Lab VII – From XSS to Session Hijacking to SQLi Attacks

CPS 499-02/592-02

Software/Language Based Security

Fall 2020

Dr. Phu Phung

Evan Krimpenfort

Part I: From XSS to Session Hijacking Attacks

Task I: Inject the XSS code to steal the victim's cookie

Part a: Construct the XSS code

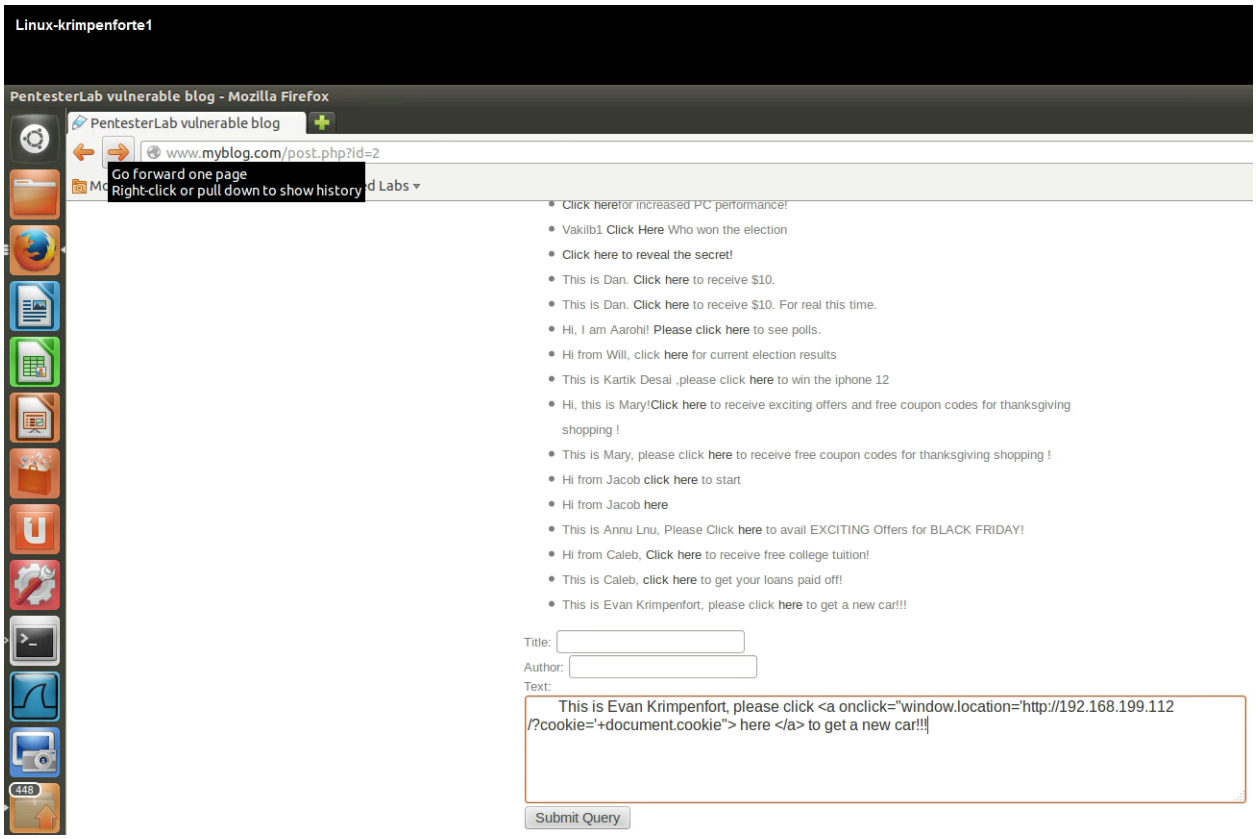

Figure 1: XSS code written before submission

What this code does in Figure 1 is if the person clicks on "here," they will go to the home page of this Linux machine. However, by going to that home page, the cookie of that document will be shown at the top of the screen since it was asked to be retrieved.

Part b: Test the XSS Code:

Part i: Test the Injection

Figure 2: XSS code written after submission

As seen in Figure 2, the code part of the query now embedded into the slightly highlighted "here" statement. In Figure 3, we can see that once "here" is clicked, it will go to the home page of the VM. However, it doesn't show the cookie and that is because right now, the attacker is attacking itself and the information doesn't exist because the attacker isn't logged in.

