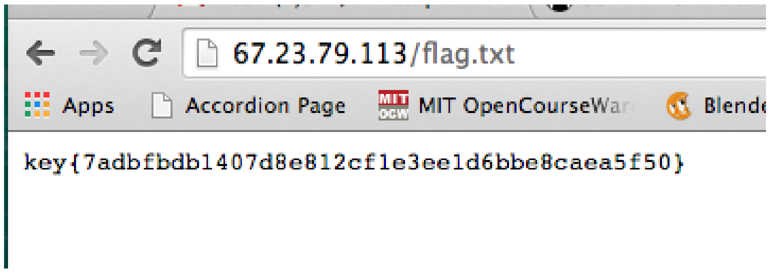
Group 9
Louis Ades, Matthew Ahrens, Jeremy Goldman, MacGill Davis
Security - CTF Writeup

1) Just ask for it

This one took us too long to get, and we actually got three of the harder ones first. We understood that it had to be something on the main blog's page, and that it had to be some sort of GET HTTP request because we were just "asking for it". We found this one after we were done with our initial exploration of the side and just tried different ideas in the url bar. Eventually we just randomly brute-forced a few different url bars, until we found that the correct one was "*/flag.txt*". Others that were attempted include:

- */flag*
- */key*
- */key.txt*
- */canihavetheflag*
- Anything else along those lines

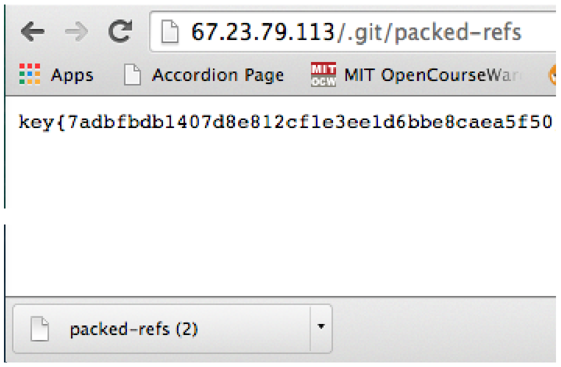
This challenge shows the issue with hiding content by not explicitly linking to it, because attackers can simply traverse your web directory through trial and error to access key data if it's not secured properly.



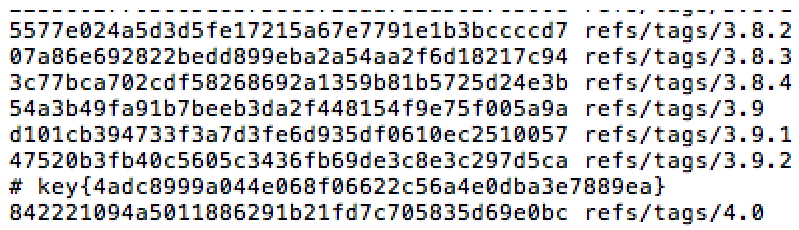
```
key{7adbfbdb1407d8e812cf1e3ee1d6bbe8caea5f50}
```

2) Just ask for it -- but with packed refs

No one on our team knew exactly what the term ‘packed refs’ meant, but we knew it had to do with git. We discussed the dangers of deploying git controlled websites because of what may be stored in the .git directory -- hidden by default to the user. After some quick googling, we realized that the git directory has a packed-refs content, and so even though .git itself was forbidden when we tried it in the url, .git/packed-refs gave us back a keys text file with the key in it.



