection.

Filtering techniques: There exist several implementations for cross-site scripting filters both on the client-side and server-side of Web applications, which use many different techniques, but where most also contain some limitations [7]. This paper focuses on client-side filtering, where some of the most used techniques will be discussed here. A popular technique is to use regular expressions, which has been proved to contain several flaws in its design [8]. A popular client-side XSS filter using regular expressions is NoScript [9] for Mozilla Firefox, first released in 2005 and actively updated by the maker Giorgio Maone. The filter is matching HTML code for injected JavaScript in the request by utilizing regular expression rules for simulating the HTML parser, which would potentially lead to false-positives, as it is better to over-approximate these rules than to let an attack bypass the filter [8]. Another method for client-side XSS filtering is string-matching, used by the filter in the Google Chrome browser, XSS Auditor [8]. Auditor works by matching the HTML code for injected JavaScript code for the request with the response from the website after it is been parsed by the browser's HTML parser, see [8] for more details. This means that Auditor does not need to approximate any of the HTML parser rules, since the parsing is already done when the matching algorithm starts. This is achieved by the location of Auditor, which is between the HTML parser and the JavaScript engine, which makes it possible to block scripts after parsing, by blocking them from being sent to the JavaScript engine for execution.

Regular expressions and string matching is among the techniques being implemented in the top five most used Web browsers for desktop, which according to the online measurements from StatCounter [10] are Chrome, Firefox, Internet Explorer/Edge and Safari. Both Chrome and Safari use the mentioned string matching based XSS Auditor filter. XSS Auditor was first build into the browser engine WebKit, which Safari uses, before also being integrated into a fork of WebKit called Blink, which Chrome uses. Internet Explorer and Edge both have a filter implemented based on the regular expression technique, first introduced in Internet Explorer 8 [11]. Firefox however, being the second most used Web browser, does not have a built-in filter, but rather relies solely on CSP support, which again relies on websites to properly define the CSP rules. By not having a client-side filter the defense in depth principle is also lost, where a potential filter would provide an extra layer of security for the end-users of the application. In this paper we present an implementation for a built-in client-side filter for this extra layer of security.

A. Implementation of Client-Side Filter in Firefox

The client-side XSS filter for Firefox proposed in this paper is based on the Google Chrome browser's XSS Auditor, but with some design modifications. Due to various differences in Chrome's and Firefox's internal architecture, the proposed filter in this paper is tightly coupled to Firefox and is, hence, not meant to be a copy of XSS Auditor. The basis of the filter is to first get the input data to the website, before checking if any of this data is considered dangerous, in which case a matching comparison is done for all the scripts before they are sent to the browser for execution. Both filters are doing the filtering after the HTML parser, but the proposed Firefox filter is doing the actual matching later in the rendering process than Auditor. Whereas Auditor is doing the matching before the JavaScript engine, by examining all the DOM tree nodes, the proposed Firefox filter is not doing the matching before it is actually prepared to be sent to the JavaScript engine, in Firefox's internal `ScriptLoader.cpp` class, as seen in Figure 1 below. This means that the Firefox filter is only doing matching on the scripts sent to Firefox's internal script handler, and not the whole DOM tree.

The implementation of the proposed filter is focusing on the most common way to inject and execute JavaScript on a webpage, by using the HTML script tag. The rules for filtering are based on different ways of making JavaScript code from script tags execute in the browser. OWASP's guidelines XSS Filter Evasion Cheat Sheet [12], which contains many attack vectors trying to circumvent typical XSS filtering techniques, provided a lot of examples for the creation of this paper's filtering rules.

The filter is implemented as its own class, which could then be used in parts of Firefox requiring filtering protection. This class contains several methods for detecting potential attacks, as inline scripts and external scripts needs to be processed differently. When using the filter, it start by fetching all the input data to the website in form of GET- and POST-parameters, before checking each of these parameters if they contain any potential malicious code that can be used for executing a cross-site scripting attack. In this case, the filter checks for opening HTML script tag, `<script`. If there are any occurrences of this tag in any of the input parameters, the filter will continue its examination of the input. There are now two cases in which the input data will be considered and marked as dangerous. Either if the script tag is non-empty or it contains a non-empty attribute `src`. If any of these two conditions are being fulfilled, the filter marks the input parameter as dangerous before a matching algorithm is started to try and find the input data in any of the JavaScript code sent to Firefox for execution. This is done by comparing the actual string representation of the parameter with the string representation of all JavaScript code entered through Firefox. If this matching algorithm does find a match, the whole script that contains the input data will be blocked from execution in the browser, stopping a potential attack. If no match is found, the webpage and all its contents will load and function without any intervention from the filter.