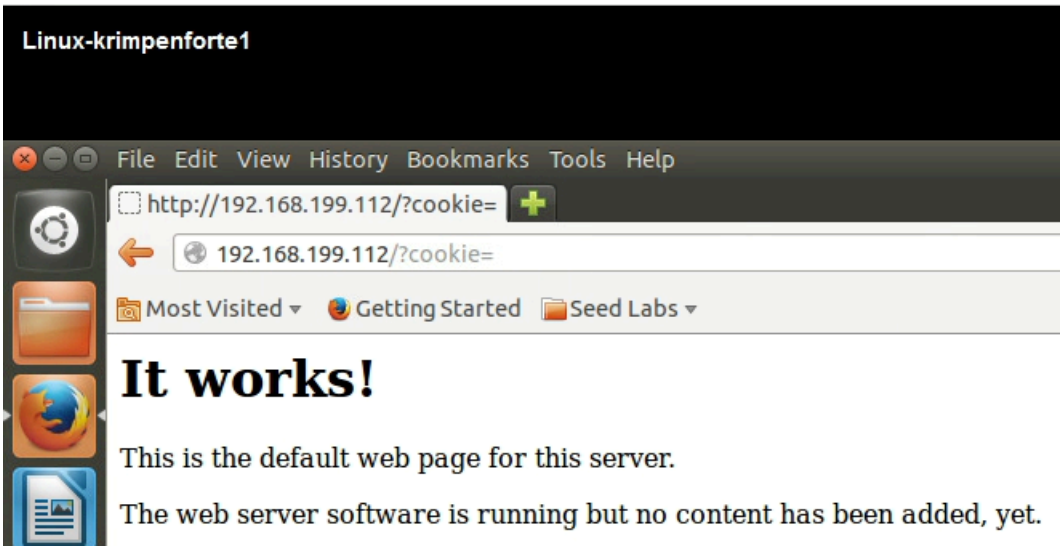

Figure 3: After "here" has been clicked

Part ii: Mark the Request

```
192.168.199.112 - - [09/Nov/2020:23:12:15 -0800] "GET /?cookie= HTTP/1.1" 200
 484 "http://www.myblog.com/post.php?id=2" "Mozilla/5.0 (X11; Ubuntu; Linux i
686; rv:23.0) Gecko/20100101 Firefox/23.0"
```

Figure 4: Looking at the Access Log

The request achieved in this attack was the *"GET /?cookie= HTTP/1.1"* request.

Task II: Simulate the attack as the victim

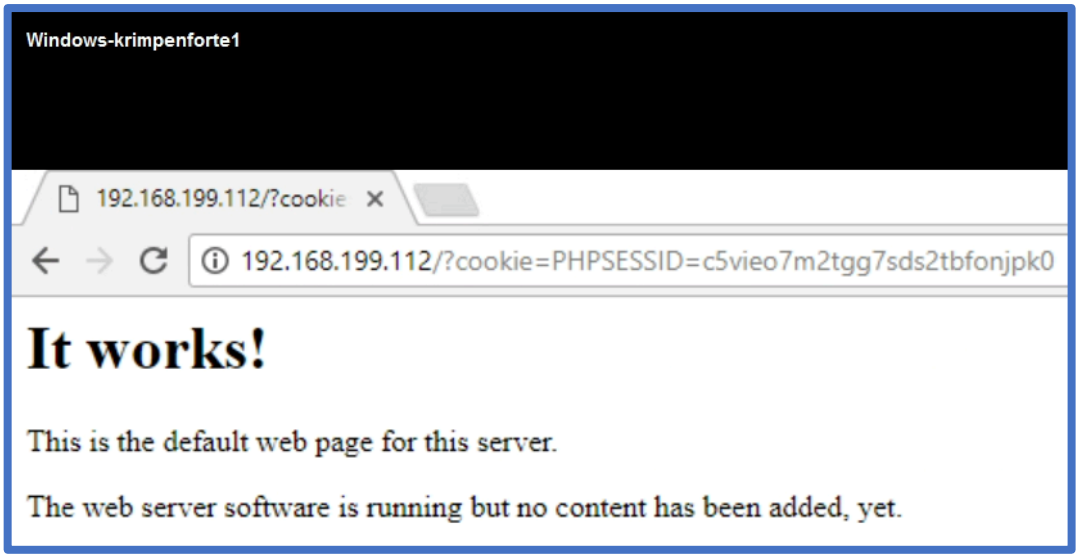

Figure 5: Victim has been attacked

What's happening in Figure 5 is the windows machine user has clicked on the XSS link. Since the Windows user logged in through admin and a cookie was created, the cookie can now be seen in the URL.