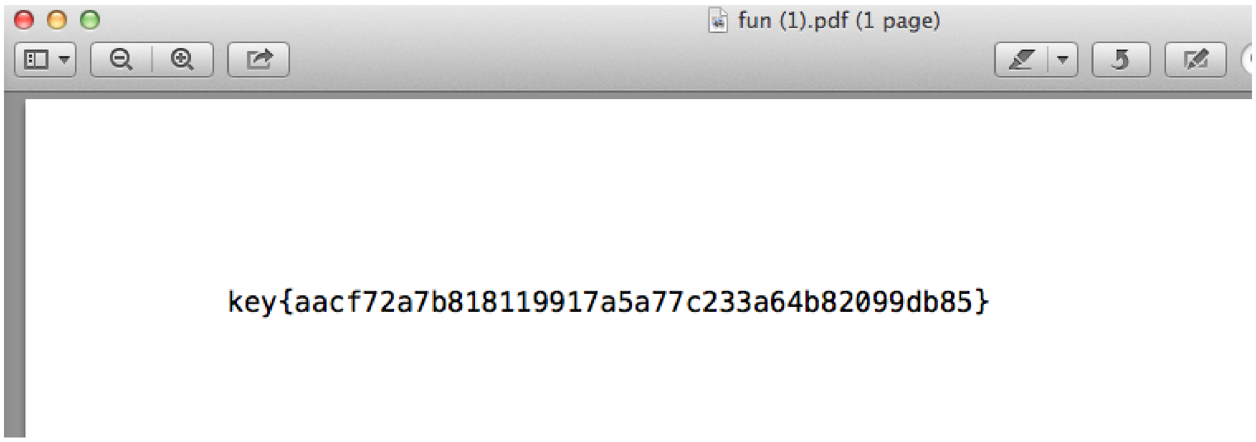
```
key{7adbfbdb1407d8e812cf1e3ee1d6bbe8caea5f50
```



```
5577e024a5d3d5fe17215a67e7791e1b3bccccd7 refs/tags/3.8.2
07a86e692822bedd899eba2a54aa2f6d18217c94 refs/tags/3.8.3
3c77bca702cdf58268692a1359b81b5725d24e3b refs/tags/3.8.4
54a3b49fa91b7beeb3da2f448154f9e75f005a9a refs/tags/3.9
d101cb394733f3a7d3fe6d935df0610ec2510057 refs/tags/3.9.1
47520b3fb40c5605c3436fb69de3c8e3c297d5ca refs/tags/3.9.2
# key{4adc8999a044e068f06622c56a4e0dba3e7889ea}
842221094a5011886291b21fd7c705835d69e0bc refs/tags/4.0
```

3) Analyse the Binary

We didn't do this the easy way that was discussed in class. From the blog post on the main url, we downloaded fun.exe. We discussed whether we should run it, but knew that that wouldn't be it (and also would be quite dangerous!). We renamed the file with no extension in kali, and upon right-clicking it, Kali prompted to open the file in wireshark, letting us know it was a binary pcap. From there, we could use 'follow TCP stream' to look at the contents and see it was a file transfer of a PDF file. We then extracted the PDF file and opened it to see the key. As demonstrated in class several times, this challenge reasserted the openness data can have on a wireless network if not secured properly.



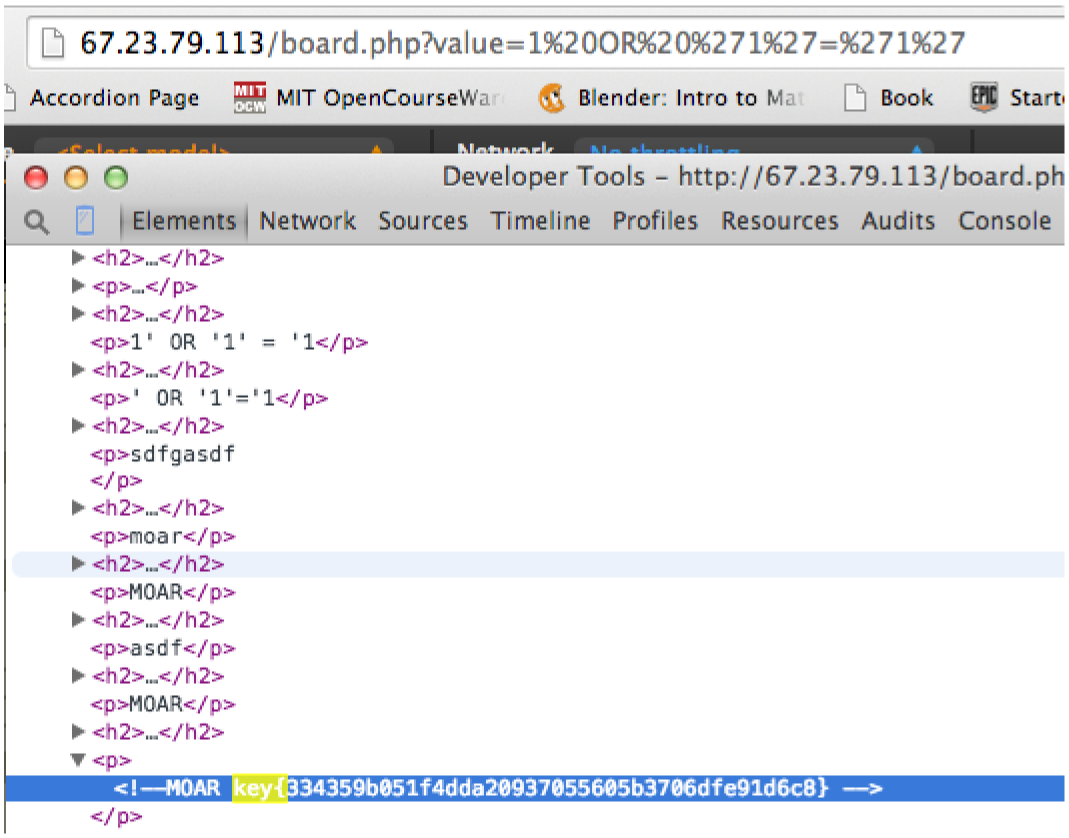
key{aacf72a7b818119917a5a77c233a64b82099db85}

