Cross-Site Scripting Protection in Firefox

Creating a filter for Firefox protecting against Reflected cross-site scripting attacks

Andreas Svardal Vikne

Master's thesis in Software Engineering at

Department of Computing, Mathematics and Physics,
Western Norway University of Applied Sciences

Department of Informatics,
University of Bergen

June 2018

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 websites, 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 strategy. In this thesis, a client-side filter for Mozilla Firefox is presented and implemented, with the goal of protecting against Reflected cross-site scripting attacks, while maintaining high performance. By conducting tests on the implemented solution, the conclusion is that the filter does provide more protection than the original Firefox version, at the same time achieving high performance, which with only some further improvements would become an effective option for end-users of web applications to protect themselves against Reflected cross-site scripting attacks.

Acknowledgements

I would like to thank my thesis advisor, Associate Professor Pål Ellingsen, for his invaluable comments and guidance he has given me during the writing of my thesis. His immerse positivity and support has also been highly appreciated.

Contents

Acronyms

ASCII American Standard Code for Information Interchange. 39

CSP Content Security Policy. v, 16, 26, 27, 29, 34, 40, 54, 55

CSS Cascading Style Sheets. 24, 32

DOM Document Object Model. iv, 3, 8, 10, 24

GDB GNU Project Debugger. vi, 36

HTML HyperText Markup Language. 11, 24, 26, 27

HTTP Hypertext Transfer Protocol. 8, 13, 48, 53, 57

IDE Integrated Development Environment. 36

OWASP Open Web Application Security Project. 1, 14

SQL Structured Query Language. 7

SQLi Structured Query Language injection. 6, 7

URL Uniform Resource Locator. 8, 9, 11, 13–15, 21, 38, 39, 41, 45, 46, 54, 55

VS Code Visual Studio Code. 36

W3C World Wide Web Consortium. 17

XML Extensible Markup Language. 24

XSS Cross-Site Scripting. iv, 1, 3, 4, 6–8, 11–15, 17–23, 27, 31, 33, 37, 43, 48, 49, 54–56

XUL XML User Interface Language. 24

List of Figures

List of Tables

1 Introduction

1.1 Background

Cross-Site Scripting (XSS) has for long been among the top threats against Internet security as defined in numerous reports containing detailed information about the prevalence and danger regarding this class of vulnerability. One of these reports is the "Open Web Application Security Project (OWASP) Top 10 - 2017" report, which contains a list of the 10 most critical web application security risks [7]. Even though cross-site scripting has fallen to a 7th place in the "OWASP Top 10 - 2017" report [7], cross-site scripting still remains one of the most serious attack forms. Another report, being published annually for the past 12 years, by WhiteHat Security, called "2017 - WhiteHat Security Application Security Statistics Report" [8], also identifies that cross-site scripting is among the top two most critical web vulnerabilities. An interesting and troubling 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 most websites. The statistics being presented suggest that the vulnerabilities receiving most attention are vulnerabilities that are easy to fix, which is not the case for cross-site scripting. As a result of this, it is suggested that organizations must adopt a risk-based remediation process, which means that the most critical vulnerabilities should be prioritized first, like cross-site scripting. A report [9] published by Bugcrowd Inc., a web-based platform that use crowd-sourced security for companies to identify vulnerabilities in their web applications, has analyzed the data from their platform, including information about the most common vulnerabilities found. The data in their report is based on all BugCrowd's collected data from January 2013 through March 2017, which contains of over 96 000 submissions, where the by far most reported vulnerability is cross-site scripting with a submission rate of 25%. They also have data on the most critical vulnerabilities sorted by type, where cross-site scripting is considered the second most critical, which correspond to the same result found in WhiteHat Security's report. These are some of the most recent numbers regarding cross-site scripting statistics, but there have been published numerous studies on XSS vulnerabilities, attacks and its prevalence. One study from Hydara et al. [10] 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 being provided so far. As seen from the more recent numbers from OWASP, WhiteHat and BugCrowd, this conclusion still holds true, that XSS vulnerabilities remains to be at large.

1.2 Problem Description

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. With the observation about how prevalent this type of attack is, and according to the mentioned WhiteSecurity report that it is being not prioritized nor easy for websites to fix and remediate, 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. Amongst the top 5 most used web browsers [11], Mozilla Firefox is the only browser which does not include any kind of built-in filtering against cross-site scripting attacks, which may compromise users in the case of a vulnerable web application.

1.3 Related Work

This thesis is based upon existing research regarding the state of current client-side filtering for cross-site scripting attacks, containing weaknesses and suggestions for further improvements. Based on these existing results, a filtering solution for Firefox was first proposed in a paper by the author of this thesis, "Client-Side XSS Filtering in Firefox" [12]. This thesis builds on the same work and expands on the results given there.

1.3.1 Regular Expressions Considered Harmful in Client-Side XSS Filters [1]

This paper analyses existing client-side filters and techniques for cross-site scripting, before presenting a better solution based on a different design. The filters analyzed uses regular expressions for their filtering, which the paper concludes being either unacceptably slow or easily circumvented. They then present a new filter design which achieve high performance and high precision, by implementing the filter to blocking scripts after the HTML parser but before script execution, and by using string matching instead of regular expressions. Their filter, XSS Auditor, was first implemented in the WebKit rendering engine, and is now enabled by default in the Google Chrome browser.

1.3.2 Precise Client-side Protection against DOM-based Cross-Site Scripting [2]

This paper also discusses flaws with existing client-side filters, including the already introduced XSS Auditor. The paper focuses most on flaws related to DOM Based XSS vulnerabilities, which is a subtype of XSS, which is not the primary focus of XSS Auditor, but yet relevant as XSS Auditor is being considered as state-of-the-art in client-side Cross-Site Scripting (XSS) mitigation. After elaborating several limitations in existing filters, the paper presents an alternative filter design mainly focusing on DOM Based XSS, by looking at attacker-controlled syntactic content, utilizing runtime taint tracking and taint-aware parsers to stop attacks.

1.3.3 Cross-Site Scripting (XSS) Attacks and Defense Mechanisms: Classification and State-of-the-Art [3]

This paper contains statistics and analysis of the current situation concerning vulnerable websites with the main focus on cross-site scripting attacks. The paper contains a thorough discussing regarding cross-site scripting theory, from basic concepts, discussion of different types and their outcome, to describing how to both perform an attack and to identify if a website is vulnerable to XSS attacks. Then the paper analyses several existing filters for protecting against XSS attacks, filters both placed on the client-side and server-side of web applications, finding different limitations contained in all of them. The paper ends with proposing several additions and guidelines for creating better XSS defensive techniques and filters.

1.3.4 Related Work Conclusion

All the presented papers chosen for related work contains analyses for different existing solutions for filters protecting against cross-site scripting attacks. All of them go into specific details regarding the flaws of different filters, which are then used to create some requirements that needs to be addressed when creating a new filter. Although the three chosen papers does analyze some of the same filters, they all provide some unique evaluation of the findings. The three papers also focuses on different areas concerning cross-site scripting filters, either a specific type of XSS attack or a more general analysis regarding XSS filters and their current state. Reading and collecting all the proposals from the different papers, comparing them and putting them together, creates a solid understanding of the different limitations and differences between the current state-of-the-art filters. This is knowledge that is important when proposing a new solution based on many of these ideas presented.

1.4 Thesis Outline

Chapter 2 will go into detail about web security and more specifically about XSS, explaining everything from what it is to different ways of protecting against it, focusing mainly at the client-side of web applications. This section will also include information about the current state regarding XSS prevalence and existing work, before ending with a detailed description of the methodology used in this thesis work. Following, in Chapter 3, will be describing the web browser, Mozilla Firefox, which is the application that this thesis work are building upon. Chapter 4 will then describe all the design choices and the actual implementation of the work done, before Chapter 5 will contain an analysis of how well the work is done, in terms of protection effectiveness, performance and integration into Firefox. The chapters, Chapter 6 and Chapter 7, contains a conclusion based on all the work done, before ending with some suggestions for further improvement.

2 Theoretical Background

2.1 Web Security

Web applications need to be protected against malicious users who want to steal and tamper their data. Web security is a broad concept, including many different aspects, protection mechanisms and potential outcomes. To be able to protect a web application, basic understanding of information security is therefore needed, as it regards some basic principles and objectives for why security is important and how to utilize it correctly.

2.1.1 Information Security Principals

Information security defines three important objectives of security [13], which are maintaining confidentiality, integrity, and availability. Confidentiality is about protection of information and data from being accessed by unauthorized parties. When someone gets access to data that they should not have access to, like sensitive information about users, it is considered a breach of confidentiality. Integrity is about the authenticity of information, ensuring that it is not altered and to make sure the source of the information is genuine. In web applications, this could be if an attacker is redirecting you to a different site than you originally intended to visit, as the site you get redirected to is not genuine. And lastly, availability regards that information should be accessible for the authorized users, which of course should be done in such a way that there is no breach in confidentiality or that someone might alter the available data when accessing it. All these objectives of security are important when creating secure web applications. To be able to fulfill them all, web applications need to protect against several different attacks from malicious parties trying to steal their and their users data. This is not an easy task, as there exists so many different types of attacks for targeting all kinds of vulnerabilities that are often contained in web applications.

2.1.2 Web Security Statistics

Several companies and organizations are doing annual research and assessment work containing a lot of collected data from a huge number of web applications and reports regarding security breaches and vulnerabilities. One

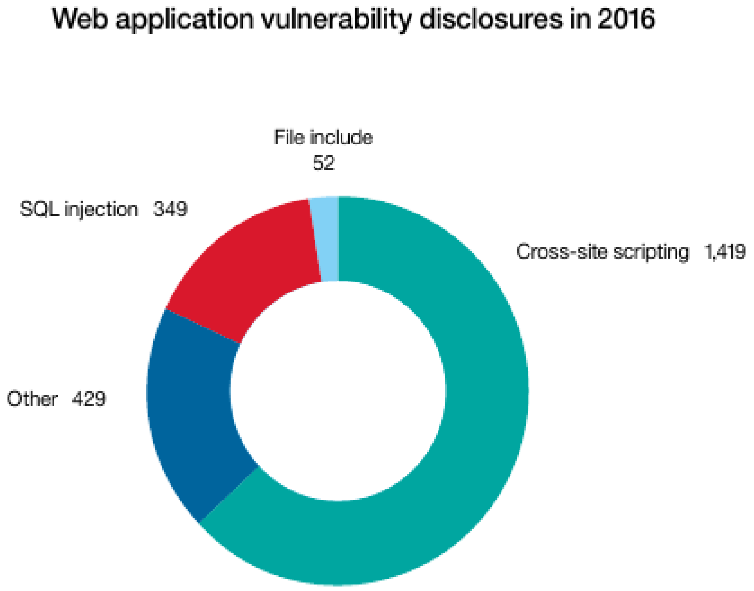


Figure 1: Web application vulnerability disclosures in 2016.
Figure taken from "IBM X-Force Threat Intelligence Index 2017" [4]

of these reports [8], already mentioned in the introduction, from WhiteHat Security, goes into depth describing the current web security state. This report does not only contain information about XSS attacks, but a whole range of other web vulnerabilities with information of how prevalent they are, as well as which industries and areas that are the most vulnerable to different attacks. WhiteHat's report also contains a list of a vast number of web application vulnerability classes, describing 64 different web vulnerabilities that needs to be protected against for web applications. This is a huge number of vulnerabilities, and while not all are relevant for every web application, many of them are critical, which needs to be addressed accordingly, where the injection attacks XSS and Structured Query Language injection (SQLi) is considered the most critical. Another report, by IBM, "IBM X-Force Threat Intelligence Index 2017" [4], is another comprehensive report containing statistics from different security events including web security, identifying what vulnerabilities are used and targeted industries. IBM also concludes that XSS and SQLi vulnerabilities are the most critical and prevalent, as seen in Figure 1, which need more attention by the different industries. As a whole, containing all web vulnerabilities, both reports have identified a small decrease in vulnerabilities in web applications, but also that attackers are targeting the most critical vulnerabilities more, in which one of the most critical, XSS vulnerability, is the least prioritized by applications to fix. Another concerning factor identified by both reports is that it takes too long to fix web vulnerabilities, which means both the application itself as well as the end-users are at a higher risk of being affected by a security breach.

2.1.3 Injection Attacks

The reports from WhiteHat Security and IBM, as discussed above, makes it clear that the most prevalent attack on web applications is injection attacks, which includes attackers trying to break the confidentiality by stealing data from the web application itself or from the users of the web application. Injection attacks are performed with attackers inputting untrusted input to web applications that is executed as a command or query in such a way that it alters the course of execution, which could result in stealing of sensitive information or altering of data. There exists several types of injection attacks, but the most prevalent is by far SQLi and XSS. SQLi involves unauthorized users to inject Structured Query Language (SQL) commands that can read or modify data from a database connected to the web application. This is achieved through the usage of user-supplied input that gets used as part of a SQL query without the web application validating or encoding the input correctly. As attackers can read and modify data upon a successfully executed SQLi attack, it is possible to steal sensitive user data such as usernames and passwords, alter the contents of the stored information or simply delete everything contained in a database, which would incur huge complications for the affected web applications. The other critical vulnerability, XSS, will be covered in more depth in the following section.