Task III: Perform a session hijacking attack

Part a: View the webserver

```
192.168.199.87 - - [10/Nov/2020:09:14:38 -0800] "GET /favicon.ico HTTP/1.1" 404 504 "http://192.1
68.199.112/?cookie=PHPSESSID=c5vieo7m2tgg7sds2tbfonjpk0" "Mozilla/5.0 (Windows NT 10.0; Win64; x6
4) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/61.0.3163.100 Safari/537.36"
```

Figure 6: The log showing the windows user

The information seen here is the HTTP request getting the *document.cookie* information back. You also know this is from the windows machine because you can see it on the second line towards the right.

Part b: Report the steps

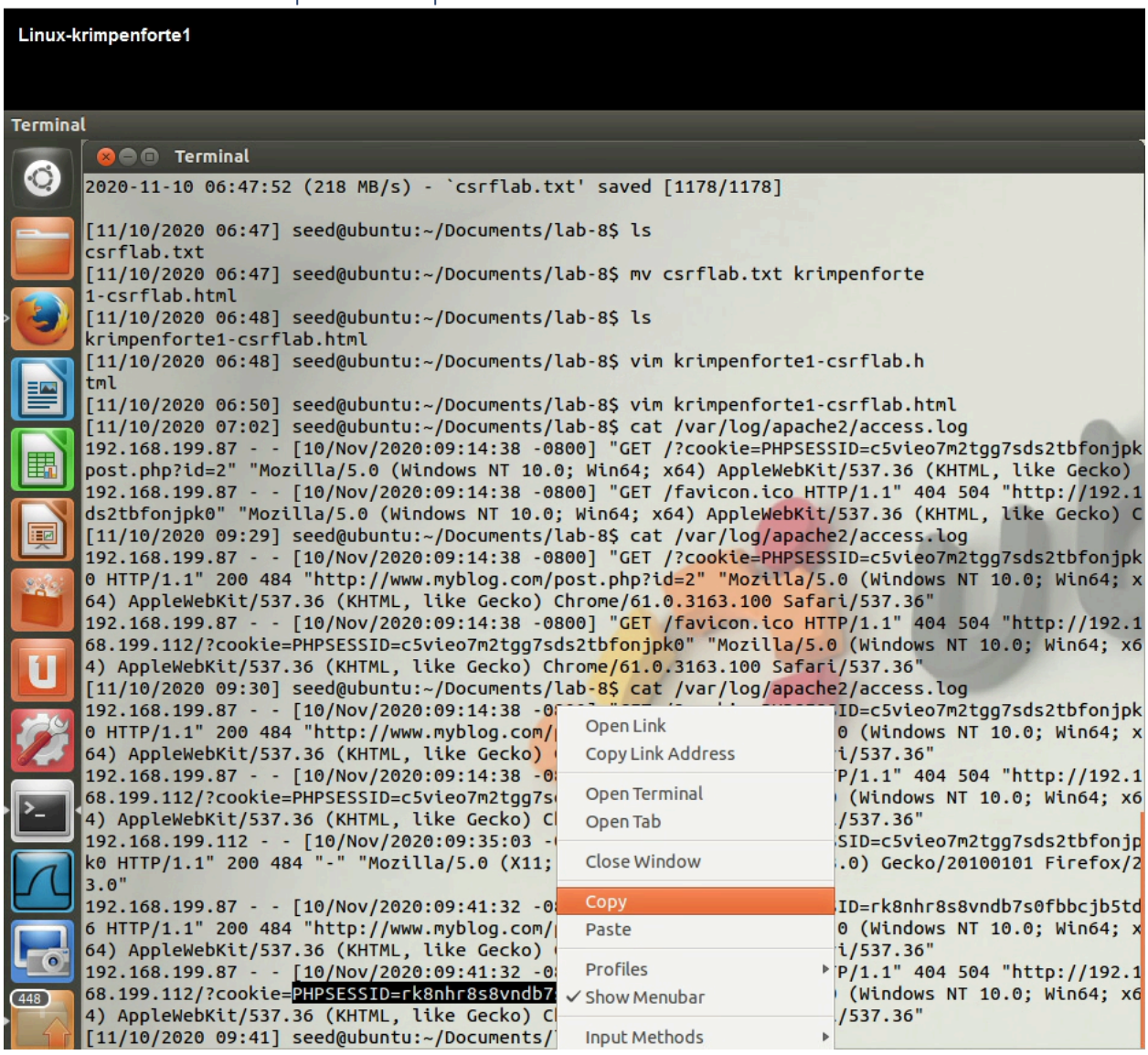

Figure 7: Copying the cookie from the logged in Windows machine

