**INTRODUCTION**

The aim of this study is improve knowledge about Cross Site Scripting Vulnerability.

**DETAILS**

**What is Cross Site Scripting (XSS) Vulnerability?**

Cross-site scripting (also known as XSS) is a web security vulnerability that allows an attacker to compromise the interactions that users have with a vulnerable application. It allows an attacker to circumvent the same origin policy, which is designed to segregate different websites from each other. Cross-site scripting vulnerabilities normally allow an attacker to masquerade as a victim user, to carry out any actions that the user is able to perform, and to access any of the user's data. If the victim user has privileged access within the application, then the attacker might be able to gain full control over all of the application's functionality and data.

**How does XSS work?**

Cross-site scripting works by manipulating a vulnerable web site so that it returns malicious JavaScript to users. When the malicious code executes inside a victim's browser, the attacker can fully compromise their interaction with the application.

Diagram

Description automatically generated

**XSS Proof Of Concept**

You can confirm most kinds of XSS vulnerability by injecting a payload that causes your own browser to execute some arbitrary JavaScript. It's long been common practice to use the alert() function for this purpose because it's short, harmless, and pretty hard to miss when it's successfully called. In fact, you solve the majority of our XSS labs by invoking alert() in a simulated victim's browser.

Unfortunately, there's a slight hitch if you use Chrome. From version 92 onward (July 20th, 2021), cross-origin iframes are prevented from calling alert(). As these are used to construct some of the more advanced XSS attacks, you'll sometimes need to use an alternative PoC payload. In this scenario, we recommend the print() function.

As the simulated victim in our labs uses Chrome, we've amended the affected labs so that they can also be solved using print(). We've indicated this in the instructions wherever relevant.

**What are the types of XSS attacks?**

* **Reflected XSS**, where the malicious script comes from the current HTTP request.
* **Stored XSS**, where the malicious script comes from the website's database.
* **DOM-based XSS**, where the vulnerability exists in client-side code rather than server-side code.

**Reflected XSS**

**Reflected XSS**, is the simplest variety of cross-site scripting. It arises when an application receives data in an HTTP request and includes that data within the immediate response in an unsafe way.

Here is a simple example of a **Reflected XSS** vulnerability:

https://insecure-website.com/status?message=All+is+well.

<p>Status: All is well.</p>

The application doesn't perform any other processing of the data, so an attacker can easily construct an attack like this:

https://insecure-website.com/status?message=<script>/\*+Bad+stuff+here...+\*/</script>

<p>Status: <script>/\* Bad stuff here... \*/</script></p>

If the user visits the URL constructed by the attacker, then the attacker's script executes in the user's browser, in the context of that user's session with the application. At that point, the script can carry out any action, and retrieve any data, to which the user has access.

**Impact Of Reflected XSS Attacks**

f an attacker can control a script that is executed in the victim's browser, then they can typically fully compromise that user. Amongst other things, the attacker can:

* Perform any action that user can perform in that Web Application
* View any information that victim can view
* Modify any information that the victim is able to modify
* Initiate interactions with other application users, including malicious attacks, that will appear to be comes from the initial victim user.

There are various ways which an attacker might trick a victim user to make a request that they control, to deliver a reflected XSS attack. These include placing links on a website controlled by the attacker, or on another website that allows content to be generated, or by sending a link in an email, tweet or other message. The attack could be targeted directly against a known user, or could be an indiscriminate attack against any users of the application.

The need for an external delivery mechanism for the attack means that the impact of reflected XSS is generally less severe than stored XSS, where a self-contained attack can be delivered within the vulnerable application itself.

**Stored XSS**

Stored cross-site scripting (also known as second-order or persistent XSS) arises when an application receives data from an untrusted source and includes that data within its later HTTP responses in an unsafe way.

Suppose a website allows users to submit comments on blog posts, which are displayed to other users. Users submit comments using an HTTP request like the following:

POST /post/comment HTTP/1.1

Host: vulnerable-website.com

Content-Length: 100

postId=3&comment=This+post+was+extremely+helpful.&name=Carlos+Montoya&email=carlos%40normal-user.net

After this comment has been submitted, any user who visits the blog post will receive the following within the application's response:

<p>This post was extremely helpful.</p>

Assuming the application doesn't perform any other processing of the data, an attacker can submit a malicious comment like this:

<script>/\* Bad stuff here... \*/</script>

Within the attacker's request, this comment would be URL-encoded as:

comment=%3Cscript%3E%2F\*%2BBad%2Bstuff%2Bhere...%2B\*%2F%3C%2Fscript%3E

Any user who visits the blog post will now receive the following within the application's response:

<p><script>/\* Bad stuff here... \*/</script></p>

The script supplied by the attacker will then execute in the victim user's browser, in the context of their session with the application.