4) I want to see MOAR

On the board, we noticed looking at any particular post had an id field. From inspecting the page and looking at hidden field names, we also knew we could query against the board by looking at value or by the search term ?s=. We used sql injection in the value field to get every post, even the ones hidden or not served by the board by default. The string we used was:

?value=1' OR '1' = '1

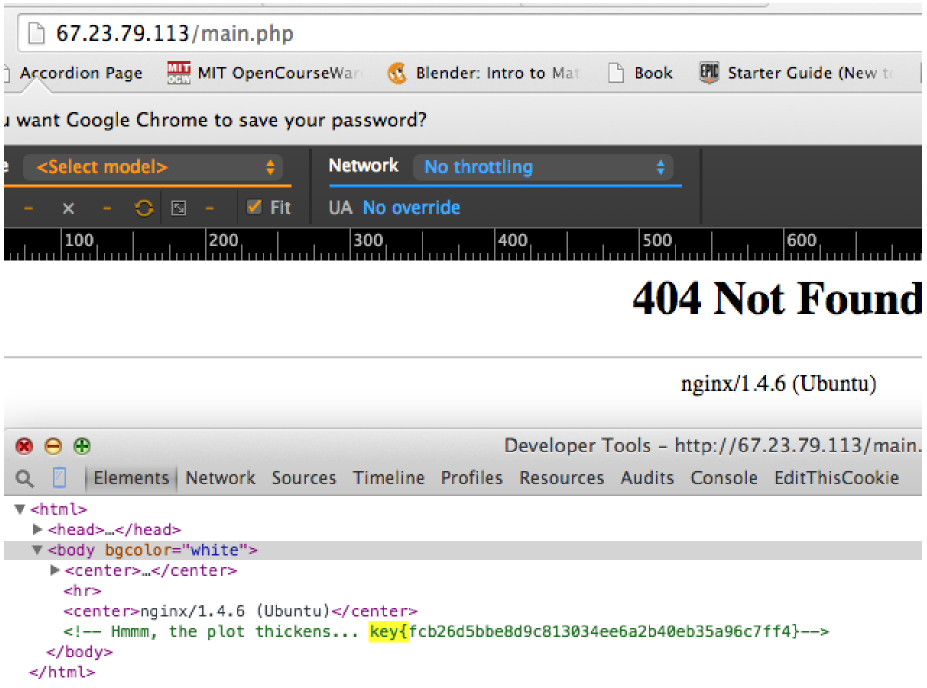
We then inspected the page and searched for MOAR and for "key{" finding the commented section screenshotted below. The major difficulty of this key was that interlopers were trying to sabotage the board by putting fake keys in the posts.



```
▶ <h2>…</h2>
▶ <p>…</p>
▶ <h2>…</h2>
  <p>1' OR '1' = '1</p>
▶ <h2>…</h2>
  <p>' OR '1'='1</p>
▶ <h2>…</h2>
  <p>sdfgasdf
  </p>
▶ <h2>…</h2>
  <p>moar</p>
▶ <h2>…</h2>
  <p>MOAR</p>
▶ <h2>…</h2>
  <p>asdf</p>
▶ <h2>…</h2>
  <p>MOAR</p>
▶ <h2>…</h2>
▼ <p>
    <!—MOAR key{334359b051f4dda20937055605b3706dfe91d6c8} —>
  </p>
```

5) Don't tell me something looks wrong

This was one of the ones, if not the one, we got first. By trying typical php paths: admin.php, login.php, and similar, we found the login page. We tried typical sql injection as before:
1' OR '1' = '1
in both the login and password fields. We noticed that even though the login "failed" by giving us the 404 page, the url displayed main.php, and didn't kick us back to the login page. Since we knew there must be something up, we inspected the page in chrome developer tools and searched for the key, finding it in a comment. The security issue this exploits is that we had a partial failure of credentials. We passed the username and password truth value, but failed the "lookup" or something similar. In a live system, if we got in, would we have been able to assume someone else's credentials by spoofing the password check?