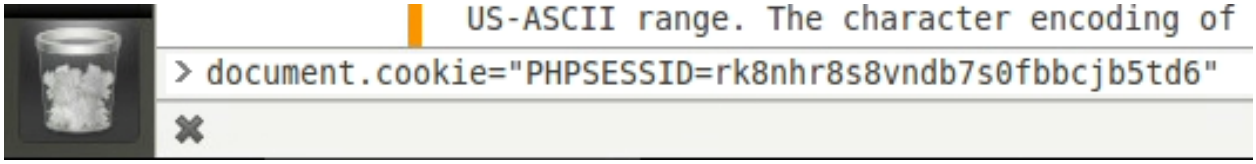

Figure 8: Setting the cookie to the one already logged in

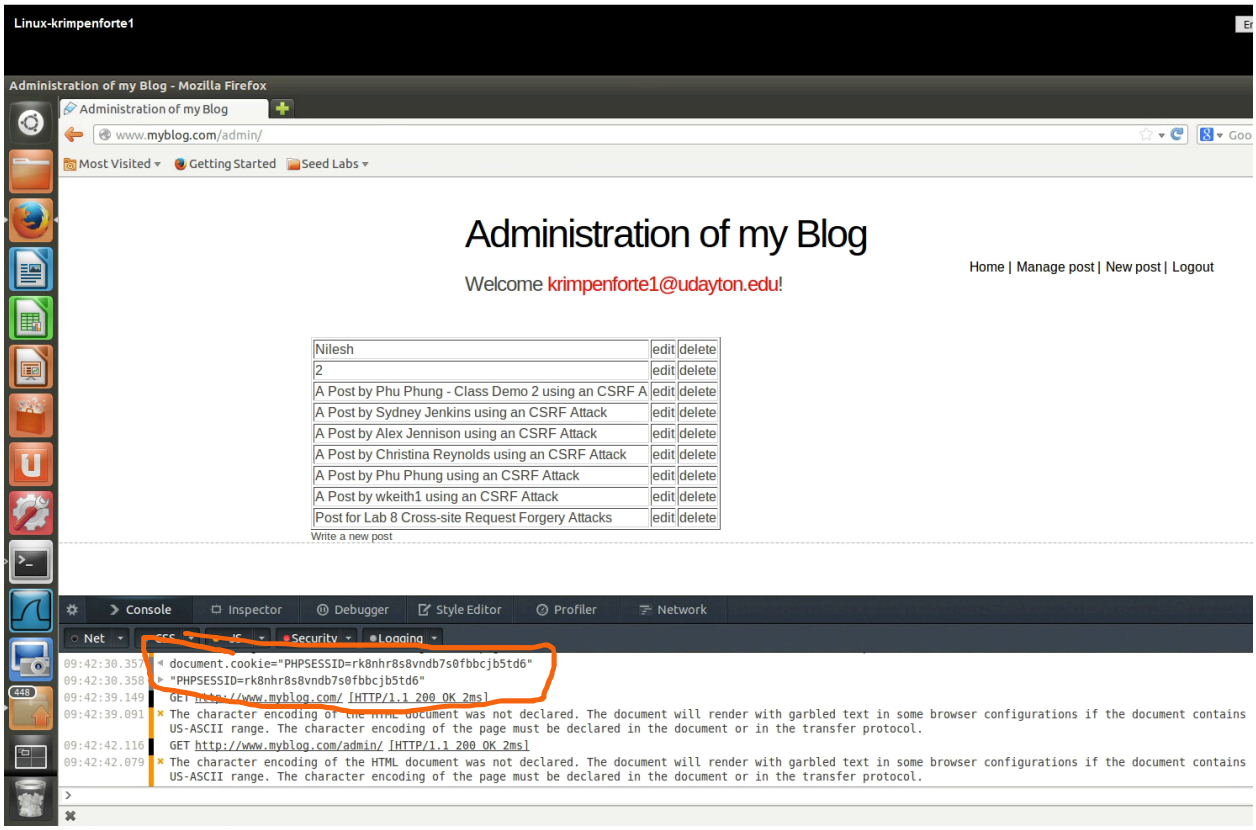

Figure 9: Attack Successful

With the *document.cookie* being overwritten from the stolen Session ID from the logged in windows machine (Figure 8), I was successfully able to get into admin without a username and password In Figure 9.

Part II: From XSS to SQL Injection Attacks

Task IV: Identify SQL Injection Vulnerabilities

Part a: Vulnerability discovery

Without the admin role, this application is not vulnerable to SQLi attacks because when trying to execute SQL code in the URL, the inputs are filtered through and can't be executed properly.