**Impact Of Stored XSS Attacks**

If an attacker can control a script that is executed in the victim's browser, then they can typically fully compromise that user. The attacker can carry out any of the actions that are applicable to the impact of **Reflected XSS Vulnerabilities**.

In terms of exploitability, the key difference between reflected and stored XSS is that a stored XSS vulnerability enables attacks that are self-contained within the application itself. The attacker does not need to find an external way of inducing other users to make a particular request containing their exploit. Rather, the attacker places their exploit into the application itself and simply waits for users to encounter it.

The self-contained nature of stored cross-site scripting exploits is particularly relevant in situations where an XSS vulnerability only affects users who are currently logged in to the application. If the XSS is reflected, then the attack must be fortuitously timed: a user who is induced to make the attacker's request at a time when they are not logged in will not be compromised. In contrast, if the XSS is stored, then the user is guaranteed to be logged in at the time they encounter the exploit.

**DOM-Based Cross-Site Scripting**

DOM-based XSS vulnerabilities usually arise when JavaScript takes data from an attacker-controllable source, such as the URL or an input field, and passes it to a sink that supports dynamic code execution, such as eval() or innerHTML. This enables attackers to execute malicious JavaScript, which typically allows them to hijack other users' accounts.

To deliver a DOM-based XSS attack, you need to place data into a source so that it is propagated to a sink and causes execution of arbitrary JavaScript.

The most common source for DOM XSS is the URL, which is typically accessed with the **window.location** object. An attacker can construct a link to send a victim to a vulnerable page with a payload in the query string and fragment portions of the URL. In certain circumstances, such as when targeting a 404 page or a website running PHP, the payload can also be placed in the path.

**How To Test For DOM-Based Cross-Site Scripting**

**Testing HTML Sinks**

To test for DOM XSS in an HTML sink, place a random alphanumeric string into the source (such as location.search), then use developer tools to inspect the HTML which contain our string and find where our string appears. We need to note that, the browser’s “View Source” option won’t work for DOM XSS testing because it doesnt change things that have been performed in the HTML by JavaScript. In developer tools, we can find our string in DOM by using “Ctrl + f”.

For each location where our string appears within the DOM, we need to identify the context. Based on this context, we need to refine our input to see how it is processed. **For example, if our string appears within a double-quoted attribute, then try to inject double quotes in our string to see if we can break out of the attribute.**

Note that browsers behave differently with regards to URL-encoding. Chrome, Firefox, Safari will URL-encode *location.search* and *location.hash*. While IE11 and Microsoft Edge (pre-Chromium) will not URL-encode these sources. If our data gets URL-encoded before being processed, then an XSS attack is unlikely to work.

**Testing JavaScript Execution Sinks**

Testing JavaScript execution sinks for DOM-based XSS is a little harder. With these sinks, your input doesn’t necessarily appear anywhere within the DOM, so you can’t search for it. Instead we’ll need to use the JavaScript debugger to determine whether and how our input is sent to a sink.

For each potential source such as *location*, we need to find cases within the page’s JavaScript code where the source is being referenced. In Chrome’s developer tools, we can use Control+Shift+F to search all the page’s JavaScript code for the source.

Once we’ve found where the source is being read, we can use the JavaScript debugger to add a break point and follow how the source’s value is used. We might find that the source gets assigned to other variables. If this is the case, we’ll need to use the search function again to track these variables and see if they're passed to a sink. When we find a sink that is being assigned data that originated from the source, you can use the debugger to inspect the value by hovering over the variable to show its value before it is sent to the sink. Then, as with HTML sinks, you need to refine your input to see if you can deliver a successful XSS attack.

**Testing For DOM XSS Using DOM Invader**

This tool helps us to detect DOM-based XSS vulnerabilities. To use the tool, Open burp browser->Click burp icon at top right of the browser->Select Dom invader panel-> Change options according to our need

I am using these settings: (these settings can be customized according to the needs)

(Video Link about Dom Invader: <https://www.youtube.com/watch?v=Wd2R47czzO0> )

Graphical user interface, text, application, chat or text message

Description automatically generated

**Exploiting DOM XSS With Different Sources And Sinks**

In principle, a website is vulnerable to DOM-based XSS if there is an executable path via which data can propagate from source to sink. In practice, different sources and sinks have differing properties and behavior that can affect exploitablity, and determine what techniques are necessary. Additionally, the website’s scripts might perform validation or other processing of data that must be accomodated when attempting to exploit a vulnerability. There are a variety of sinks that are relevant to DOM-based vulnerabilities. The list of them is:

Graphical user interface, text, application, email

Description automatically generated

The *document.write* sink Works with *script*  elements, so on that sink for example we can use this simple payload:

document.write('... <script>alert(document.domain)</script> ...');

**LAB1-) DOM XSS in document.write sink using source**

This lab contains a DOM-based cross-site scripting vulnerability in the search query tracking functionality. It uses the JavaScript document.write function, which writes data out to the page. The document.write function is called with data from location.search, which you can control using the website URL.