B. Mozilla Firefox Architecture

For implementing this filter into Firefox, it is important to know how the source code is built up and how the scripts are being evaluated. Mozilla Firefox source code has a layered architecture where the code is organized as separate modular components. Firefox is multi-threaded and follows the rules of object-oriented programming, where access to internal data is achieved through public interfaces of the classes [13]. One of the primary requirements of Firefox is that it must be entirely cross-platform, which is why the browser consists of several components focusing on this area, like making sure the operation system dependent logic is hidden from the application logic. The main components can be divided up into the user interface XML User Interface Language (XUL) [14] and the browser and the rendering engine Gecko [15]. XUL is Mozilla's own language for building portable user interfaces, which is an XML language. Gecko is Mozilla's browser engine built to support many different Internet standards, including HTML 5, CSS 3, DOM, XML, JavaScript and others. Gecko contains many different components for document parsing (HTML and XML), layout engine, style system (CSS), JavaScript engine called SpiderMonkey, image library, networking, security, as well as other components. The implementation of the proposed filter is located in Gecko, right before JavaScript code from a site is being sent to SpiderMonkey for processing. Both inline and external scripts from HTML script tags are being loaded into the class `ScriptLoader.cpp`, where they are passed on to the JavaScript engine for compiling and execution. Because all scripts from script tags pass through this class, this is the main area where the filter will be used. The flow of such scripts through the application is shown in Figure 1. This makes sure that all scripts are caught and can be effectively stopped from executing by simply not sending them to the

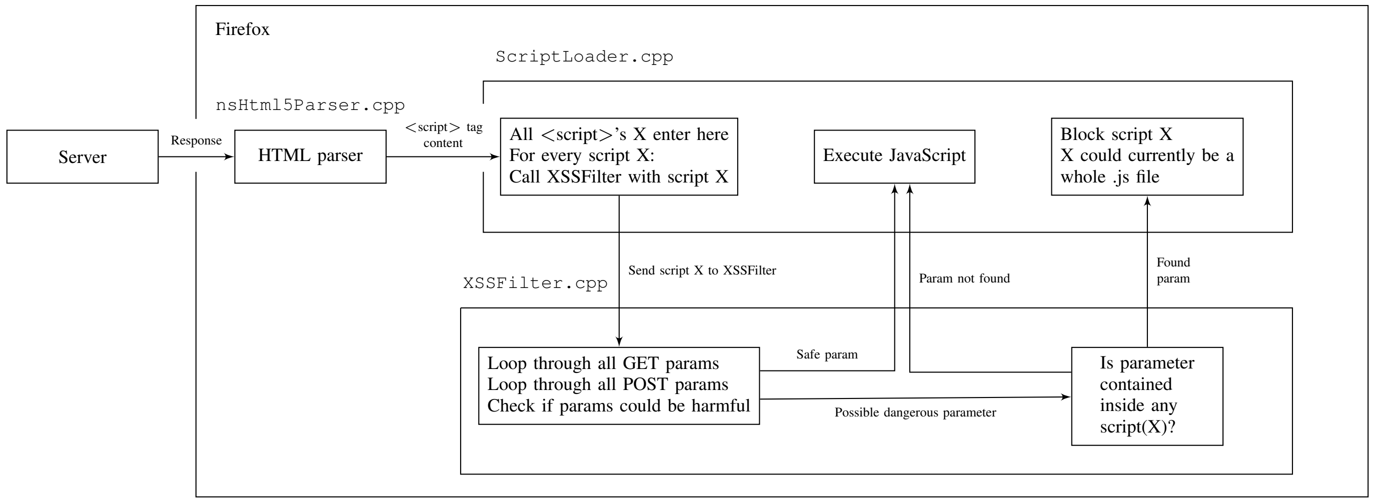
Firefox
ScriptLoader.cpp
nsHtml5Parser.cpp
Server
Response
HTML parser
<script> tag content
All <script>'s X enter here
For every script X:
Call XSSFilter with script X
Execute JavaScript
Block script X
X could currently be a whole .js file
Send script X to XSSFilter
Param not found
Found param
XSSFilter.cpp
Loop through all GET params
Loop through all POST params
Check if params could be harmful
Safe param
Possible dangerous parameter
Is parameter contained inside any script(X)?

