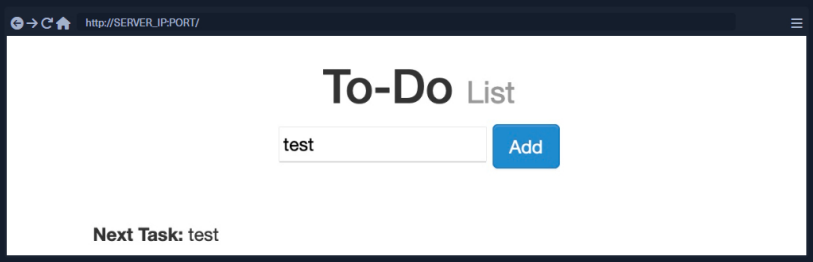
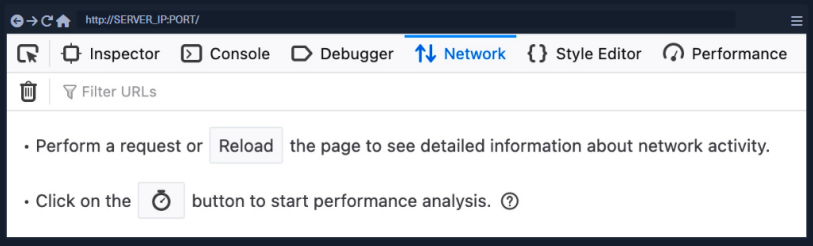
DOM XSS

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:

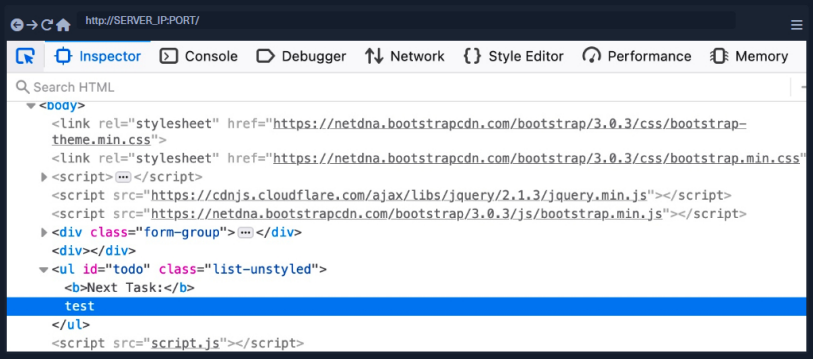


However, if we open the `Network` tab in the Firefox Developer Tools, and re-add the `test` item, we would notice that no HTTP requests are being made:



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]`:



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

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

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:

Code: javascript

```
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:

Code: javascript

```
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.

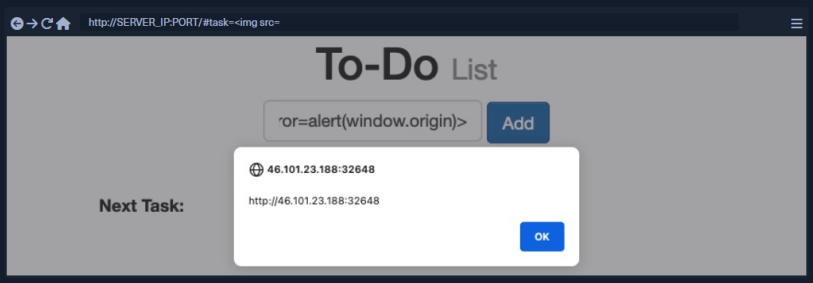
DOM Attacks

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:

Code: html

```
<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. 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.

Connect to Pwnbox
Your own web-based Parrot Linux instance to play our labs.

Pwnbox Location

UK 160ms

Terminate Pwnbox to switch location

Start Instance

∞ / 1 spawns left

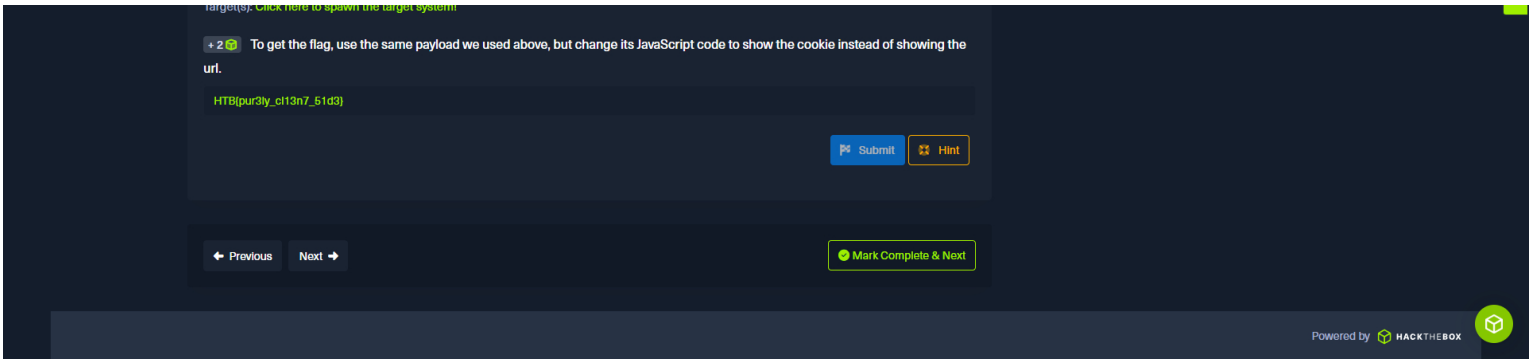
Waiting to start...

Enable step-by-step solutions for all questions

Questions

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

Cheat Sheet


target(s): Click here to spawn the target system!
+ 2 To get the flag, use the same payload we used above, but change its JavaScript code to show the cookie instead of showing the url.
HTB{pur3ly_cl13n7_51d3}
Submit
Hint
Previous
Next
Mark Complete & Next
Powered by HACKTHEBOX