Part b: SQL Injection Attacks

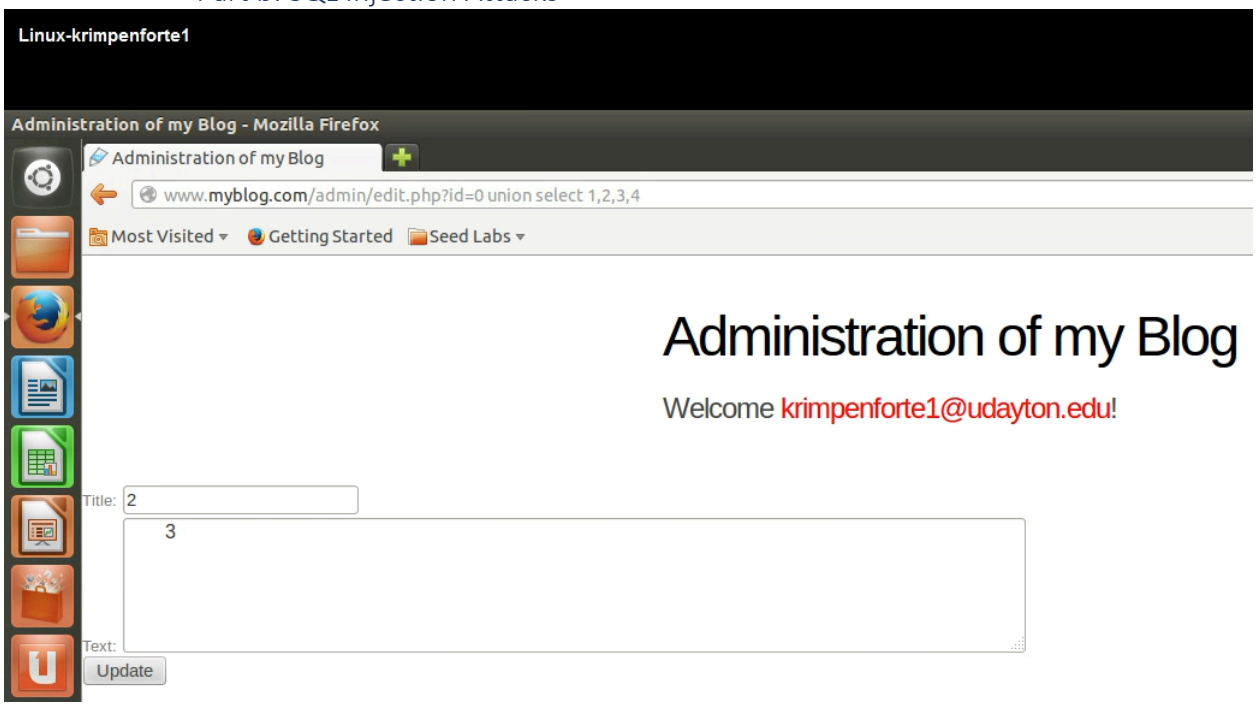

Figure 10: The edit page is vulnerable to SQLi

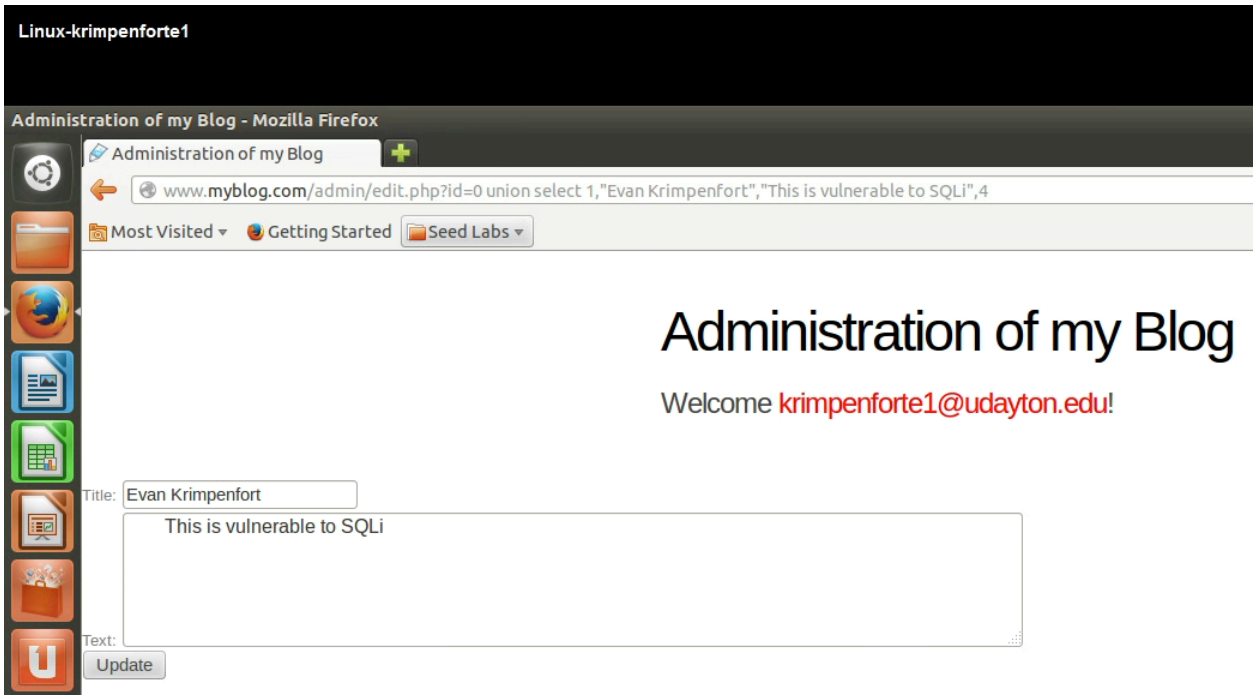

Figure 11: I can add my name and fill the columns