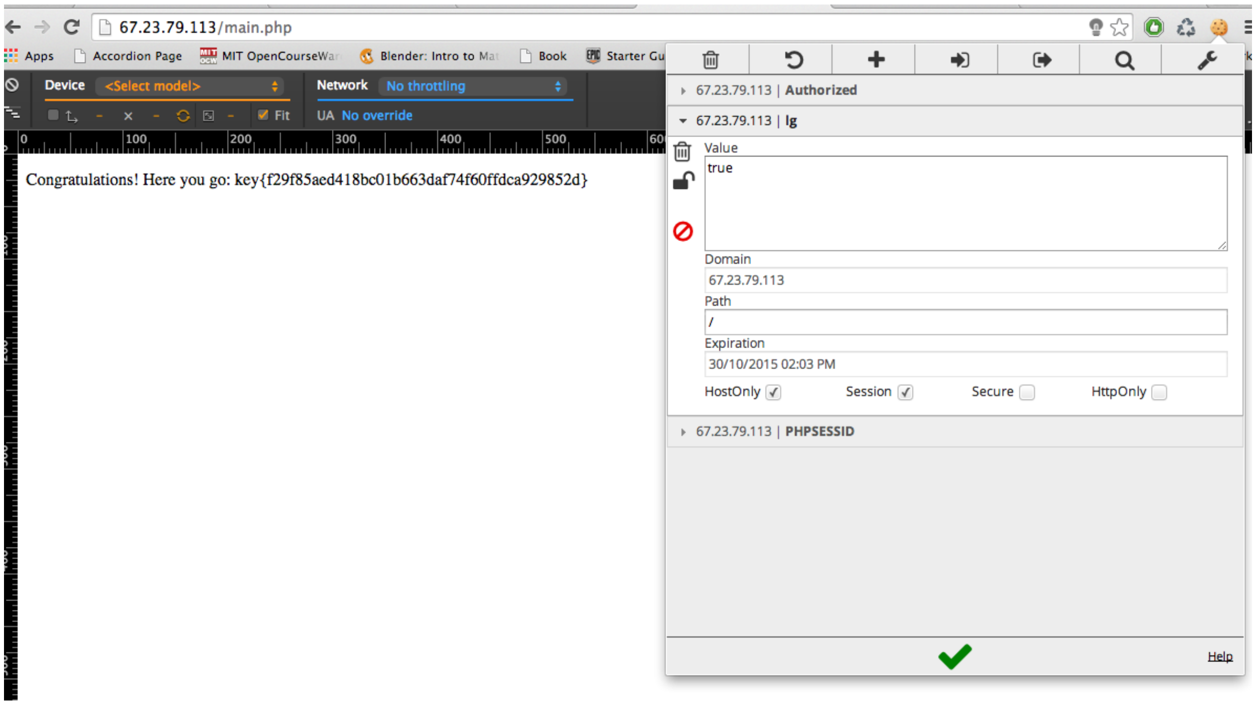
404 Not Found

nginx/1.4.6 (Ubuntu)

```
▼<html>
  ▶<head>…</head>
  ▼<body bgcolor="white">
    ▶<center>…</center>
     <hr>
     <center>nginx/1.4.6 (Ubuntu)</center>
     <!-- Hmmm, the plot thickens... key{fcb26d5bbe8d9c813034ee6a2b40eb35a96c7ff4}-->
   </body>
 </html>
```

6) More wrong

Knowing that this key and the previous one were related, we thought about what else could be on this page. We had several ways to look at the contents of the page, and saw in chrome developer tools that two cookies had been added to our browsers: Authorized and lg. We assumed lg meant logged in or something similar, and saw it was set to false. Using the chrome extension “EditThisCookie” we changed the value of lg from false to true and refreshed the page. We could have also used tamper if the process was more intensive, but this was simpler for the task. The security flaw here is that server side validation is being stored in a client side resource, likely for the purpose of persistence. The server should be guarding whether someone is logged in or not (also whether they are authorized or not) and should only be storing superfluous data in browser storage like cookies.