2.2 Cross-Site Scripting (XSS)

Cross-site scripting vulnerabilities are caused by insufficient validation/sanitation of user submitted data in form of JavaScript code, that is used and returned by the website in the response without making sure the content is safe to use, which could compromise the users of the site. An attacker could potentially use this vulnerability to rewrite the contents on the website creating fake login forms to steal users' sensitive information, hijack user sessions or redirect them to other malicious websites.

There are three main types of cross-site scripting attacks, but there also exists some other defined types:

- Stored XSS, also called Persistent XSS
- Reflected XSS, also called Non-Persistent XSS
- DOM Based XSS
- Others - Plug-in XSS, Universal XSS, Self XSS

2.2.1 Stored/Persistent XSS

Stored XSS occurs when the injected script is stored on a publicly accessible area of a website, which means on the actual website itself. Typical places susceptible to Stored XSS attacks are in comment sections, message board posts or in chat rooms. Since the input data is stored in these places, if the input data is an injected script, the injected script might get executed upon loading of the page, if the page is vulnerable. When a user visits one of these places, the browser will retrieve the data and render it, which in turn will execute the Stored XSS attack in the browser's context. Figure 2 illustrates the flow of a typical Stored XSS attack. Other places susceptible to Stored XSS attacks might include areas of a website only accessible to administrators, like a visitor log or other logs containing information about the usage of the website from users, as it is possible to inject JavaScript code into Hypertext Transfer Protocol (HTTP) headers [14] like the `Referer` [15] or `User-Agent` [16] headers. As the data from these headers are not unlikely to show up in some kinds of logs, a successful XSS attack here would be performed in the context of an administrator's browser, where it might be possible to not only get access to sensitive information from a single victim, but rather data from the whole web application. This type of XSS is very difficult to protect against on the client-side, as the client has 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. A user does not need to visit any specific Uniform Resource Locator (URL) or include

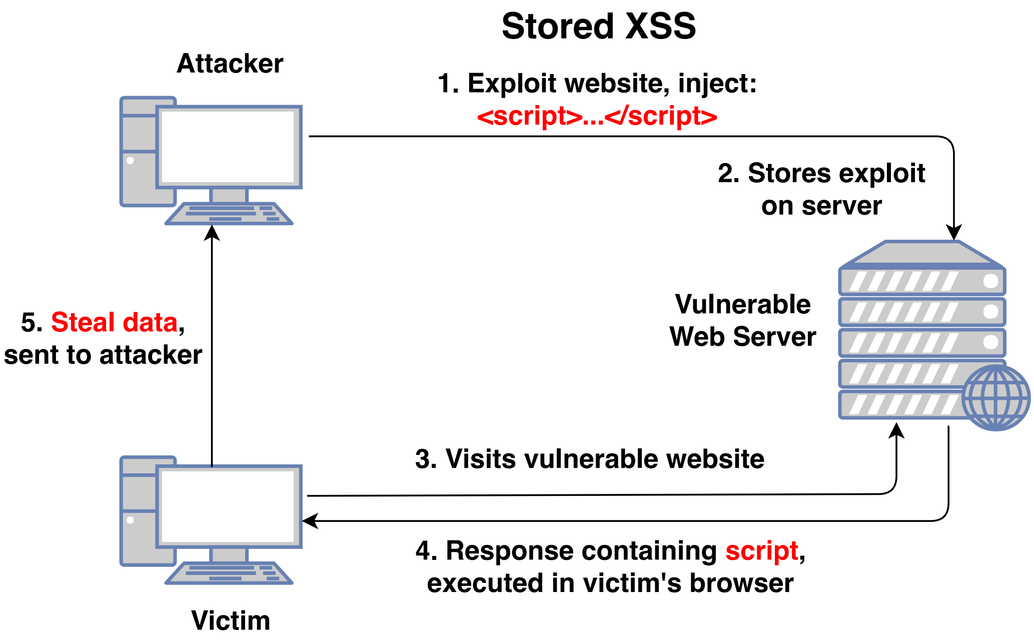


Figure 2: Stored/Persistent XSS

anything in the request to a website for a Stored XSS attack to be executed. From the client's perspective, all JavaScript code coming from a website is legitimate and should be rendered accordingly.

2.2.2 Reflected/Non-Persistent XSS

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 website first making sure the data is 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, like it is in a Stored XSS attack. For a Reflected XSS attack to work, the attacker needs to somehow make the victim request a special query, containing the malicious script. As mentioned, the search field is a typical input field that can be attacked. When searching for a query, the website often returns a page containing some results, which also will generate an unique URL containing the submitted query. This is how an attacker would create a specially crafted URL containing the exploit code, which then needs to be

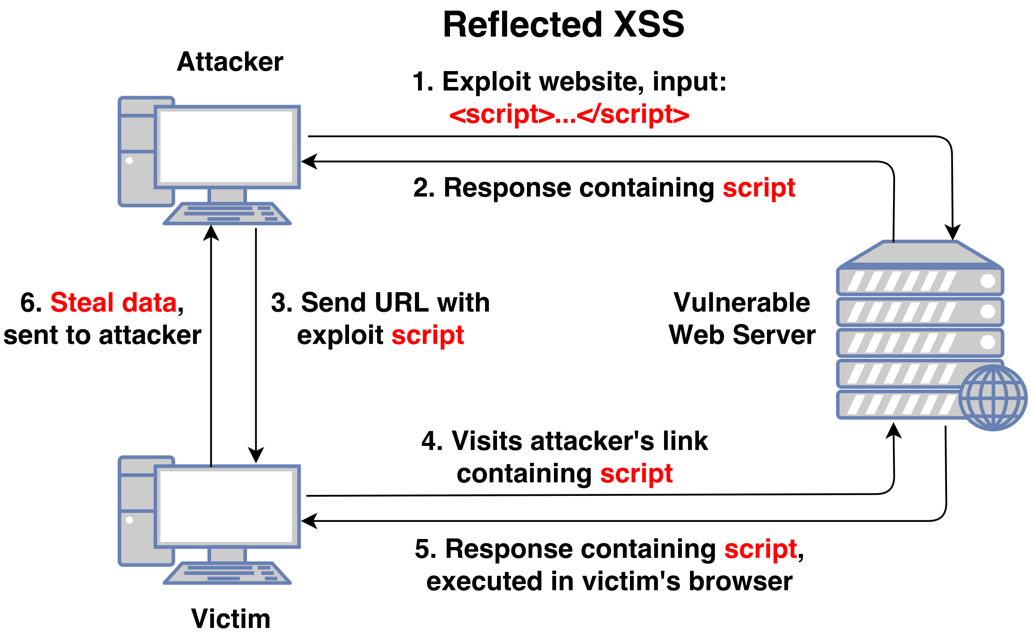

Reflected XSS
Attacker
1. Exploit website, input: <script>...</script>
2. Response containing script
6. Steal data, sent to attacker
3. Send URL with exploit script
Vulnerable Web Server
4. Visits attacker's link containing script
5. Response containing script, executed in victim's browser
Victim

Figure 3: Reflected/Non-Persistent XSS

shared with a victim. If a user visits this particular URL, the attack code will run and execute in the user's browser. Figure 3 illustrates the flow of a typical Reflected XSS attack. As seen from this figure, a Reflected XSS attack contains a request to and response from a website, where the code inserted in the request is being used in the response. It is this particular data flow that protection mechanisms can take advantage of, where it is possible to compare the contents of the request with the contents of the response, to identify a potential attack. In this thesis, this technique is utilized, which means it focuses on primarily stopping Reflected XSS attacks.

2.2.3 DOM Based XSS

DOM Based XSS is a type of XSS attack that in contrast to the other two types of XSS attacks only rely on JavaScript vulnerabilities on the client-side of the website, and not the server-side. DOM Based XSS attacks exploits how a website uses JavaScript to dynamically change the DOM of a web page. The DOM of a web page is the structure of the page, containing information for the browser on how to render the page, with the usage of different HTML tags and attributes. The DOM of a page makes it possible for JavaScript

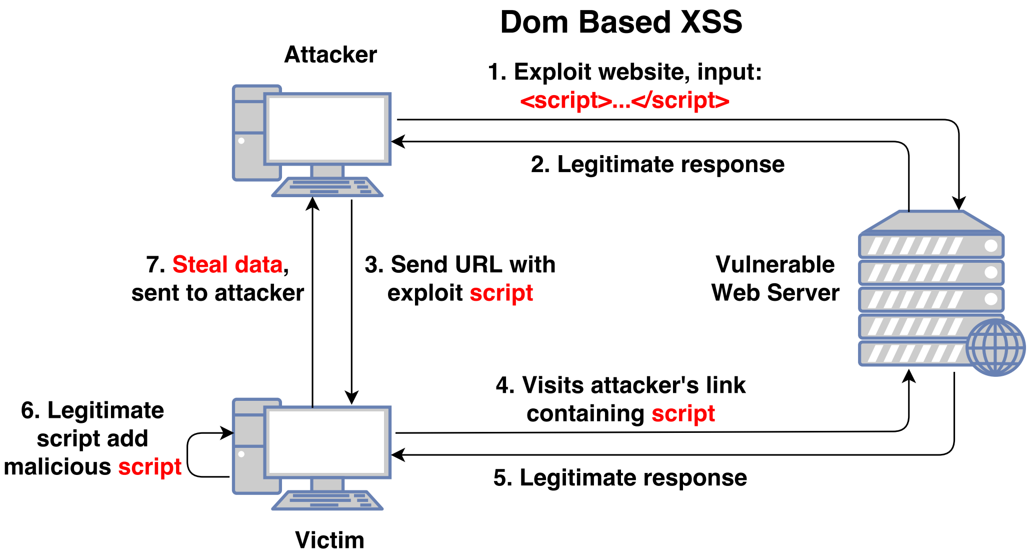


Figure 4: DOM Based XSS

code to interact with the page, making the page more dynamic. This also makes it possible for malicious code to change the page, if JavaScript input is not handled correctly. If a website includes some JavaScript code in the response that directly uses input from an input source, like the URL, a DOM Based XSS might be executed. Figure 4 illustrates the data flow of a typical DOM Based XSS attack. These attacks can actually be performed without even sending the attack script to the web server at all, by using a special HyperText Markup Language (HTML) character, the fragment identifier #, in the URL. When using the fragment identifier, everything behind it will not be part of the request. This means that from the user inputs some data, to the malicious code is executed in the browser, the malicious code is neither part of the request nor the response of the website, but rather part of the DOM of the web page, if the content after the fragment identifier is used by client-side code in the response. DOM Based XSS is the least common type of XSS attacks, but it is also the most difficult to find and protect against. Since the attack only relies on flaws on the client side, by using JavaScript code, server-side filtering can not detect this attack at all, which is a good reason why it is necessary to have protection also on the client-side of a web application.

2.2.4 Other XSS Types

Although there exits three main types of cross-site scripting, as these attacks have evolved and been used in different ways, XSS types could now be categorized into some additional sub-categories, Universal XSS, Plug-in XSS and Self XSS.

Universal XSS Universal XSS [17] is a form of XSS attack that exploits the browser itself, browser extensions or website extensions in order to exploit a website. Universal XSS is a very dangerous type of XSS as it does not exploit the website directly, meaning that a website does not need to contain any vulnerabilities to be exploited. Modern web browsers support extending their functionality by utilizing plug-ins, small programs that adds more features to the browsers. There also exists plug-ins that are not loaded through the browser, but by the website itself. These plug-ins often have access to the contents of the websites, and often require input from the user for its functionality to work. By having user input in combination with features for displaying or editing contents on a web-page, the plug-in might create an opening for allowing a cross-site scripting attack against the web-page it is being used on. An example could be a plug-in that allows websites to display pdf-files. If an attacker injects some JavaScript code in the filename of the displayed pdf-file, this JavaScript code could be rendered in the browser, if the plug-in does not have proper validation and encoding for the input field used for the filename. XSS vulnerabilities introduced by insecure plug-ins are often categorized as Plug-in XSS, which could be considered as a sub-type of Universal XSS.

Self XSS Self XSS is when users themselves create and execute the attack in their own browsers, which can not exploit other users, as in the case with the three main types of XSS. Self XSS is mostly a social-engineering attack used to trick users into executing XSS attacks on themselves, often by making them copy and paste JavaScript code into their own browsers. Awareness around this particular attack was gained through the popular social media website `Facebook.com`, as this attack became quite widespread against the users of their site, which led to Facebook publishing a warning [18] against Self XSS scams. Facebook even created a warning displaying when a user opens the developer console window in their browser while visiting their site `facebook.com`, to mitigate the attack.

2.2.5 Cross-Site Scripting Exploits

As described in Chapter 2.2, XSS attacks occur because web applications are using unsanitized input data when displaying and rendering content. For a successful XSS injection, from the attacker's perspective, the input containing the malicious JavaScript content needs be entered into the web application in a way that its somehow gets executed in the browser. The next sections will explain how this is done, and give some examples of how typical XSS attacks are performed.

Injection input sources When performing an XSS attack, it is possible to inject the malicious script into the web application by using several different input sources. An input source is considered an entry point for user input to enter into the application. The most common input sources for XSS attacks are from the GET- and POST- parameters, which most often comes from HTML input elements. A typical example is the search field found on many websites, which most often is a HTML `input` tag. After using the search field, the search query is likely to be included in the URL of the returned web page, which would consist of a GET parameter containing the query. HTTP headers is another input source for script injections, as discussed in Chapter 2.2.1. Injecting script content through HTTP cookies, which is a small piece of data sent to the user's web browser from a server, is also an option, although this is much less common, as a potential attacker would most likely need to get access to other users' cookies for injecting their script. Since the end goal of a XSS attack often includes getting access to such cookies, using them as an input source for an attack seems less likely, although in theory it is still a possibility.