Figure 1: Information flow in application

JavaScript engine at all. Even though all script content from script tags enter through the class `ScriptLoader.cpp`, not all input that should be interpreted as JavaScript's gets sent here. Gecko handles scripts differently based on where they originate from. HTML event handlers are being processed in another class `EventListenerManager.cpp`, before sent to the JavaScript engine. This means that for the proposed filter to work on all possible scripts from a website, it would be necessary to also use the filter in this location.

III. Analysis

A main challenge during the implementation was to properly understand the application architecture. Depending on how JavaScript code is inserted into a website, Firefox is processing the input in different modules in the application, which proved challenging to identify. The proposed filter for Firefox presented in this paper is as described in the previous section only focusing on the HTML script tag, which means all the script processing could be done in the same place in the Firefox source. However, by neglecting other means of injecting JavaScript code into a website, the filter is not capable of detecting all possible XSS attacks. Some other common HTML tags used for cross-site scripting attacks are tags like 'svg', 'object' and the usage of event handlers. It is however very possible to locate where in the Firefox source JavaScript code from other HTML tags is being processed, and to add filtering capabilities to those areas in a similar fashion done with the 'script' tag. A similar limitation is the fact that the filter only considers GET- and POST- parameters for the input. It is possible to use other input entry points like cookies, local storage, or HTTP header fields for executing cross-site scripting attacks. Neglecting support for these alternative attack vectors is also a limitation in XSS Auditor [16], but since they are valid attack vectors, they should at least be considered for improving the proposed filter in this paper.

A. Attack mitigation efficiency

When testing the implemented filter in practice, Firefox was able to successfully detect and block simple cross-site scripting attacks using the script tag for the injection point. Simple attack vectors like `<script>alert(xss)</script>` and `<script src=http://xss.rocks/xss.js></script>` both were successfully blocked by the filter when injected into a sample vulnerable website. Other more advanced attack vectors from the OWASP XSS Filter Evasion Cheat Sheet [12], like embedding spaces or tabs within the injected input, neglecting to include closing tags or substituting space with a non-alpha character were also tested, which were successfully detected and blocked by the filter. However, there is a case where the filter only was able to block parts of the injected input using only the script tags: when the input contains more than one occurrence of the script tag. An example would be the input
`<script>alert(1)</script>`
`<script>alert(2)</script>`.
In this case, the first script tag sequence containing the `alert(1)` would be blocked, at which point the filter would stop examination and hence the `alert(2)` from the second script tag would be executed, which could effectively be used to launch a successful cross-site scripting attack. This is due to the filter being limited to only detect and block the first script tag found.

As seen with the implemented filter, there is a lack of filtering rules and conditions, which makes it quite ineffective in its current form. Even with a case of only using the script tag, the filter was unable to detect all injected attacks. Not to mention all the other ways attack vectors using different HTML tags an attacker could use. By studying the OWASP's XSS Filter Evasion Cheat Sheet, where a lot of these different attack vectors are shown, with the purpose of evading common filters, the cheat sheet is effectively showing that for every attack vector, it is possible to properly detect and block the attack by using the correct rules and conditions. This also applies to the proposed filter in this paper, that it is possible to implement all these rules and their variations to be able to filter away most cross-site scripting attacks.

Another property of the implemented filter is how it handles a detected injected script, and how that affects its functionality. When the filter detects that a script from the input is found in any script loaded into the browser engine for processing, the whole script loaded for processing is being stopped before it is executed. The rest of the scripts on the website would still be loaded and executed as usual. There is however also another approach that is common for XSS filters, which is to block loading the entire website where a potential XSS attack was discovered. There are advantages and disadvantages to each of these methods. An advantage to only block specific parts of a website is that the user is still able to browse and view the other parts of the website not affected by the injection, making it a better user experience with less disruptions in case of an attack. In cases of false-positives, where there are no real attacks, this technique is more forgiving by not blocking all the website's content. A disadvantage of only blocking parts of the webpage is that in the case of a detected attack, it is not unlikely that an attacker would probably try to use different advanced attack vectors, which could trick the filter like the case described above, using double script tags. Therefore in regards of security, it is best to block the whole webpage when a potential injection attack is detected. By utilizing blocking of the whole website instead of only the parts where the script was detected would effectively allow the implemented filter described to successfully block the attack with double script tags, making the filter much more secure by just this single modification to its design. There are however negative effects by blocking the whole webpage, which is the user experience would greatly be affected by a lot of discovered attacks, which would disrupt the user from normal browsing activities and where the user ultimately maybe choose to disable the filter altogether. This is especially the case with false-positives, where there actually is no attack, but the filter still blocks the entire page from loading.