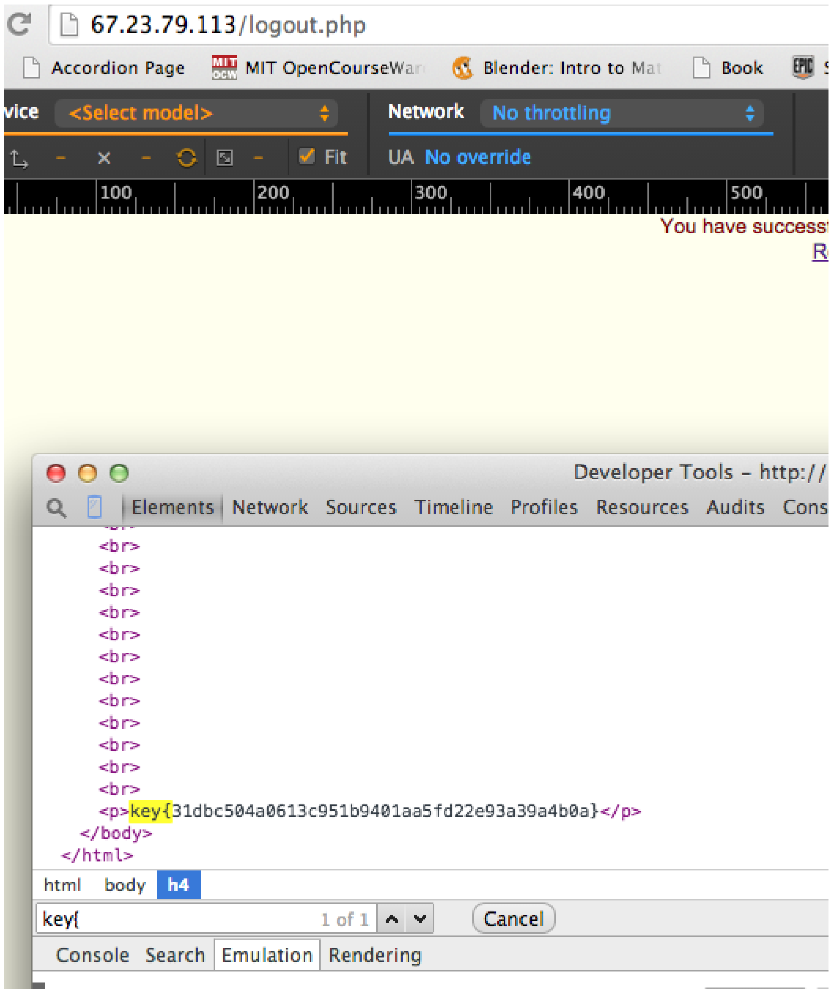
Device <Select model>
Network No throttling
UA No override

Congratulations! Here you go: key{f29f85aed418bc01b663daf74f60ffdca929852d}

67.23.79.113 | Authorized
67.23.79.113 | lg
Value: true
Domain: 67.23.79.113
Path: /
Expiration: 30/10/2015 02:03 PM
HostOnly ✓ Session ✓ Secure ☐ HttpOnly ☐
67.23.79.113 | PHPSESSID

7) You have to logout too!

The one was a given after logging in. We know where there is a login.php, there is likely a logout.php, and it turns out we were correct. Again, the security issue here is that even though there isn't a link to it, doesn't mean it isn't accessible, even with authentication mechanisms (especially if those mechanisms are on the client side).



```
<br>
<br>
<br>
<br>
<br>
<br>
<br>
<br>
<br>
<br>
<br>
<br>
<p>key{31dbc504a0613c951b9401aa5fd22e93a39a4b0a}</p>
</body>
</html>
```