Attack vectors For a successful XSS attack, the injected script content needs to be entered into the web application in a way that would actually render the script in the browser. This could be done by using a wide variety of attack vectors, depending on how the web application uses the input when generating the response. Attack vectors are typically a combination of HTML tags that include the script to be injected and executed. These tags could either embed the script content directly or reference an external resource containing the JavaScript code. The most common attack vector is the usage of the `script` tag. Another very common attack vector is the usage of the `img` tag in combination with `on-event` handlers [19]. The `on-event` handlers are properties that lets HTML elements react to events, where events are different actions like when an element is being clicked, getting focus, or when it is loaded. The reaction to an event can be specified to load script content, which is why they are often used in XSS attacks. OWASP's "XSS Filter Evasion Cheat Sheet" [20] is a comprehensive list of attack vectors utilizing a lot of different techniques, including many uses of `on-event` handlers. Other than the most common `script` and `img` tag, the `iframe`-, `body`-, `svg`-, `object`- and `style`- tag are also HTML tags not uncommonly used in XSS attacks. OWASP's list [20] contains descriptions of these and many more, including techniques to hide the injected script from being detected by potential XSS filters.

Example attack A typical scenario for a XSS attack starts with an attacker looking for input fields on a web page where the submitted data is output without being encoded. As mentioned above, the search field is a common input source. An attacker could therefore exploit a vulnerable search field, with the intention of trying to hijack another user's session. The search field is often exposed for an attack, as when you input a query, the same query is most likely being returned and rendered by the website. If this input is not properly being encoded, it could allow the attacker to input JavaScript code that is being executed in the browser's context when the website returns the query, which could be achieved using the `script` tag as the attack vector. For hijacking a user's session, the attacker would need some JavaScript code that extracts the user's session data, typically found in a cookie from the logged in targeted user. The exploit code, `<script>document.location='http://attacker/cookieStealer.js?c=document.cookie</script>`, could then be inserted into the search field. After creating this exploit, the attacker would need to copy the URL from the result page after doing the search. Since this is a Reflected XSS attack, the attacker would then need to share this URL to potential users of this exploited site. If a targeted logged in user now visits this particular URL, the user's session cookie is being sent to the attacker. The attacker could then use this cookie to log in onto the exploited website, which means the attacker would be impersonating the user.

Another popular XSS attack is to rewrite the contents of a website, creating fake forms for tricking user's to enter sensitive data like credit-card information or login details. The attacker would then make these forms submit the sensitive data to themselves, rather than to the exploited website.

A typical thing that XSS attacks have in common is that they are often not easy to detect by the end-users themselves. In case of both the cookie stealing and fake forms exploits, the attacker could simulate the actual behavior of the exploited website, making it almost impossible for users to detect that they have been compromised. By having a client-side filter in the browser, user could not only be notified of a potential attack, but the filter could also completely stop it from occurring in the first place, which is the intent of the filter.

2.2.6 Counter-Measures

There exists many counter-measures for XSS attacks, consisting of several techniques as well as more specific policies to follow, for securing web applications. It is highly recommended to utilize a variety of many different counter-measures, as it might be challenging to implement them being completely robust and secure from unknown attacks and not all policies is fully supported by all web browsers.

Validation/Sanitization The first step towards protecting against XSS attacks is to make sure that valid malicious code does not enter the web application at all. 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 response from the website, used in the output in users browsers. 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. A common mistake is to rely only on blacklist validation, which is often trivial for attackers to circumvent, by utilizing alternative input variations. White-listing is in general considered much safer, only allowing the characters that the web application should accept, for example an integer or a date. In case of free-form text input, white-listing becomes difficult, as the users should be allowed to enter almost any character, hence the free-form. Any validation technique becomes ineffective and difficult to implement in the case of free-from text, which is why input validation should not be used as the primary defense against cross-site scripting attacks, and why output encoding is needed.

Output encoding Output encoding is the most effective remediation for cross-site scripting attacks when done properly. Output encoding should be implemented every place untrusted input is being output and rendered in the browser, making sure the input is displayed as data and not executed as code in the browser. It is important to implement the output encoding according to the context it is being used in, because different encodings are needed depending on the context used. JavaScript, HTML and URL's all use various encodings, which is why there are no single solution to how output encoding should be implemented. Typical strategies are to escape unicode, a typical character encoding, converting unwanted characters to benign equivalents, percent encoding and escaping hex values, as described in more detail in OWASP's XSS (Cross Site Scripting) Prevention Cheat Sheet [21].

Content Security Policy (CSP) Another powerful counter-measure is Content Security Policy (CSP), 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. To enable CSP, the web server needs to utilize the `Content-Security-Policy` HTTP response header [6], where the policy for the application is specified, including desired directives. Each directive describes a policy for a certain resource type or policy area, for example to prevent inline scripts from running, only allowing content to be loaded for some trusted domains or restricting all content to only load from the site's own origin. CSP also have a reporting feature, which means when a policy is being violated, it is possible to get a report sent to a desired location, containing information about the violation. This could be helpful for web application owners to know if their policies are too strict or needs modifications, as a policy can consist of many different directives. Even though CSP can stop most cross-site scripting attacks by utilizing a set of well-defined directives, it is stated in the World Wide Web Consortium (W3C) Recommendation [22] that CSP is not meant as a first line of defense mechanism, but rather an element in a defense in depth strategy, as an added layer of security. A study by Weichselbaum et al. [23] was done in 2016, including 1,680,867 hosts with 26,011 unique CSP policies, observing that 94.68% of all policies that attempts to limit script execution are ineffective, as well as 99.34% of the hosts have policies that offer no benefit against XSS at all. This is a very clear indication that CSP in practice is difficult to utilize correctly and this is why it should not be used as the primary defense against cross-site scripting attacks.

Same-origin policy Same-origin policy [24] is a policy implemented inside web browsers that isolates potentially malicious documents by restricting how a document or script loaded from a specific origin can interact with resources from other origins. For two web pages to have the same origin, they need to have the same protocol, port and host, which means they are allowed to load resources from each other. Cross-site scripting attacks often involve the usage of different external JavaScript files for collecting data from compromised users, which could be blocked by utilizing the same-origin policy.

HTTPOnly cookie flag As mentioned in Chapter 2.2.5, cookies could contain valuable information for attackers, which means they should be protected from unauthorized access. The `HTTPOnly` cookie flag is an additional flag included in the Set-Cookie HTTP response header [25], preventing JavaScript code from accessing the contents of cookies. This is not considered a counter-measure for XSS, but rather for mitigating the risk of an attacker accessing other users cookies in the case of an attack.

Disabling JavaScript A more drastic approach that would effectively stop XSS is to disable JavaScript, 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 a general solution.

2.3 Cross-Site Scripting Filters

Filters try to stop cross-site scripting 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 in 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 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 is possible to perform without sending the attack code to the server at all, as discussed in Chapter 2.2.3. This 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. There are several existing server-side filters, which typically needs to be integrated into the source code of the web application. A study made by S. Gupta and B.B. Gupta [3] has a quantitative discussion for server-side filters, discussing some of the state-of-the art techniques they are using. The study concludes that there are generally several flaws with server-side filters that needs to be addressed, like too much altering of existing code-base, long learning phase, as well as too many false-positives and false-negatives. The study also emphasizes that server-side filters do not detect DOM-based XSS attacks. With all the combined flaws and design limitations of server-side filters, it becomes evident that only relying on server-side protection is not enough, and why it is necessary with client-side filters as an extra layer of security.

Client-side filters Client-side filters are located in the client, which typically would be the web browser used to access web applications. Client-side filtering could 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 filters. 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 protect against all attack types of XSS and achieving good protection following the defense in depth strategy. This thesis focuses on client-side filtering, which includes a discussion of varies existing solutions, presented in the next sections.

2.3.1 Regular Expression Based Filters

Using regular expressions is a popular technique for client-side filters, where the filter is typically located between the network layer and HTML parser in the browser. Regular expressions are then used to identify potential malicious code in the HTTP requests and to approximate the rules of the HTML parser to know which content in the HTTP response that would be treated as script content [1]. By doing these approximations, the filter do not have to recreate the browser's own HTML parser, which would lead to the HTTP response being parsed twice, first for the filter to identify and remove potential malicious code and then for the browser to parse the page as normal. These approximations does, however, have their drawbacks, as they incur a higher number of false-positives, due to several flaws in their design [1]. These flaws is a consequence of attackers trying to make the content from the request, the actual attack code, differ from the response, so that the approximation rules would not detect it as an attack. Some common flaws are that the filters do not correctly approximate the decoding process of different encodings or do not take into consideration that different characters can be used to delimit HTML attributes.

A popular client-side XSS filter using regular expressions is an extension called NoScript [26], for the Mozilla Firefox browser, 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 [1]. Due to a lot of false-positives, NoScript try to solve this by prompting the user to repeat the request with the filter disabled, allowing the user to decide for themselves if they think it were a false-positive. This is a decent approach for security-aware users, but in general, users do not have the knowledge or desire to take action in the case of security-related issues [2].

2.3.2 XSS Auditor - A string-matching Based Filter

String matching is another method for client-side XSS filtering, used by the filter in the Google Chrome browser, called XSS Auditor. XSS Auditor works by matching the HTML code for injected JavaScript code from the request with the response from the website after it is been parsed by the browser's own HTML parser [1]. This means that XSS 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 XSS Auditor, which is between the HTML parser and the JavaScript engine, as shown in Figure 5. This placement makes it possible to block scripts after parsing, by blocking them from being sent to the JavaScript engine for execution. The location of XSS Auditor have benefits like performance, by not having to simulate the HTML parser, and the fact that the JavaScript engine have a narrow interface it is reasonable to assure that all scripts are being processed by the filter before being executed.

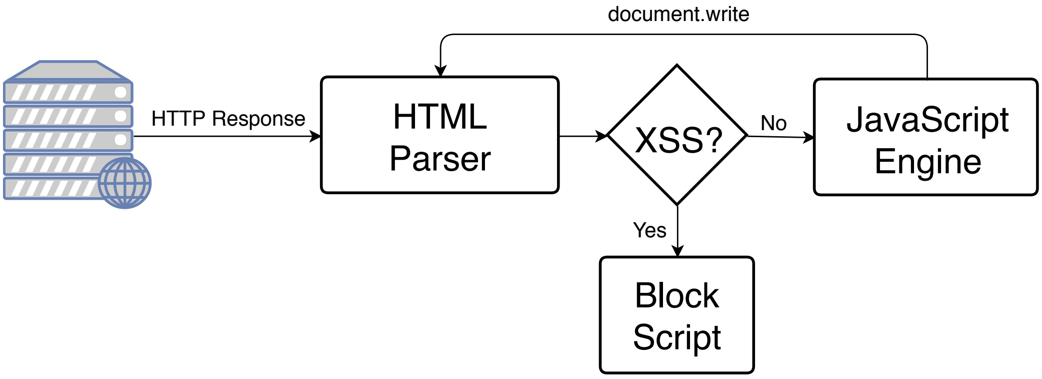


Figure 5: XSS Auditor design

XSS Auditor also have some limitations, some of which are discussed in the paper from Stock et al. [2], which lists several flaws with the design and string-matching algorithm used in XSS Auditor. As mentioned in the paper, these are mainly flaws regarding protection of DOM-based XSS, which is not the main type of attack that XSS Auditor is protecting against. It is, however, relevant to take notice of these limitations, as it might be desirable to not make the same limitations when designing and implementing a new filter.

Scope issues are related to the fact that XSS Auditor does not support every type of XSS or are neglecting functionality that enables XSS attacks. One example being that XSS Auditor relies encountering dangerous elements during the HTML parsing of the response, which is not always the case, for example, when a web page is using the JavaScript function `eval()` [27]. `eval()` is a function that evaluates the string representation of JavaScript code inserted inside its parentheses, which means if `eval()` uses data from the URL of the loaded web page, this evaluation could be done without entering the HTML parser, which means that XSS Auditor would not detect it.

Another flaw in XSS Auditor is that some special characters needs to be present in the request for the filter to be activated. If any of these characters are not present, the filter deactivates. As the paper describes, it is possible to successfully execute a XSS attack without any of these special characters being used at all.

Double injections is another limitation that XSS Auditor does not protect against, which is the inability to detect attacks containing concatenated values coming from more than one source of user input. An attacker could use two different input sources due to application specific code that concatenates two or more user inputs. When creating an attack using double injections, the exploit code consist of two or more parts, but gets executed as one concatenated attack code. Since XSS Auditor's string-matching algorithm checks for the whole script code, the algorithm would not detect the attack, as the whole script code does not exist from any single user input source.

2.3.3 State of Current Browsers

Regular expressions and string matching are among the techniques being implemented in the top five most used web browsers for desktop, which according to the desktop browser market share worldwide from StatCounter [28] are Chrome, Firefox, Internet Explorer/Edge and Safari. Table 1 contains information on the state of their XSS protection status. 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 [29]. 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 strategy is also weakened, where a potential filter would provide an extra layer of security for the end-users of the application.