There is no simple answer to which of these techniques to use, but there are ways that websites themselves can choose what to do. By setting the HTTP header X-XSS-Protection, webpages could choose to either allow, sanitize or block detected cross-site scripting attacks [17]. This header is currently supported by other major Internet browser, but not Firefox, as Firefox does not supply built-in XSS filtering. By adding support for this header in Firefox and the implementation of the proposed filter in this paper, it would be possible also for Firefox to let the webpages themselves choose how to deal with detected cross-site scripting attacks, either allowing everything, only blocking assumed affected content, or block the whole webpage from loading.

An additional limitation of the implemented filter is the support for different input encodings. When receiving input into a webpage, the input might be encoded with different encodings, like hex encoding, which is not currently supported by the filter. This is however easily fixed by first adding a check for what encoding is used, if any, before properly decoding the input. This is a very important feature that needs to be taken into consideration, as using different character encodings is a common way to obscure cross-site scripting attacks.

B. Performance

The performance of the implemented filter is an important factor for its usefulness. We followed Mozilla's own methodology for comparing page load times across browsers [18], using popular websites to load in the browser, repeated several times, while measuring the loading time for each page. We chose 10 of the most popular news websites from Alexa [19], knowing that news sites typically contain a lot of scripts for ads and tracking. To make sure the modified browser actually ran the code for our implemented filter, we used the search function on each of the websites and tested with two different parameters, one safe and one unsafe, which would activate the filtering. We also wanted to conduct a performance test for actual vulnerable Web applications. From a website containing a list of Web applications vulnerable to XSS attacks [20], even though it was an old archive, we collected four different websites all vulnerable to XSS attacks, and then injecting them with the simple script `<script>console.log(1)</script>`. This simple script injection was chosen because it makes it easy to compare the load time between the modified Firefox and original Firefox, as this simple script would not alter the rendering of the page itself, but still be a valid cross-site scripting attack. As we also injected the 10 chosen news site with a script input, we did not expect any big different in performance between these sites and the acutual vulnerable sites, as the filter would run the same matching algorithm on all sites. As expected, even though the filter from this paper successfully detected and blocked this injection on all these vulnerable websites, there were no overhead compared to the news sites. For the full performance test, a total of 1040 page loads were performed for each browser, including both the 10 news sites and the four vulnerable sites. This resulted in an average difference of only 32.1 ms for each page load, which equals a performance overhead for the modified browser of about 0.7 % compared to the average loading times for the original Firefox browser, which is an insignificant overhead. As there are several limitations with the current implementation of the filter, a more complete version addressing these limitations would probably incur a higher overhead, but at a starting point at 0.7 % it is reason to believe the added overhead would not be of any significance. This was however a test with several limitations, as there might have been too few total page loads for each browser, the visitor traffic to the tested websites might be different and there might have been small interferences in the Internet connection when performing the test. Although there these factors might have affected the results, it is worth noting the the two browsers were tested in the same time span, which should not incur too much variation. After taking the average of the 1040 page loads for each browser, the achieved results do highly indicate that the modified browser do not incur any significant performance overhead.

IV. Conclusion and Future Work

Information flow vulnerabilities can occur when applications handle untrusted data. When this happens, users of the application might be negatively affected, without any means of protecting themselves. By utilizing client-side filtering, like proposed in this paper, the user do have a means to protect themselves from malicious attackers. By default, the Firefox browser have no such protection mechanism built in, which this paper has a proposal for adding. As seen in the analysis in Section III, there are still many important additions to be done before the filter is ready for everyday usage, but the filter do work for basic cases, which already provides more protection than the default Firefox browser, proving a solution efficient enough to work, achieving high performance with almost no overhead. When the rest of the presented additions is implemented, this filter would work as an important extra protection for the end-users of vulnerable Web applications, efficiently protecting against reflected cross-site scripting attacks.

A reasonable next step would be to further expand the filtering capabilities of the filter. This would be achieved by implementing the proposed improvements from Section III, covering all attack vectors from all possible injection points, adding more rules and conditions for the filtering, have proper decoding of input and adding support for the X-XSS-Protection header. After making these improvements, it is also necessary to do further testing, for both loading speeds and focus on security, with specially crafted attack vectors, and to make sure the filter is as robust and secure as desired, making it an effective way for protecting the end-users of websites from cross-site scripting attacks on the client-side.