Part c: Continue the same attack

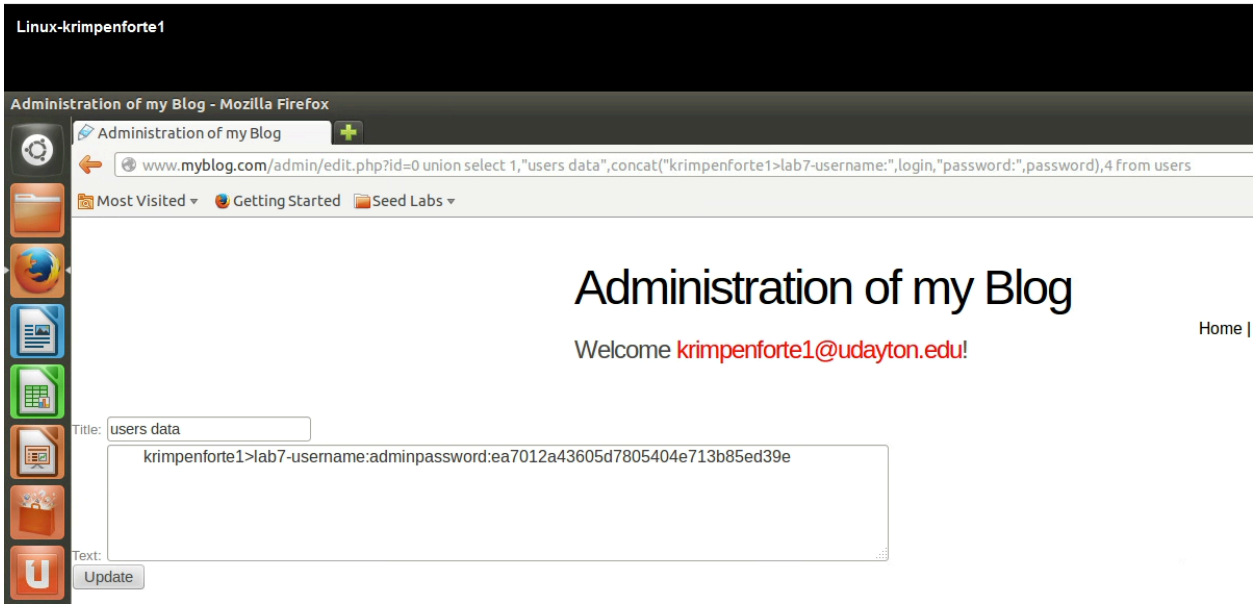

Figure 12: Exploited the username and password of the admin

Similar to how the attack was done in the previous lab, SQLi was used to get the admin username and password.