XSS Protection					
Built-in filter	✓	✗	✓	✓	✓
CSP	✓	✓	✓*	✓	✓

* Limited support

Table 1: Top 5 Web Browsers XSS Protection Status.
Data retrieved from Mozilla [6]

2.4 Methodology

The work done in this thesis is to create a built-in filter protecting against Reflected XSS vulnerabilities inside the Mozilla Firefox browser. The choice of protecting against XSS for Mozilla Firefox is made for several reasons, one being that XSS vulnerabilities are of the most critical and prevalent web vulnerabilities in existence today with lacking protection mechanisms on both the server- and client-side of web applications [10] [8] [7]. This, in combination with the fact that Mozilla Firefox, which is the second most used web browser [28], does not provide a built in filter for XSS protection, in contrast with the other major web browsers, Chrome, Edge, Safari and Internet Explorer, which do have such a filter built-in. The work of this thesis will, therefore, be to create this filter built into and integrated with the existing source code of Mozilla Firefox, which is possible due to the fact that Mozilla Firefox is fully open source, allowing full access to the source code of the browser. This would be a case-study/pilot-case for the effect of building, integrating and running a filter protecting against XSS inside of Mozilla Firefox. As this is the second most used browser, with a market share of approximately 11.7%, as of the statistics from StatCounter's desktop browser market share worldwide for April 2018 [28], and the fact that XSS vulnerabilities are as prevalent as they are, it would be beneficial to look at a possible solution for adding this extra layer, the added filter, to the defense in depth strategy combining several XSS protection mechanisms for optimal overall protection.

For the work to be considered a possible usable solution, it needs to be evaluated throughly. There exists several different web browsers, all competing to being the best one, in terms of different factors such as performance, security, usability, customization and general look and feel. In such a competitive industry, web browser need to make sure that every included functionality is integrated and running as smoothly and efficient as possible, meaning an additional feature need to be well defined and robustly integrated. In the case of creating a filter for XSS, it needs to be secure, providing the necessary protection, and at the same time be efficiently integrated so the overall performance of the browser is not affected in any huge negatively direction. This means that the work done needs to be evaluated in terms of at least two different categories, how well it protects against XSS attacks and how much it affects the performance compared to Firefox without the filter implemented. The overall validation of the filter would be a qualitative research, as of how well the filter is implemented into the existing solution, but at the same time contain a quantitative method for measuring the performance of the filter, which could be accurately measured and compared to the original browser. By analyzing the performance number, however, it is not possible to correctly classify it as either right or wrong, but rather an estimation and analysis about if the added feature are in fact within reasonable limits to be considered a well performing solution.

3 Mozilla Firefox

Mozilla Firefox is a free and open-source web browser developed by Mozilla, with it's first major release in 2002 [30]. Firefox's 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 [31]. One of the primary requirements of Firefox is that it must be completely 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.

This chapter will explain some of the most relevant parts of Firefox, with regards to the filter created in this thesis work. The parts explained have been slightly simplified, making it easier to understand the relation of how everything is working together, again with regards to the added XSS filter.

3.1 Firefox Overview

The main components of Firefox can be divided into the user interface XML User Interface Language (XUL) and the browser and the rendering engine Gecko. XUL is Mozilla's own language for building portable user interfaces, which is an Extensible Markup Language (XML) language [32]. Gecko is Mozilla's browser engine built to support many different Internet standards, including HTML 5, Cascading Style Sheets (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 [33].

Mozilla also have a build system [34] using the `make` tool [35], consuming `Makefiles`. The command-line interface `Mach` [36] is used to help developers perform common tasks for working with the Mozilla codebase, making it easy to start building, debugging and testing Mozilla projects.

Firefox consists of over 36 million lines of code [37], written in several languages, which are mostly C++ and JavaScript, but also HTML, C, Rust, XML, Python and Java, as well as other less used. The source code directory of Firefox [38] contains of many folders where the code is grouped based on their functionality. Some of these groups consist of functionality related to document parsing, JavaScript execution, image loading, extensions and networking, just to mention a few. Mozilla also have strict rules about *how* the code should be implemented, not just how it is structured into directories. As mentioned above, Firefox is object-oriented, using a lot of public interfaces. They have also implemented several utility- and helper-classes for writing specific functionality inside their code-base. Although the source code is mostly written in the C++ language, which provides this functionality built-in, Mozilla uses many of their own methods for these functions. This means that it is necessary to acquire specific knowledge regarding these coding rules before attempting to make changes to the Mozilla codebase, as it is a complex piece of software.

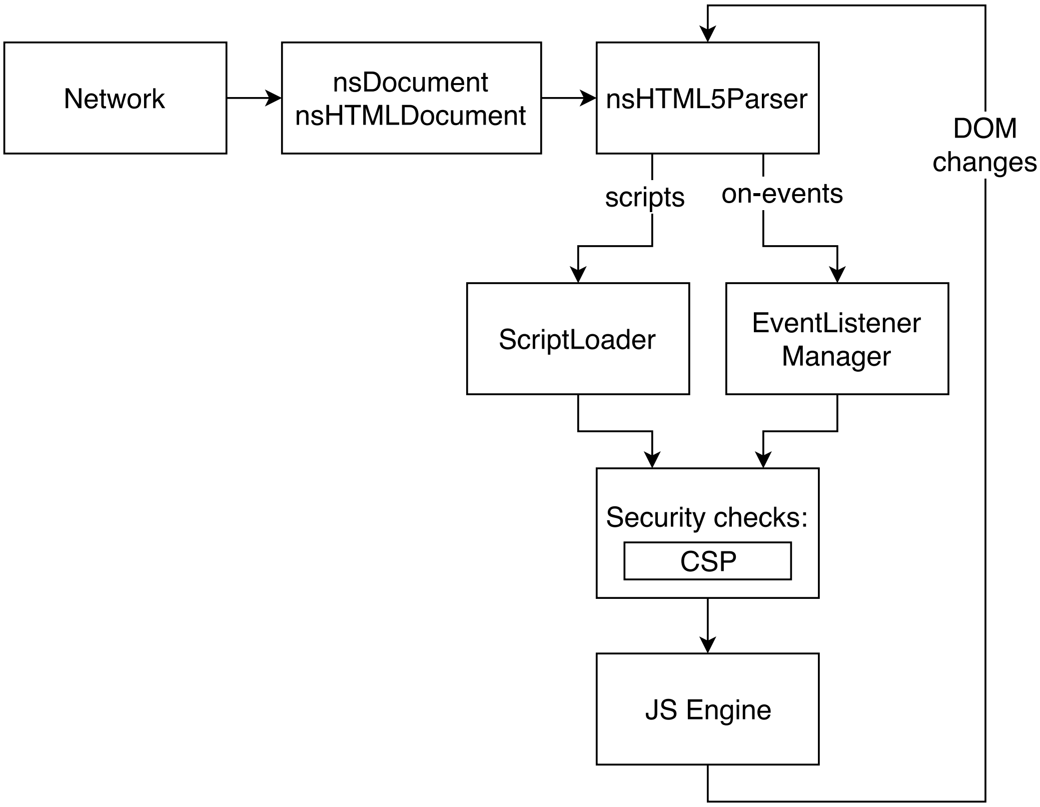


Figure 6: Simplified data flow for rendering a web page

3.1.1 Loading of a Web Page

As mentioned above, Firefox consists of several components, include its rendering engine Gecko, which is the most relevant part for the implementation of this filter, as it contains everything related to document parsing and handling of JavaScript execution. Figure 6 is a simplified description of the loading of a document in Firefox, containing only the relevant parts which are important regarding the XSS filter. When a typical HTML web page is loaded through Firefox, two internal document classes, `nsDocument` and `nsHTMLDocument`, are created, controlling the creation and representation of the web page to be loaded. These documents are responsible for creating and calling all the relevant parsers, like the HTML parser [39], `nsHtml5Parser`, as well as initializing the script executioner class, `ScriptLoader`, which is responsible for handling script content coming from `script` tags. The HTML parser receives data from the network that needs to be parsed. Every time the parser encounters some script content, the relevant parts of Firefox that handle this content is invoked. In the case of `on-event` handlers, the `EventListenerManager` class is invoked. A common source for script content is the `script` tag, where the script loader class, `ScriptLoader`, would be invoked with the discovered script. The script loader class will then try to extract the script and either execute it as an inline or external script. Before the script is passed to the JavaScript engine for execution, a security check is performed for finding out if the script is allowed to run. This security check involves checking with the CSP rules if it is allowed to load, if these rules are specified by the loaded website. If the script pass this check, it will be handed over to the JavaScript engine which will execute the script in the browser. The HTML parser will continue parsing the data entering through the network, repeating the steps when new script content is discovered.

3.2 Security Mechanisms

Firefox includes many internal security mechanisms for making sure that the browser itself is not being compromised by attackers, as Gecko loads JavaScript content from untrusted and potentially malicious web pages, which then again run on the user's computer. These security mechanisms include several complicated concepts regarding same-origin policy, compartments, and principals, all explained in detail at Mozilla's own website [5]. This section will try to give a simplified explanation of why all these concepts are important and how they are used. The reason why this is interesting to look at is because a counter measure for XSS, CSP, is implemented inside Firefox using the principal concept. Since CSP provide similar functionality as the work done in this thesis are providing, the filter created should also ideally be implemented in a way that follows the same principles, fulfilling the necessary security requirements.

3.2.1 Same-Origin Policy

The same-origin policy is restricting how a document or script loaded from a specific origin can interact with resources from other origins, as described in Chapter 2.2.6. The security model for web content is based on this policy, which is also used inside Firefox as a script security mechanism [5]. As Firefox's rendering engine Gecko consist of different languages, its core in C++ and its front-end in JavaScript, these to parts needs to interact with each other in a secure manner. The JavaScript front-end is actually running with system privileges, meaning that if it is compromised, attackers might get control of the user's computer. As this JavaScript code is interacting with web content from web applications, which again is susceptible to XSS attacks, it is important to make sure that JavaScript code from Gecko itself is not affected by any such attack, which is achieved by utilizing the principle of the same-origin policy.

3.2.2 Compartments and Principals

A security measure in Gecko is that it is divided into different `compartments`. Compartments could either be internal parts in Gecko or a content window, a typical website, where different parts can only access other parts if they are in the same compartment. The concept of compartments is, therefore, using the same-origin policy principle. Every part inside a compartment is, therefore, same-origin with the others and no additional security checks are performed when parts inside the same compartment talk to each other. If Firefox loaded the website at `http://example.com/subfolder/`, all the HTML elements and script content residing on this exact address would

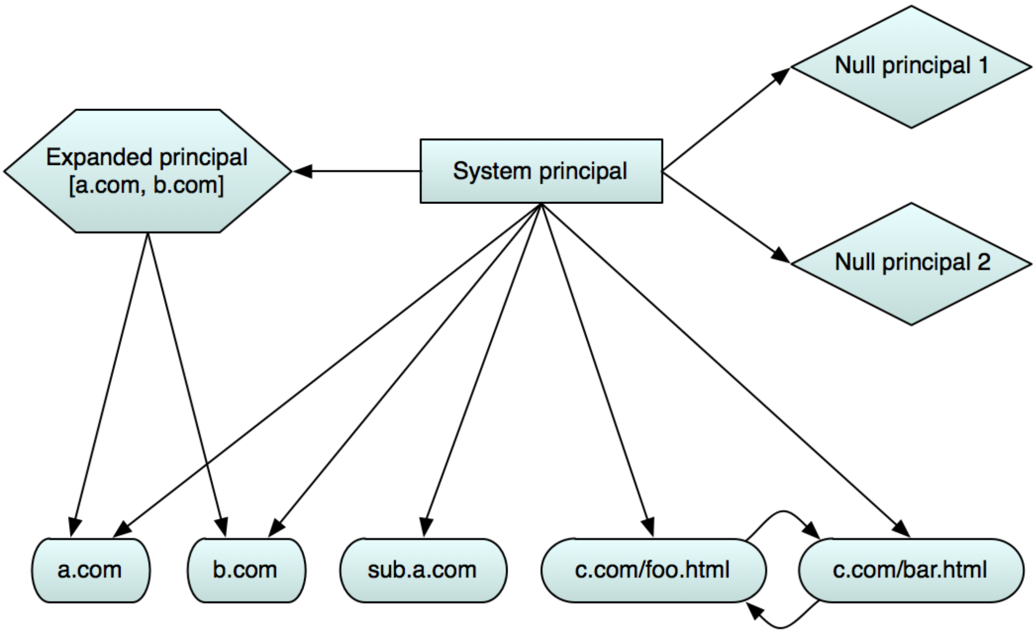


Figure 7: An overview of the relationships between the different security principals.
Figure taken from Mozilla's website, about "Script Security"[5]

be inside the same compartment. There are, however, different ways for compartments to access parts of other compartments, where the main rules are that higher privileged compartments have access to less privileged compartments, but not the opposite, unless the higher privileged compartment explicitly chooses to share its access.

To be able to determine the security relation between different compartments, a concept called `security principals` is used, which is something every compartment have. Figure 7 illustrates the relationship between different principals, as there are several different principals, each with its own rules. System principals pass all security checks, which is what the JavaScript code from Gecko is running with. Content principals are associated with web content, meaning that content from a specific origin could access parts from content inside the same origin. An expanded principal is specified as an array of origins, meaning that it contains several content principals. The expanded principal itself gets access to its contents, but the content principals within does not get access to the expanded principal. Finally, there is the null principal, which fails almost all security checks, meaning it has no privileges and can only be accessed by itself and the JavaScript code from within Gecko.