REFERENCES

[1] OWASP Foundation, "Owasp top 10 - 2017 the ten most critical web application security risks," accessed: 2017-12-27. [Online]. Available: https://www.owasp.org/images/7/72/OWASP_Top_10-2017_(en).pdf.pdf

[2] WhiteHat Security, Inc., "2017 whitehat security application security statistics report," 2017, accessed: 2017-12-21. [Online]. Available: https://info.whitehatsec.com/rs/675-YBI-674/images/WHS 2017 Application Security Report FINAL.pdf

[3] Bugcrowd Inc., "2017 state of bug bounty report," 2017, accessed: 2018-01-09. [Online]. Available: https://pages.bugcrowd.com/hubfs/Bugcrowd-2017-State-of-Bug-Bounty-Report.pdf

[4] Hydara, Isatou and Sultan, Abu Bakar Md and Zulzalil, Hazura and Admodisastro, Novia, "Current state of research on cross-site scripting (XSS)–A systematic literature review," Information and Software Technology, vol. 58, 2015, pp. 170–186.

[5] OWASP Foundation, "Types of cross-site scripting," March 2017, accessed: 2018-03-05. [Online]. Available: https://www.owasp.org/index.php/Types_of_Cross-Site_Scripting

[6] The World Wide Web Consortium, W3C, "Content security policy level 2," December 2016, accessed: 2018-01-11. [Online]. Available: https://www.w3.org/TR/2016/REC-CSP2-20161215/

[7] S. Gupta and B. B. Gupta, "Cross-Site Scripting (XSS) attacks and defense mechanisms: classification and state-of-the-art," International Journal of System Assurance Engineering and Management, vol. 8, no. 1, 2017, pp. 512–530.

[8] D. Bates, A. Barth, and C. Jackson, "Regular expressions considered harmful in client-side xss filters," in Proceedings of the 19th international conference on World wide web. ACM, 2010, pp. 91–100.

[9] G. Maone, "NoScript - JavaScript/Java/Flash blocker for a safer Firefox experience! - features - InformAction," accessed: 2017-12-28. [Online]. Available: https://noscript.net/features

[10] StatCounter, "Desktop browser market share worldwide dec 2016 - dec 2017," December 2017, accessed: 2018-01-11. [Online]. Available: http://gs.statcounter.com/browser-market-share/desktop/worldwide

[11] D. Ross, "Ie8 security part iv: The xss filter," July 2008, accessed: 2018-01-11. [Online]. Available: https://blogs.msdn.microsoft.com/ie/2008/07/02/ie8-security-part-iv-the-xss-filter/

[12] OWASP Foundation, "Xss filter evasion cheat sheet," October 2017, accessed: 2017-12-27. [Online]. Available: https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet

[13] Mozilla Developer Network, "An introduction to hacking mozilla," Mars 2017, accessed: 2017-12-28. [Online]. Available: https://developer.mozilla.org/en-US/docs/Mozilla/An_introduction_to_hacking_Mozilla

[14] ——, "Introduction," September 2014, accessed: 2017-12-28. [Online]. Available: https://developer.mozilla.org/en-US/docs/Mozilla/Tech/XUL/Tutorial/Introduction

[15] ——, "Gecko faq," September 2015, accessed: 2017-12-28. [Online]. Available: https://developer.mozilla.org/en-US/docs/Gecko/FAQ

[16] Stock, Ben and Lekies, Sebastian and Mueller, Tobias and Spiegel, Patrick and Johns, Martin, "Precise Client-side Protection against DOM-based Cross-Site Scripting." in USENIX Security Symposium, 2014, pp. 655–670.

[17] Mozilla Developer Network, "X-xss-protection," October 2017, accessed: 2017-12-28. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-XSS-Protection

[18] D. Strohmeier, P. Dolanjski, "Comparing browser page load time: An introduction to methodology," November 2017, accessed: 2018-01-15. [Online]. Available: https://hacks.mozilla.org/2017/11/comparing-browser-page-load-time-an-introduction-to-methodology/

[19] Alexa Internet, Inc., "The top 500 sites on the web," January 2018, accessed: 2018-01-15. [Online]. Available: https://www.alexa.com/topsites

[20] "Xss archive," accessed: 2018-03-05. [Online]. Available: http://www.xssed.com/archive