Task V: SQLi Attacks

Part a: Construct the SQLi query to display the content

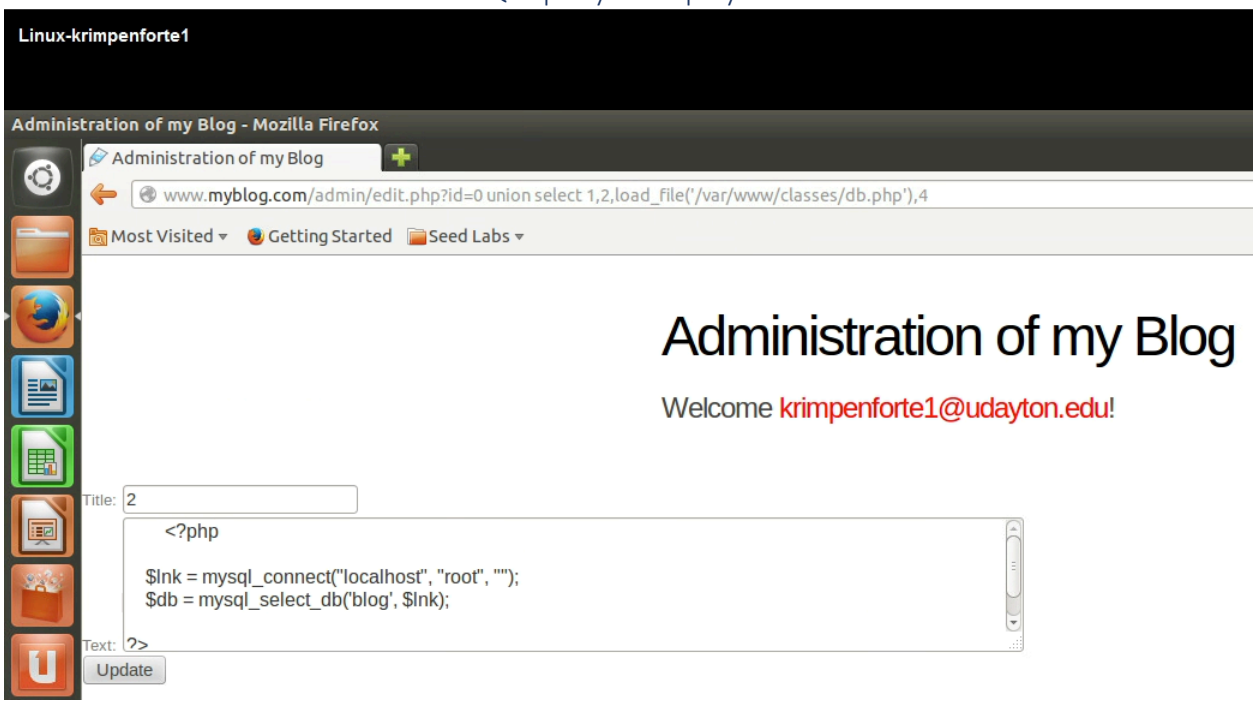

Figure 13: successfully displayed the db.php file

Part b: Construct the SQLi query to create the PHP file