3.2.3 Content Security Policy (CSP)

Content Security Policy (CSP), as described in Chapter 2.2.6, is a security feature that is also implemented in Firefox. Since CSP is part of the script security model, it also has a principal. This means that CSP is created through a principal and access to it needs to be done through a principal. The main class, the `nsDocument` class, is the place where the CSP is initialized, by using a principal. As the `nsDocument` creates and holds a reference to the CSP Principal, other classes can get access to the CSP through the `nsDocument` class. Some noteworthy places that CSP is used inside Firefox are the script loader class, `ScriptLoader`, and the `EventListenerManager` class. These are locations which handle content related to script execution, and therefore also the place where the proposed filter in this thesis should be placed.

4 Design and Implementation

This chapter will go through everything from the development process of the implemented filter, including the requirements, design, tools used and the actual implementation of the solution.

4.1 Design Choices

Software development includes a lot of choices that need to be made during the development life cycle, regarding analyzing the problem, coming up with a solution, making the design and figuring out how it should be implemented. When creating a filter for Firefox defending against cross-site scripting attacks, it is possible to choose many different approaches towards the same main goal, but yet achieving differently in different categories such as performance, availability, usability, maintenance and of course security. In this section, some of the design choices made for this thesis work will be explained in detail.

4.1.1 Requirements

For the filter implementation presented in this thesis, there is a couple of preferred capabilities that should be achieved. These requirements are mostly based upon the analysis of the papers from Related Work, Chapter 1.3.

Usability The filter should be easy to use, by not requiring any user-interaction at all. The NoScript plug-in for Firefox, mentioned in Chapter 2.3.1, is an example of something that is not wanted, as NoScript do require a fair amount of user interaction, as the plug-in have a lot of false-positives. In a worst case scenario, a user might accidentally allow an attack to get executed, even though the filter did stop the attack and warned about it, as users might not understand what it means and the risk of ignoring the warnings.

Low false-positives It is important that the filter do not interfere with a user's normal browsing sessions, unless it is to protect the user from an actual attack. To achieve this, the filter should have a low number of false-positives, which means that the filter should minimize the number of times where it think there is an attack when in reality it is not. The opposite of a false-positive is a false-negative, which is when the filter thinks a script is safe to load when in reality it is an attack and should be blocked. In practice it is difficult to guarantee both non-existent false-positives and false-negatives in a filter meant for defending against cross-site scripting attacks, as there are so many different ways of using JavaScript in web applications, which again is one of the reasons why cross-site scripting attacks are so prevalent. There is, however, a balance to be made, to make sure that the filter do protect against most attacks, which means it might introduce some false-positives, but at the same time it cannot be too strict either. An example of a too strict filter is again the NoScript plug-in for Firefox, which is really aggressive, introducing a lot of false-positives which would interfere a lot during normal browsing sessions, again requiring user interactions as a workaround.

High performance The filter should not incur a lot of performance overhead, which would make the loading of web pages slower, which again would interfere with the usage of normal web browsing. When using the filter, there should be no noticeable delay when loading web pages in comparison with the version of Firefox without the filter. This is an important requirement, because of the competition between web browsers, as discussed in Chapter 2.4.

Provide protection against Reflected XSS The whole point of a filter protecting against cross-site scripting attacks is to provide this protection properly. As there exists several different types of XSS, as discussed in Chapter 2.2, it is important to clarify that the main focus of the filter is to protect against the Reflected XSS type. This is the type of XSS that filters for the other major web browsers also primarily focuses on, as it is very prevalent and the easiest to discover, as described in Chapter 2.2.2. It is, however, desirable to also protect against DOM Based XSS, which there will be some basic protection against, as a byproduct of the Reflected XSS protection. Complete DOM Based XSS support will, however, be lacking, as in the case of XSS Auditor, as explained by Stock et al. [2].

4.1.2 Browser Extension vs Internal Implementation

This thesis work is to add some functionality to the Firefox browser, which there are several ways of accomplishing. Firefox do provide support for browser extensions [40], which can extend and modify the capabilities of the browser. These extensions are built using JavaScript, HTML and CSS by using the WebExtensions API, a cross-platform system for developing extensions. They can provide a lot of functionality for altering the contents of or extracting information from a web page, either with or without required user interaction. There are, however, some reasons why browser extensions are not suitable for this thesis work, explained in the following paragraphs.

Availability The main reason why browser extensions are less suitable is because they are something that users themselves need to find, install and use. It should not be necessary for users to know about what cross-site scripting is and why it is important to protect against it, for them to take advantage of this filter. By making this protection a choice for the user, the filter would most likely not be used by the majority of users. This is why an integration with Firefox itself would be a better solution, as then all users would take advantage of the filter without the need of any knowledge about it or action required.

Performance Even if there are users choosing to install and use such a security filter, there is another drawback by making it as a browser extension, which is a performance issue. When creating a browser extension for Firefox you can only use the API's supported by Firefox [41], utilizing JavaScript code that talks to the internals of the browser itself. This means there are more layers that the data needs to go through, from getting from the filter to the internals of Firefox, which is needed for functionality of the extension to work. If the filter, however, is placed inside the internals of Firefox, some redundancy will be removed, which again will lead to a better performance, which is what is chosen for this filter design.

Security The purpose of the proposed filter is to protect against Reflected XSS attacks, which means the injected script is contained in both the request and response. By implementing the filter as a part of the internal implementation of Firefox, it is easier to have a more robust integration being more secure, as Firefox have a lot of coding principals including many security features, as described in Chapter 3.2.

4.1.3 Blocking Technique

When detecting an XSS attack, the filter needs to take action to block the injected script. There are mainly two ways of doing this, either blocking only the injected script or blocking the whole web page from loading. By only blocking the injected script you interfere less with the browsing experience of the user, as they can still use the website as normal, without the parts potentially affected by the injected script, which is what has been chosen for this proposed filter.

4.1.4 Filtering Technique

As discussed in Chapter 2.3, there exists XSS filters based mainly on the two filtering techniques regular expressions and string matching. For this thesis work, the string matching technique and design from XSS Auditor was chosen as the main basis. XSS Auditor used in the Google Chrome browser does achieve high performance, few false-positives and low interference with normal web browsing, providing protection against mainly Reflected XSS attacks, as desired from the requirements in this thesis.

4.2 Design Overview

The main design of the filter is to compare every script returned in the response with every potential dangerous script from the request. If there is an occurrence of a script appearing in both the request and response, the cross-site scripting filter will block this particular script from being executed. The filter itself is structured as its own class inside Firefox's source code, which makes it easy for other components in Firefox to use the filter when needed. The filter is placed after the HTML parser, but before script execution, providing benefits regarding both security and performance. The following sections will describe the design of the filter in more detail.

4.2.1 Placement

By basing the solution on the filtering principals of XSS Auditor, the placement in Firefox will also be similar to how Auditor is placed inside of Google's Chrome browser. Auditor is placed between the HTML parser and JavaScript execution environment, which provides several benefits, regarding high security and performance, as explained in Chapter 2.3.2.

The filter needs to know what Firefox would intercept as script content to be able to filter on the correct data. If the filter was placed before the HTML parser, the filter would need to simulate the rules of the parser to try to approximate and identify what Firefox would intercept as script content. This means that each loaded document would be parsed twice, once from the filter and once from Firefox's own parser, which would incur a lot of performance overhead. Since Firefox need to parse the HTML documents regardless of the filter's presence, by placing the filter after the HTML parser, it can use the results from Firefox's own parsing when determining which content to filter on, which again would not add any extra performance overhead regarding the actual parsing process. Since the filter do not need to approximate the parser rules when placed behind the HTML parser, the filter can also be sure that it will discover, identify and act upon all the scripts entered through Firefox, as the parser in Firefox will properly identify all script content before they are processed further. As explained in Chapter 3, script content from `script` tags and `on-event` handlers get sent to the classes `ScriptLoader` and `EventListenerManager`, which will further examine the data and conduct the necessary security checks before they are sent to the JavaScript engine for being executed, as shown in Figure 6. By extending on this figure, extracting the relevant parts, Figure 8 shows the placement of the XSS filter, residing in the same location as the CSP security feature.

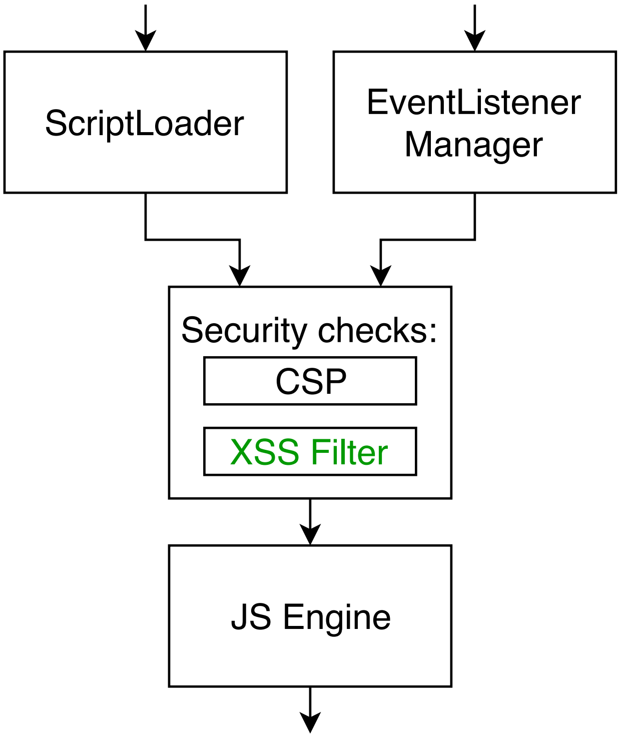


Figure 8: XSS Filter placement inside Firefox

4.2.2 Filter Class Structure

The filter class contains many methods for handling the different stages needed in the filtering process. Since the filter can be invoked from different locations, the filter class contains several input points that all starts the filtering process. This process contains a series of different tasks that are performed in a particular order, before concluding whether there exists a cross-site scripting injection or not. This includes methods for fetching the input from the request to different methods for comparing this data with either inline script, external script or on-event handlers, all of which need to be processed differently.

4.3 Environment

This section will describe the system and tools used when developing the Firefox filter. The operating system used is Arch Linux [42], a lightweight and flexible Linux distribution. For developing and writing the source code, the free and open-source text editor Visual Studio Code (VS Code) [43] was used.

4.3.1 Tools

Several different tools were utilized during the development of the proposed filter.

Development Software When developing computer software, there exist several Integrated Development Environment (IDE) and code editors with a lot of added functionality for helping with software development. For the development and writing of the source code for this thesis work, a lightweight, free and open source text editor, VS Code [43], was used. VS Code provides the necessary syntax highlighting and autocomplete, while also making it easy to navigate around in the huge Firefox source code. Without adding extra additions to VS Code, it does not handle building and debugging of the Firefox code, which is one of the reasons it is a lightweight editor. For these operations, however, there are more specialized tools that are better suited for the development of Firefox, as Mozilla have their own recommendations and tools available.

Mach As mentioned in Theoretical Background 2, the tool `mach` [36] is a command-line interface used to start the building, debugging and testing of Mozilla projects, which also was used in the development of this modified version of Firefox. `mach` makes it possible to configure Firefox builds through the usage of a `mozconfig` configuration file [44].

GNU Project Debugger (GDB) For debugging, GNU Project Debugger (GDB) [45] was used, a tool that can start programs, make it stop on specified conditions, examine what is happening at runtime and change things in the program as it runs. GDB is a tool that can be invoked using the `gdb` command, but when debugging Firefox it is possible to start GDB through the usage of the `mach` command. After starting the debugging mode, GDB makes it possible to create breakpoints in the code, which lets the debugger inspect the state of the application as it is running.

4.4 Implementation

This section will describe the implementation of the filter, how it is implemented and integrated into Firefox, also containing details about every part of the filtering process.

4.4.1 Data Flow

The data flow in Firefox is illustrated in Figure 6, found in Chapter 3.1. This figure is then being expanded in Chapter 4.2.1, Figure 8, where it is shown that the classes `ScriptLoader` and `EventListenerManager` perform several security checks, including using the XSS filter. When the `XSSFilter` class is being invoked from these classes, it first need to get all the input data from the request. This data is retrieved through the `nsDocument` class. The relevant input data fetched are all the GET- and POST-parameters contained in the request. These parameters are saved in a list, which is then examined further. Every parameter is checked if it contains any potential malicious content, which in the case of a cross-site scripting attack would be any input that contains some form of script content. This examination is explained further below, in the next section. If the filter identifies any parameters as potentially unsafe, it will compare them to every script entered into the filter class, from the `ScriptLoader` and `EventListenerManager` classes. If any of these scripts are also found in the request, the filter will mark the script as unsafe, which will again notify these classes to not send the detected script to the JavaScript engine for execution. All the other scripts will be executed as normal.

4.4.2 Examining Input Data

After fetching all the GET- and POST-parameters from the request, these needs to be analyzed for potential malicious content, which as mentioned above, would consist of any type of script content. It is not a simple task to identify whether or not these parameters contain any actual script content, as there exists many different ways for creating and trying to hide the malicious content of a parameter. A good source of many such attack payloads is OWASP's guidelines "XSS Filter Evasion Cheat Sheet" [20], which contains many examples of injections trying to circumvent typical XSS filtering techniques, including variations of using the `script` tag, `on-event` handlers, as well as other, less used attack vectors. This is why the filter does not actually identify any script content in the parameters before marking them as potentially unsafe, but rather make an assumption based on their contents. If a parameter only contains alphanumeric characters, `[a-z] [A-Z] [0-9]`, or the underline character, _, the parameter is considered safe, and should not be processed further by the filter. These are very common characters that can not be used to execute any scripts, making them safe to include in the response. The reason why the underline character is included is that it is often used in the case of a space in a parameter, which should be considered safe. If there any other characters than the one specified, the filter would include the parameter in further processing, which will be described in more depth in the next section.