To solve this lab, perform a cross-site scripting attack that calls the alert function.

1-) There is a search input field on the main page of the lab. So I try to enter “test” string. Then I looked for page’s html codes to see what does this web app do with input

Graphical user interface, text, application

Description automatically generated

2-) As we can see, application takes the input and put it between the **“<script>”** tags inside the <img src> function. So I need to break out from the **<img>** tag or the **<script>** tag. I preferred to break out the <img> tag.

3-) So First, I closed double quotes to get out from the string part and then I closed <img> tag with “**>**”. My payload was this : "><svg onload=alert(1)> . (This lab can also completed with </script><script>alert(1)</script> payload. This payload is escaped from script tag.)

4-) Then the lab is completed.

Graphical user interface, application, Teams

Description automatically generated

We need to note that, in some situations the content that is written to *document.write* includes some surrounding context that you need to take account of in your exploit. For example, you might need to close some existing elements before using our JavaScript payload.

**LAB 2-) DOM XSS in document.write sink using source location.search inside a select element**

This lab contains a DOM-based cross-site scripting vulnerability in the stock checker functionality. It uses the JavaScript document.write function, which writes data out to the page. The document.write function is called with data from location.search which you can control using the website URL. The data is enclosed within a select element. To solve this lab, perform a cross-site scripting attack that breaks out of the select element and calls the alert function.

1-) At first glance, there isn’t any input field on the lab. So I try to find a usable sink to put my malicious xss payload in it.

2-) I searched for some sinks to put my malicious payload in it. I couldn’t see any sink in home page. So I try to go to one of the products page. And saw document.write() “**sink**” inside of it. There is a storeId parameter is getting processed on document.write function.

3-) As I can see, storeId paremeter is getting value from url. So I put “**&storeId=”test”**” value in the url and looked for html codes of the page.

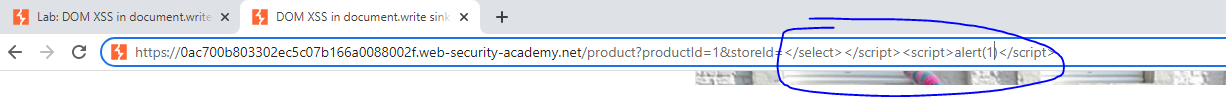


4-) Our test value is added inside the pages html code, so we can break first </select> tag and then </script> tag to inject our malicious payload

Graphical user interface, text, application, email

Description automatically generated

5-) And the malicious payload is something like this:



6-) And the lab is completed

Graphical user interface, text, application

Description automatically generated

The *innerHTML* sink doesn’t accept *script* elements on any modern browser, nor will *svg onload* events fire. This means that we need to use alternative elements like *img* or *iframe*. Event handlers such as *onload* and *onerror* can be used in conjunction with these elements. For example:

element.innerHTML='... <img src=1 onerror=alert(document.domain)> ...'

**LAB 3-) DOM XSS in innerHTML sink using source location.search**

This lab contains a DOM-based cross-site scripting vulnerability in the search blog functionality. It uses an innerHTML assignment, which changes the HTML contents of a **div** element, using data from location.search. To solve this lab, perform a cross-site scripting attack that calls the alert function.

1-) This search functionality uses **div** element on the page at **innerHTML sink**. So, this input field won’t be affected by <script> or <svg onload…> events. Then we need to use alternative elements like <**img**> and **<iframe>.** We can use **onload** and **onerror** event handlers to trigger xss vulnerability. So I inject this malicious payload to the search function of the page:

Graphical user interface, application, Teams

Description automatically generated

2-) And the lab is completed.

Graphical user interface, application

Description automatically generated

**Exploiting Cross-Site Scripting Vulnerabilities**

The traditional way to prove that you've found a XSS vulnerability is to create a popup using the alert() function. This isn't because XSS has anything to do with popups; it's simply a way to prove that you can execute arbitrary JavaScript on a given domain. You might notice some people using alert(document.domain). This is a way of making it obvious which domain the JavaScript is executing on.

Sometimes you'll want to go further and prove that an XSS vulnerability is a real threat by providing a full exploit. In this section, we'll explore three of the most popular and powerful ways to exploit an XSS vulnerability.

**Exploiting Cross-Site Scripting To Steal Cookies**

Stealing cookies is a traditional way to exploit XSS. Most Web App use cookies for session handling. We can exploit XSS vulnerability to send victim’s cookies to our own domain. Then we can manually inject those captured cookie to our browser to impersonate the victim.

But this approach has limitations,

* The victim might not be logged in
* Many application use HttpOnly flag on it’s cookies to prevent JavaScript to reach them
* Sessions might be locked to additional factors like the user’s IP adress
* Session might time out before we able to hijack it.

**REFERENCES**

https://www.youtube.com/watch?v=Wd2R47czzO0