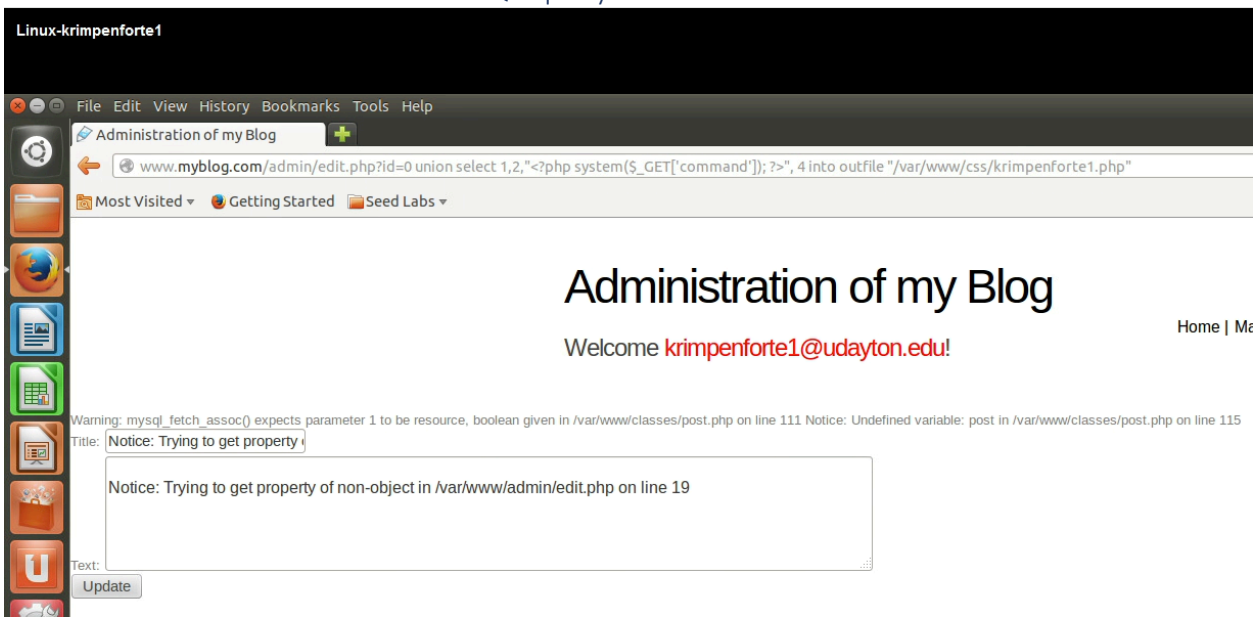

Figure 14: successfully created krimpenforte1.php outfile

Part c: Execute the Injected PHP page

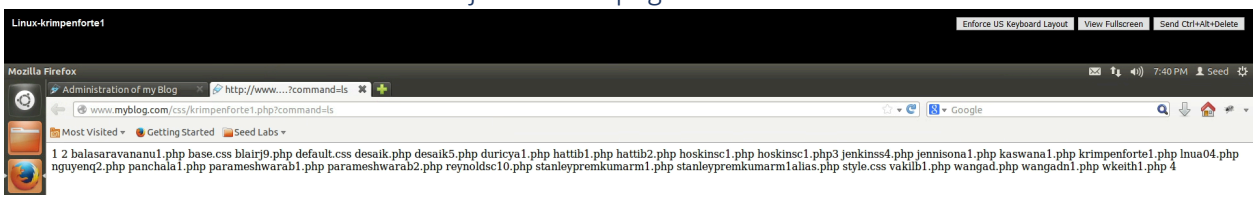

Figure 15: ?command=ls