4.4.3 Looking for Injections - Matching Algorithm

If there are any potentially harmful content in the request parameters, for every script received in the response, the filter is running a matching algorithm which tries to identify whether any of these scripts are also contained in any of the parameters. Depending on the type of script received from the response, the filter handles the matching a bit differently. With inline scripts, a comparison of the string representation of the actual script content is done with each and everyone of the script content from all the inline scripts entered through the `ScriptLoader` class. `ScriptLoader` also handles external scripts, in which case it first gets the information about the external URL where the actual script is located, before it executes the content inside the script. For the filter, in the case of an external script, it does not do a comparison between the contents of the external script with the parameters, but rather a comparison between the string representation of the external URL and the parameters. As for other attack vectors, like the `on-event` handlers, the same approach as the inline script matching is done. A similarity between the inline and external script matching, however, is that before the actual matching takes place, the content from the scripts and the parameters need to be normalized. This means that these contents might differ slightly, as the parameters content might have changed after going through the HTML parser in Firefox, which again means that some of the same changes need to be done by the filter for it to detect all injections properly. Several possible factors that need to be addressed when normalizing the contents are listed below, with basis in the rules from OWASP's filter evasion cheat sheet [20].

Basic evasion techniques A basic normalization technique is to not differentiate between upper- and lower-case characters. The script injection `<script src="http://xss.rocks/xss.js"></script>`, which try to load an external script through a different domain, and the slightly different `<script src="http://xss.ROCKS/xss.js"></script>` would thus both be treated as the same injection, as the uppercase characters in the second example would be converted to lowercase. Another basic technique is to use added whitespace or other characters that does not change the behavior of the injected script, but that tries to hide the script from being recognized by filters. An example attack could be the injection `<script>alert (1)</script>`, where additional spaces is included, but where the injection could successfully execute the script content, `alert(1)`. This is related to using different encodings in the injections, which could include more advanced attack payloads.

Different encodings It is common for attackers to use different encodings in their attack payloads, by for example using URL encoding [46] for the injected script, which again is a means of hiding the injected string. URL encoding is something that needs to be used in URL's when the URL contains characters outside the American Standard Code for Information Interchange (ASCII) character encoding set, which is why the URL has to be converted into supported ASCII format. This is done by replacing unsafe ASCII characters with a percent sign, `%`, followed by two hexadecimal characters. It is also possible to use this encoding for any input for a website, which means the filter needs to properly decode and identify the encoded data. In this filter's implementation, it is supported by using Mozilla Firefox's own internal class for handling URL's, which also handles decoding of URL encoded data.

Different attack vectors The attack vector for injecting XSS attacks used in most examples in this thesis, utilize the script tag, `<script>`. It is, however, possible to perform XSS injections by using many other different attack vectors, as explained in Chapter 2.2.5. The filter does currently support the `script` tag and every usage of the `on-event` handler, which may be used in combination with many different attack vectors.

4.4.4 Handling of Discovered Script

If the filter does find a match between a script from the response with a script from the request, it marks that particular script as unsafe and notify the class that invoked the filter, telling the class that it should not execute this particular script. Even if a script is detected and blocked, the filter do continue to check all other script from the response with the request parameters, as there might be more than one injected script. This is an important aspect of the filter, as it only blocks the actual injected script and not the whole page from loading. By choosing a different solution where the filter is blocking the whole page when an attack is detected, the filter does not need to do any further checking, as you can not execute any more scripts as the page is not being loaded.

4.4.5 Firefox Integration

This section will briefly describe how the filter class is integrated and how it connects to other parts of Firefox. The filter is implemented as its own class inside Firefox's source code, called `XSSFilter`, making it easy for other components to use the filter when needed. The class is located in the `mozilla/dom/security` folder, which is the same location as where all the Content Security Policy (CSP) related classes resides. The filter is currently being created in places where the filtering functionality is needed, by supplying it with the owning document class, `nsDocument`, in its constructor. As discussed in Chapter 3, `ScriptLoader` is one of the primary classes that uses the filter. Upon creation of the `ScriptLoader` class, it also creates a filter instance with the main document in its constructor. Every time the main document are loading new data, like updated GET- and POST-parameters, the `XSSFilter` instance located in `ScriptLoader` also gets updated, fetching the new request data, before using it in the filtering process every time `ScriptLoader` encounters a script, either inline or external. Another internal class in Firefox, `EventListenerManager`, do also use the `XSSFilter` in a similar manner, but rather than inline and external scripts it takes care of scripts from `on-event` handlers.

The `XSSFilter` class itself is also accessing other components inside Firefox. To retrieve the GET parameters it has to access the URL from the main document class before using the `URLParams` class for parsing it correctly, making sure the content is properly URL-decoded. As for the POST parameters, the filter gets access to the `nsIHttpChannel` class through the main document, which contains the necessary data for retrieving the parameters, by utilizing different helper classes in Firefox. It also uses several helper classes for a lot of string manipulation, operations like searching for whole strings or single characters, or converting between different types of strings and encodings.

4.4.6 Challenges

There have been some challenges with the implementation of the filter. Since the filter is being implemented inside an already built software, the Mozilla Firefox web browser, the filter needs to be integrated in a way so that it can cooperate with existing code, data flow and different ways of doing things. Mozilla Firefox is a very huge piece of software, containing many different classes spread across separated modules that talk to each other by using different means. To properly understand this whole structure and following the data flow proved to be a challenging task, as there were used a lot of different coding principles and internal code for different tasks. String-handling is a good example of how complex the code is, as there exists many different types of strings and as many ways of converting between them and utilizing them correctly.

4.4.7 Unit Testing

Unit testing is a good way of assuring that separate parts of the code is working as desired. In the case of the filter implementation, the parts containing the examination of input data and the matching algorithms are the most important to test, as these are the parts dealing with the actual filtering process. Several unit tests have been implemented to verify this process, by supplying some sample injected data. As the filter require some special characters to be included in the parameter for it to be checked for in the matching process, several tests have been implemented confirming these character checks. The matching algorithm also have several tests with different injection inputs, verifying that the string matching works correctly. As for testing other parts of the filter, which relies on many different parts of other functions in Firefox, a more complete testing is done in Chapter 5.

5 Analysis and Assessment

The filter needs to be evaluated, as explained in Chapter 2.4, in terms of several different categories. The filter should be tested for how well it protects against XSS attacks and how much it affects the performance of Firefox . An analysis of the filter's implementation, some of the design choices and different limitations are also an important part of the evaluation, as it will highlight what is good and what needs to be improved.

5.1 Protection Effectiveness

Protection effectiveness is about how well the filter is able to protect against XSS attacks, in particular Reflected XSS attacks.

5.1.1 Methodology of Testing

To be able to measure the effectiveness of the filter, it is necessary to conduct testing by doing an examination of a known vulnerable website, as it is not the website's own security features that need to be tested, but the filter's capabilities. One way of making sure this is the case is to implement a sample website, used for the sole purpose of testing the filter. The created website should try to mimic some of the functionality found on other typical websites, as this would provide a better generalization of the filter's overall effectiveness. A common functionality found on a majority of websites is the search field, which is also susceptible to Reflected XSS attacks. The website should therefore consist of a search field, which would send the query to a web server, where the response should be a page containing the input query from the search field. Since the website has no built in security features, inputting a script into this search field would effectively execute it upon receiving the response. By visiting this vulnerable website through the modified version of Firefox, containing the XSS filter, the filter should be able to both detect and stop the injected script from being executed. This is being tested by conducting an automated test consisting of several different script injections, to see if the filter detects all of the attacks, or just a subset of them. The automated test is made possible by the usage of Selenium WebDriver [47], which makes it possible to do direct calls to a specific web browser instance, by using its native support for automation. A simple script will be created that uses Selenium, which takes a list of injections as input, which then will test each of them against the sample vulnerable website. The outcome of this script will be a list of both the successfully injected scripts and the ones that did not get injected.

The script injections that are to be tested, are collected from a variety of sources. An extensive list found on the website `gbhackers.com` [48], and three different collections gathered from `github.com` [49] [50] [51]. In total, a list containing 920 unique script injections where created from these sources. This list consist of many different attack vectors targeted at very specific functionality of common websites. Since the sample vulnerable website created is a very simple website, not containing a complex usage of different HTML tags, it is assumed that most of the injections would not be successfully injected. This is why several hundred injections was collected, to make sure that a big enough subset would actually be successfully injected, which could be used in the analysis. For achieving accurate results, the automation testing script would actually need to be executed twice. This process is shown in Figure 9. First, all the injections had to be tested against the vulnerable website *without* the filter enabled. This way, all the injections that are actually working on the vulnerable site, would be recorded in a list created by the testing script. Next, the list of injected scripts would be used to run the testing script another time, this time using a version of Firefox that has the filter enabled. The script would once again create a list containing both the successful injections and the injections stopped by the filter, which then would be used for further analysis. This is done to make sure that the analyzed results are containing actually injectable script content, so that it is known that it is the filter that stop the injections, and not something wrong with the injections themselves.

5.1.2 Results

When running the automated test as described above, the website without the filter was successfully injected with a total of 138 different script injections. Although many of the injections used similar attack vectors, there were still a good mixture of different attack vectors and encodings used, typ-

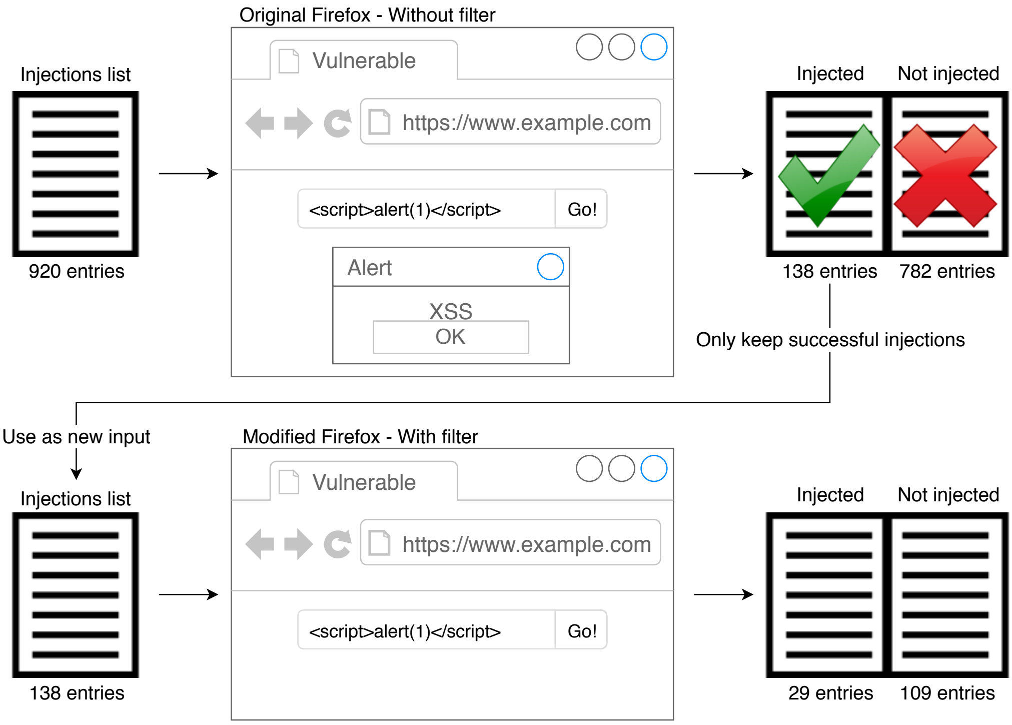


Figure 9: Testing of the implemented filter

ically trying to circumvent filtering mechanisms. When using these injections in the version of Firefox containing the XSS filter, only 29 were successfully injected. The filter did, therefore, block 109 of 138 injections. By examining the results further, it is possible to pinpoint the weaknesses of the filter, which again could be used to improve it.

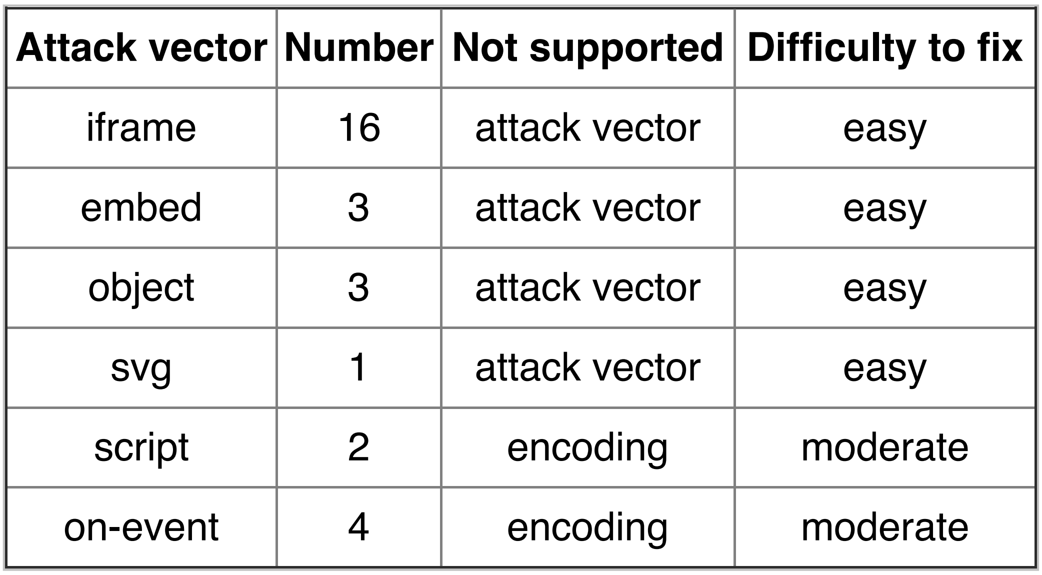
Blocked scripts Most of the script injections were both detected and blocked by the filter. This included the usage of many different variations of the `script` tag, were the injections were adding other unnecessary characters or using URL encoding trying to circumvent the filter. Because of the filter's location, behind the HTML parser, and the fact that all parameters gets URL-decoded, all of these injections were blocked. There were also a lot of usage of `on-event` handlers, utilizing similar circumvention techniques. Most of the `on-event` handlers were also blocked, used in combination with different attack vectors like the `img` tag, `svg` tag and `body` tag, since all of these `on-event` handlers had to go through the `EventListenerManager` class, were the filter was invoked from.

Injected scripts As there were a total of 29 successful injections, it is interesting to analyze why the filter did not detect them. Table 2 contains an overview of the injections, which will be further analyzed here. 16 of 29 of the successful injections used the HTML tag `iframe`, in different forms, utilizing upper/lower capitals, URL encoding and otherwise including different characters to confuse the filter. The `iframe` tag allows web pages to load other web pages inside itself, where the tag also support the usage of `on-event` handlers. Even though 16 of 138 of the `iframe`-injections got successfully executed, the filter did actually block the instances utilizing `on-event` handlers, as this is well supported by the filter. The filter does not, however, detect `iframe`'s using the `src` attribute, as it is not being invoked in the parts of Firefox that handles the script content inside these tags. 7 other injections were also cases were the attack vectors are not supported, which used the `embed` tag, `svg` tag and the `object` tag. The last 6 cases, however, used either the `script` tag or `on-event` handlers, but did not get detected. This is because they used varies encodings, like HTML entity encoding and base64 encoding, which are not supported by the filter. This is a good example showing that only dealing with the most common URL encoding is not enough, as there exists several other encodings that might be interpreted as script content by the website, that also needs to be considered.

5.1.3 Limitations

There are several limitations regarding the capabilities of the filter, which could be categorized into several categories. Some limitations were related to the actual filtering rules, which means the capability of the filter to detect different types of script injections, using different methods for trying to circumvent the filter. The other types of limitations is related to the different input and output sources supported by the filter, as there are more ways than using script tags and `on-event` handlers to inject script content into websites.

Limitations regarding filtering rules As described in Chapter 4.4.3, the filter did support URL encoded data, which turned out to work really well,



Attack vector	Number	Not supported	Difficulty to fix
iframe	16	attack vector	easy
embed	3	attack vector	easy
object	3	attack vector	easy
svg	1	attack vector	easy
script	2	encoding	moderate
on-event	4	encoding	moderate

Table 2: Testing of the implemented filter

stopping several injections. It did not, however, support HTML entity and base64 encoding, which led to script injections being executed in the browser. Support for more different encodings should therefor be implemented.

As seen from the results, every injection utilizing the `iframe` tag were successfully injected and executed in the browser, as this was one amongst several injections in which the injected attack vector was not accounted for by the filter. This is a general limitation that the filter simply supports too few attack vectors utilizing different HTML tags. Although the filter supports `on-event` handlers, which is used by a vast amount of HTML tags, these `on-event` handlers are not always necessary to trigger a script for execution, which is why this support needs to be improved.

In Chapter 2.3.2, some limitations of the XSS Auditor filter were discussed, which are tightly related to the limitations of this filter implementation, as they are based on the same string matching design. Not all of the limitations from Auditor applies though, as this filter does not require the same strict subset of special characters to be present, as Auditor requires. However, the limitations regarding partial string injections is something that has not been addressed in this filter either. If a website have several input fields were its content gets concatenated without proper validation, an attacker might take advantage of this to create a complete injection by splitting the injection into two or more fields. It is worth mentioning that this is a rather special case, as the website need to have some very specific functionality for this attack to work, but it is still a possibility that should be considered to be addressed.

Limitations regarding request input sources Another type of limitation is regarding every input source from the request, which means every source of user modified fields that might enter into a web application. The absolutely most used input sources are the GET- and POST parameters, which are currently the only sources supported by the filter. There are, however, other possible input sources where users could inject malicious content, like for example HTTP headers and cookies. Although these are more special cases, were the web applications need some more specific use-cases, they might still occur, which is why they should be considered to be supported.

5.2 Performance

The performance of the implemented filter is an important factor for its usefulness. For measuring the performance, Mozilla's own methodology for comparing page load times across browsers [52] was used. This methodology consists of choosing a set of websites that are loaded in Firefox, repeated several times, while measuring the loading time for each page load. This is a process that is automated with the help of Selenium WebDriver [47], which makes it possible to make direct calls to specified browsers using their native support for automation. For this implemented filter, it would be interesting to compare the performance of the modified Firefox instance with the original Firefox instance, which does not include a built-in XSS filter. By using the Selenium WebDriver it is possible to supply both of these instances as options, which means that the testing would be fully automated. As mentioned, it is necessary to have a set of websites to be used for testing. In the case for Firefox's own testing, they chose to pick the 200 most popular websites from the Alexa page rank site [11], because news sites typically contain a lot of trackers.

5.2.1 Methodology of Testing

For the testing of this filter, news site are also well suited, as they contain a lot of script content and most often also contains a search field for looking up articles, which is something that is useful for invoking the filter mechanism. For the testing, only a subset of the most popular news sites from the Alexa page rank site where chosen, as not every news site had a working search field. A total of 20 news sites where selected for the testing. It is assumed that most of the top news sites can be considered to be relatively safe, not containing any easy to exploit cross-site scripting vulnerabilities. This does not, however, hinder the filtering mechanism to activate, since the filter would still search the request parameters for potential dangerous contents, and do the comparison between them and the scripts contained in the response. This is done regardless of the existence of any actual vulnerabilities or not, since that is the whole point of the filter, to act as an added layer in the defense in depth strategy trying to stop attacks from potential vulnerabilities.

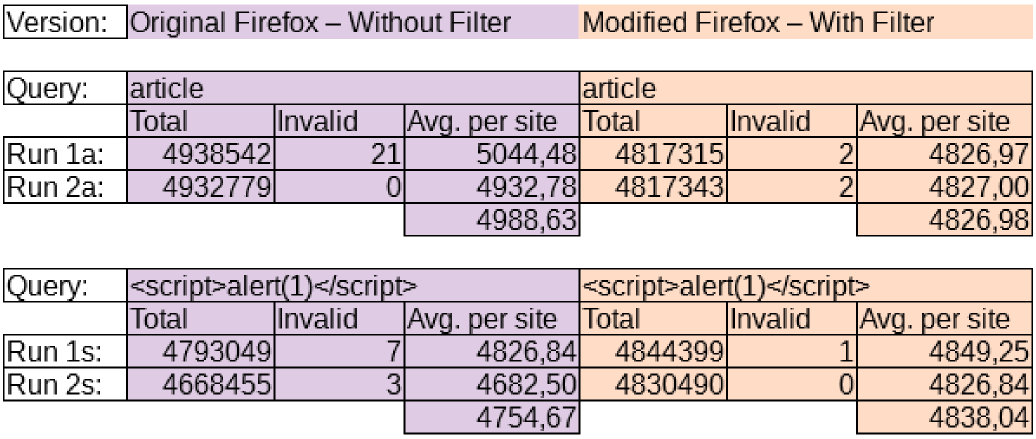
To make sure the modified browser actually runs the code for the implemented filter, each website was given some input data by using their search fields. The testing was done with two different input data, with the first one simulating a totally legit request that does not contain any script content at all, inputting the query `article`, and the second one containing a simple script, `<script>alert(1)</script>`, simulating a very simple XSS attack. In the first case, by inputting a safe query, the filter would inspect this query and not find any potentially dangerous characters, which means the filter would not need to do any additional processing. In the second case, the same inspection of the query would be done, which would mark the injection as unsafe. After marking it as unsafe, every time a filer would get a script from the response, this script would be matched against the unsafe parameter, trying to identify if the parameter is contained in any of the scripts. The performance difference between the original and modified browser should be expected to be lower from the first case than the second case, as the filter is doing more work the second query. One thing to notice here is that the filter would most likely not detect an actual attack, as previously assumed that popular news websites are probably protected against simple injection attacks.

5.2.2 Results

After running the automated test, the result does not suggest any added performance overhead by including the filter. The measured load times were actually so similar that an accurate estimate of how much the filter affected the performance is not possible to measure. Table 3 illustrates the results, where the unit of the load times are milliseconds. The columns marked "Invalid" means that a web page did not load correctly, which means it got removed when calculating the average load time. In the case of loading web pages with the query `article`, the version containing the filter did actually perform approximately 3.2% faster on average, than Firefox without the filter. In the case of using the query `<script>alert(1)</script>`, the original Firefox version performed approximately 1.7% faster on average. It is worth noting that the results did not contain any huge fluctuations when performing the test, and the biggest difference after calculating the average for each test run was about 362 ms, which was the difference between Run 1a of the original version and Run 2s of the original version. The difference between the different runs of the modified filter were really small, as seen in the figure. As the total difference between the original and modified versions are also relatively small, the conclusion is that the filter did not add any measurable performance overhead, meaning it achieves very high performance. There are, however, several factors that might have affected the testing, as described in the next section, 5.2.3. Although, since there were so few fluctuations between the calculated averages, it is assumed that the results reflect the reality fairly well.

5.2.3 Limitations

There are several factors that might have affected the performance testing, which could mean the result are misleading. When Mozilla did their own performance testing, they used a total of 200 different websites, a number much higher that what was used when testing this filter. Choosing a larger subset of websites for the testing could have given some results reflecting a more average loading time, but the 20 selected websites did achieve a very



Version:	Original Firefox – Without Filter	Modified Firefox – With Filter

Query:	article			article		
	Total	Invalid	Avg. per site	Total	Invalid	Avg. per site
Run 1a:	4938542	21	5044,48	4817315	2	4826,97
Run 2a:	4932779	0	4932,78	4817343	2	4827,00
			4988,63			4826,98

Query:	<script>alert(1)</script>			<script>alert(1)</script>		
	Total	Invalid	Avg. per site	Total	Invalid	Avg. per site
Run 1s:	4793049	7	4826,84	4844399	1	4849,25
Run 2s:	4668455	3	4682,50	4830490	0	4826,84
			4754,67			4838,04

Table 3: Loading times results, measured in milliseconds

small variance in the calculated average, so it should not be of much difference if choosing to include any more than this. Some other factors that might have had more impact on the results are fluctuations in the local Internet speed of the testing machine and the fluctuations in the web traffic received by the tested websites at the time of the testing. It is typical that these factors varies throughout the day, depending on the time. The test of the original and modified browser were done consecutively, where each test, where one test contains loading of 1000 websites, took approximately 80 minutes to perform. This is not a very huge time span, meaning these fluctuations should not be considered to be of any huge significance. Another factor that is less likely to have affected the performance is the processing power of the testing machine itself, meaning the CPU of the machine might have been running different tasks when conducting the testing of the different browsers. The testing machine was, however, left alone during the actual testing period, which should result in minimal affection from other tasks running.

These are all limitations that somehow might have affected the testing results, some easier than others to control and minify, which was done to the best of ability. Each of them should not be of any significance, and the results are considered to be very accurate, but it is still worth mentioning these limitations, as is is often small variances in the results which should be tried to be explained.

5.3 Implementation

It is also interesting to analyze how well the filter itself is implemented, in terms of how well it is integrated into Firefox, and how it affects the usage of Firefox other than the already measured performance.

5.3.1 Conform to Mozilla Firefox's Internal Coding Standards

Mozilla Firefox have strict guidelines for how things should be integrated into the browser, a coding standard for everything from simple formatting to the usage of different parts from the code. The implemented filter has tried to comply to these rules, by following the general coding standards, particularly regarding handling of strings [53], as string matching have been a major part of the filter mechanism. Getting access to other parts of the code, parsing data correctly, exception handling and testing is other examples good implementation regarding Mozilla's coding principles. There is, however, one aspect of the implementation that is not being integrated well enough for being part of a release version of Firefox. This is the fact that the filter is not utilizing the concept of script security and the usage of principals, as explained in Chapter 3.2.

5.3.2 Blocking Technique

When detecting a potential XSS attack, the filter should be able to act upon it and block the script injection. There are several ways of doing this blocking, as mentioned in Chapter 4.1.3, it is possible to only block the injected script, or the whole web page. Both of these techniques have their advantages and disadvantages, which are being discussed here.

Partial blocking One of the reasons for blocking only the injected script is that it would interfere less with a user's normal browsing of web pages, as the user could still use the other parts of the web page, which are not affected by the injected script. This is also a huge advantage in the case of a false-positive, again as the user gets less interrupted, as only a subpart of the page gets blocked.

Blocking whole page There are, however, some disadvantages when choosing to only block parts of the page. When the filter detects an attack, it is not unexpected that an attacker might have combined several techniques and parts when injecting the script into the website, hoping that one of the included parts of the script would be able to circumvent the filter. Hopefully, the filter would be advanced enough to properly detect and block all the parts of the injection, but it might be some special conditions that the filter does not account for, leading to a successful attack. This is one of the reasons why it might be a better approach, when only concerned with security, to block the whole web page from loading when an attack is detected, as the detected attack might just be part of a bigger attack. Another possibility for an attacker is to trick the filter to not block an injected script, but to block some important security feature that are actually needed by the attacked website itself. An example is a website that requires the JavaScript file `security.js` for its security features to work, which which will be included in the response when requesting the website. Since the filter compares script content from the request with script from the response, an attacker might inject a script containing the same filename, `security.js`, which would then be detected by the filter as an attack, as the file is both in the request and the response of the website. This would then disable the websites security features, which means the attacker could create an injection that combines the file `security.js` with some other malicious script executing an attack. Since the filter actually detected an attack, it would be better to block the entire web page from loading, as this would prevent this issue altogether.

5.3.3 Usability

The implemented filter does not currently support any interaction from or with the user of Firefox, which is something that should be considered, as more control of and information about the filter's behavior could be beneficial to websites and Firefox's users.

Choosing blocking technique As there are clearly advantages and disadvantages with both the blocking techniques, it is possible to make this a decision for websites to take, by utilizing the `X-XSS-Protection` HTTP response header [54]. This is a header currently supported by most of the major web browsers, Chrome, Safari, Internet Explorer and Edge, and makes it possible to decide how the browser should act when they detect XSS attacks. There are four possible values for the X-XSS-Protection header. Setting it to `0` will disable the filter and `1` will enable it and only remove the dangerous parts. By using `1; mode=block`, the filter will be enabled and the whole web page will be blocked. A last option is using `1; report=<reporting-uri>`, which will only remove the dangerous parts and use a feature from CSP where the violation is reported and sent to the specified URL.

Violation feedback Another functionality missing from the implementation, something the implemented CSP feature already has, is the ability to properly notify users of a violation. In the case of a XSS violation, this would be when the filter detects and/or blocks the attack, depending on what the previous mentioned X-XSS-Protection header is set to. This header, did as mentioned above, support a reporting feature, where a details of the violation would get sent to a specified URL. However, a violation notice should also be indicated to the users of Firefox, regardless of the reporting feature of the X-XSS-Protection header. In the filter's current state, these violation details are only shown in a special console meant for the developers of Firefox itself, and not in the developer console accessible to normal users of Firefox. The details shown to the users does not have to contain every detail about the violation, but an indication of what has happened should be displayed.

6 Conclusion

Cross-Site Scripting (XSS) vulnerabilities continues to be one of the most critical web security threats among todays web applications, despite of the large quantity of research, proposals and solutions being published and implemented [10]. This is a type of vulnerability that mainly and directly compromise the end-users of web applications, which means they need additional protection. All of the major web browsers have taken action by implementing several protection mechanisms defending against these XSS attacks. Since XSS is such a complex vulnerability, there is not a single protection mechanism that will stop all of the attacks, but rather a strategy of having several mechanisms that together provide the best protection, utilizing the defense-in-depth strategy. All of the major web browsers have included a built-in filter for XSS protection as one of these counter measures, except the second most used, Mozilla Firefox, which have neglected to include such feature. As seen from the lacking of effectiveness of the most comprehensive protection mechanism in Firefox, CSP, as discussed Chapter 2.2.6, the need for a built-in XSS filter in Firefox is evident, considering the prevalence and consequence of these attacks.

This thesis has made a proposal and implementation for such a filter, which is built-in and integrated into Firefox. The filtering principles for the filter was based on the filter used in Google's Chrome browser, XSS Auditor, which utilizes an advantageous placement inside the web browser, achieving both good protection and high performance. After doing several tests of the implemented filter, findings suggest that the filter did perform very well in protecting against a wide variety of script injections, which contained different attack vectors utilizing several methods trying to circumvent the filtering mechanism. Adding and removing characters, using URL encodings and different on-event handlers were efficiently blocked by the filter. There were, however, some limitations regarding different types of encodings and a lack of support for some attack vectors, which are something that needs to be added before the filter could be considered sufficient for every-day usage. Performance-wise, the filter did not show any measurable difference compared to the version of Firefox without the filter. By not having any huge performance overhead means that adding small additions for fixing the limitations mentioned should not incur significantly more overhead, as the most demanding filtering mechanisms are already implemented.

The modified version of Firefox containing the filter do, therefore, already provide much better protection than the original version of Firefox. Even though there are limitations that needs to be addressed for it to be a considered a fully fledged solution, it already serves as an important layer in the defense-in-depth strategy, providing a little extra to the much desired protection that is needed for XSS vulnerabilities.

7 Further Work

As discussed in Chapter 5, the implemented filter still has room for improvements considering its protection effectiveness. The areas for improvements are regarding input sources, attack vectors, support for more encodings and integration with existing Firefox code. Most of these improvements should be rather trivial to implement. Firefox's internal code has easy access to other input sources data, like the most relevant, which are HTTP headers. In the case of attack vector support, the already supported attack vectors only needed about two lines of code for them to be covered, so it should be as trivial to add support to other vectors, like the `iframe`, `embed`, `svg` and `object` tags, as mentioned in Chapter 5.1.2. The only challenge with these is to identify the location inside the Firefox code where they are being processed, as they might be handled in vastly different areas in the code. Support for more encodings should also not be too difficult to achieve, as there exist good documentation covering how different encodings work, and the fact that the filter class is structured in such a way that it is easy to add more advanced filtering rules. The most challenging task would be to better integrate the filter into the existing Firefox code, to comply with all the security principals and coding standards that are required by Mozilla. Another improvement could be to implement support for the X-XSS-Protection header, which would let websites themselves decide if they want to use it or not.

8 References

[1] 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.

[2] B. Stock, S. Lekies, T. Mueller, P. Spiegel, and M. Johns, “Precise client-side protection against dom-based cross-site scripting.” in USENIX Security Symposium, 2014, pp. 655–670.

[3] 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, pp. 512–530, 2017.

[4] M. Alvarez, N. Bradley, P. Cobb, S. Craig, R. Iffert, L. Kessem, J. Kravitz, D. McMillen, and S. Moore, “IBM X-Force Threat Intelligence Index 2017 The Year of the Mega Breach,” IBM Security,(March), pp. 1–30, 2017.

[5] Mozilla Developer Network, “Script Security,” Aug 2016, accessed 2018-05-24. [Online]. Available: https://developer.mozilla.org/en-US/docs/Mozilla/Gecko/Script_security

[6] ——, “Content security policy (csp),” January 2018, accessed: 2018-01-24. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP

[7] 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_%28en%29.pdf.pdf

[8] 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%202017%20Application%20Security%20Report%20FINAL.pdf

[9] 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

[10] I. Hydara, A. B. M. Sultan, H. Zulzalil, and N. Admodisastro, “Current state of research on cross-site scripting (XSS)–A systematic literature review,” Information and Software Technology, vol. 58, pp. 170–186, 2015.

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

[12] A. Vikne and P. Ellingsen, “Client-Side XSS Filtering in Firefox,” in SOFTENG 2018, The Fourth International Conference on Advances and Trends in Software Engineering, April 2018, pp. 24–29.

[13] Mozilla Developer Network, “Confidentiality, Integrity, and Availability,” April 2018, accessed: 2018-04-19. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/Security/Information_Security_Basics/Confidentiality,_Integrity,_and_Availability

[14] ——, “HTTP headers,” April 2018, accessed 2018-05-13. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers

[15] ——, “Referer,” June 2017, accessed 2018-05-13. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Referer

[16] ——, “User-Agent,” June 2017, accessed 2018-05-13. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/User-Agent

[17] S. Di Paola and G. Fedon, “Subverting ajax,” 2006.

[18] Facebook Inc., “Httponly,” August 2017, accessed: 2018-03-23. [Online]. Available: https://www.facebook.com/help/246962205475854

[19] Mozilla Developer Network, “DOM on-event handlers,” Jan 2018, accessed 2018-05-24. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/Guide/Events/Event_handlers

[20] 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

[21] ——, “Xss (cross site scripting) prevention cheat sheet,” October 2017, accessed: 2018-01-24. [Online]. Available: https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet

[22] 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/

[23] L. Weichselbaum, M. Spagnuolo, S. Lekies, and A. Janc, “Csp is dead, long live csp! on the insecurity of whitelists and the future of content security policy,” in Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. ACM, 2016, pp. 1376–1387.

[24] Mozilla Developer Network, “Same-origin policy,” March 2018, accessed: 2018-03-21. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy

[25] OWASP Foundation, “Httponly,” August 2017, accessed: 2018-03-21. [Online]. Available: https://www.owasp.org/index.php/HttpOnly

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

[27] Mozilla Developer Network, “eval,” April 2018, accessed 2018-05-29. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/eval

[28] StatCounter, “Desktop browser market share worldwide,” May 2018, accessed: 2018-05-09. [Online]. Available: http://gs.statcounter.com/browser-market-share/desktop/worldwide

[29] 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/

[30] Mozilla Developer Network, “History of the Mozilla Project,” April 2018, accessed: 2018-04-04. [Online]. Available: https://www.mozilla.org/en-US/about/history/details

[31] ——, “An introduction to hacking mozilla,” March 2017, accessed: 2017-12-28. [Online]. Available: https://developer.mozilla.org/en-US/docs/Mozilla/An_introduction_to_hacking_Mozilla

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

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

[34] ——, “How mozilla’s build system works,” December 2017, accessed: 2018-02-14. [Online]. Available: https://developer.mozilla.org/en-US/docs/Mozilla/Developer_guide/Build_Instructions/How_Mozilla_s_build_system_works

[35] GNU/Free Software Foundation, “Gnu make,” May 2016, accessed: 2018-02-14. [Online]. Available: https://www.gnu.org/software/make/

[36] Mozilla Developer Network, “mach,” December 2017, accessed: 2018-02-14. [Online]. Available: https://developer.mozilla.org/en-US/docs/Mozilla/Developer_guide/mach

[37] I. Black Duck Software, “The Mozilla Firefox Open Source Project on Open Hub: Languages Page,” April 2018, accessed 2018-05-22. [Online]. Available: https://www.openhub.net/p/firefox/analyses/latest/languages_summary

[38] Mozilla Developer Network, “Mozilla Source Code Directory Structure,” Jan 2018, accessed 2018-05-22. [Online]. Available: https://developer.mozilla.org/en-US/docs/Mozilla/Developer_guide/Source_Code/Directory_structure

[39] ——, "HTML parser threading," March 2013, accessed 2018-05-30. [Online]. Available: https://developer.mozilla.org/en-US/docs/Mozilla/Gecko/HTML_parser_threading

[40] ——, "Browser Extensions," March 2018, accessed 2018-05-27. [Online]. Available: https://developer.mozilla.org/en-US/Add-ons/WebExtensions

[41] ——, "Browser support for JavaScript APIs," May 2018, accessed 2018-05-27. [Online]. Available: https://developer.mozilla.org/en-US/Add-ons/WebExtensions/Browser_support_for_JavaScript_APIs

[42] "Arch Linux," Jan 2018, accessed 2018-05-30. [Online]. Available: https://wiki.archlinux.org/index.php/Arch_Linux

[43] Microsoft, "Visual Studio Code - Code Editing. Redefined," accessed 2018-05-30. [Online]. Available: https://code.visualstudio.com/

[44] Mozilla Developer Network, "Configuring Build Options," March 2018, accessed 2018-05-30. [Online]. Available: https://developer.mozilla.org/en-US/docs/Mozilla/Developer_guide/Build_Instructions/Configuring_Build_Options

[45] GNU/Free Software Foundation, "Gdb: The gnu project debugger," February 2018, accessed: 2018-03-02. [Online]. Available: https://www.gnu.org/software/gdb/

[46] R. D. W3Schools, "HTML URL Encoding Reference," May 2018, accessed: 2018-05-02]. [Online]. Available: https://www.w3schools.com/tags/ref_urlencode.asp

[47] S. Project, "Selenium WebDriver - Selenium Documentation," April 2018, accessed 2018-05-02. [Online]. Available: https://www.seleniumhq.org/docs/03_webdriver.jsp

[48] Balaji N., "Top 500 most important xss script cheat sheet for web application penetration testing," May 2018, accessed 2018-05-30. [Online]. Available: https://gbhackers.com/top-500-important-xss-cheat-sheet/

[49] Thapa, Prabesh, “Xss-payloads,” Aug 2016, accessed 2018-05-30. [Online]. Available: https://github.com/Pgaijin66/XSS-Payloads/blob/master/payload.txt

[50] FuzzDB Project, “xss-other.txt,” Oct 2016, accessed 2018-05-30. [Online]. Available: https://github.com/fuzzdb-project/fuzzdb/blob/master/attack/xss/xss-other.txt

[51] ——, “xss-rsnake.txt,” May 2016, accessed 2018-05-30. [Online]. Available: https://github.com/fuzzdb-project/fuzzdb/blob/master/attack/xss/xss-rsnake.txt

[52] 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/

[53] Mozilla Developer Network, “Mozilla internal string guide,” April 2018, accessed 2018-05-25. [Online]. Available: https://developer.mozilla.org/en-US/docs/Mozilla/Tech/XPCOM/Guide/Internal_strings

[54] ——, “X-XSS-Protection,” Feb 2018, accessed 2018-05-28. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-XSS-Protection