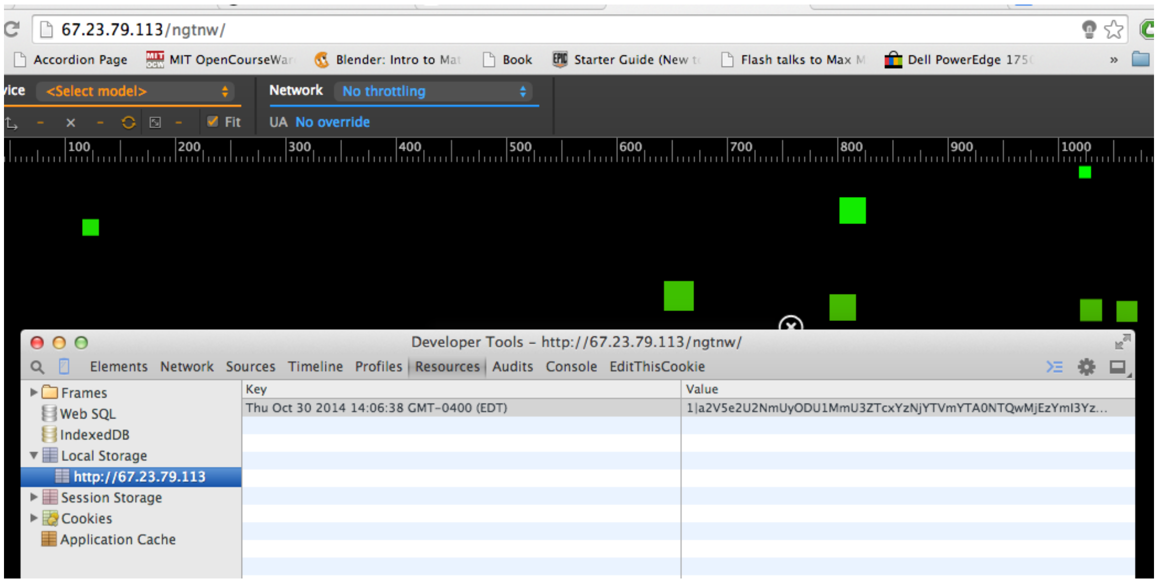
key{ 1 of 1

8) Hidden in the dump - *We did not succeed in capturing this flag*

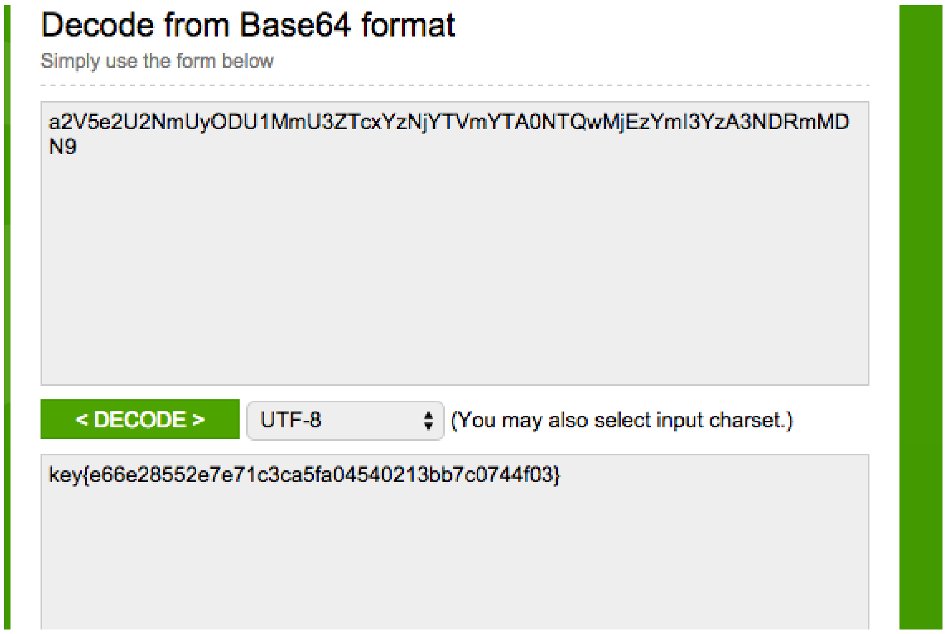
9) Not Global Thermonuclear War - All Your Base64 Are Belong to Us

We found the link for the not global thermonuculear war page within the href links on the hidden posts board page that we accessed using a sql injection. After clicking around the links of hypertext, one brought us to the game page. We recognized the game name from the hint, and remembered the extension, /ngtnw, for easy access of our teammates to the page.

After we accessed to /ngtnw, we were given a geometry-based shoot-em-up game, where after losing all of your lives, you'd be prompted to submit a name, and then your score would be recorded. At first we thought we had to tamper data to make our score 64, but when that failed, we looked around into where our score would be stored. There was nothing in cookies, but we discovered that Local Storage held data from our game--a value to a single key was given, and it consisted of our score, concatenated with a "|" symbol, and then an extremely long, peculiar-looking string. Realizing the hint had in it, "All your Base64…", we decided to try taking the second part of this value, and decode it using an online Base64 Decoder. Decoding it gave us a string of format "key{*flag*}", indicating that we did, in fact, crack the flag.



Key	Value
Thu Oct 30 2014 14:06:38 GMT-0400 (EDT)	1|a2V5e2U2NmUyODU1MmU3ZTcxYzNjYTVmYTA0NTQwMjEzYml3Yz...


Decode from Base64 format

Simply use the form below

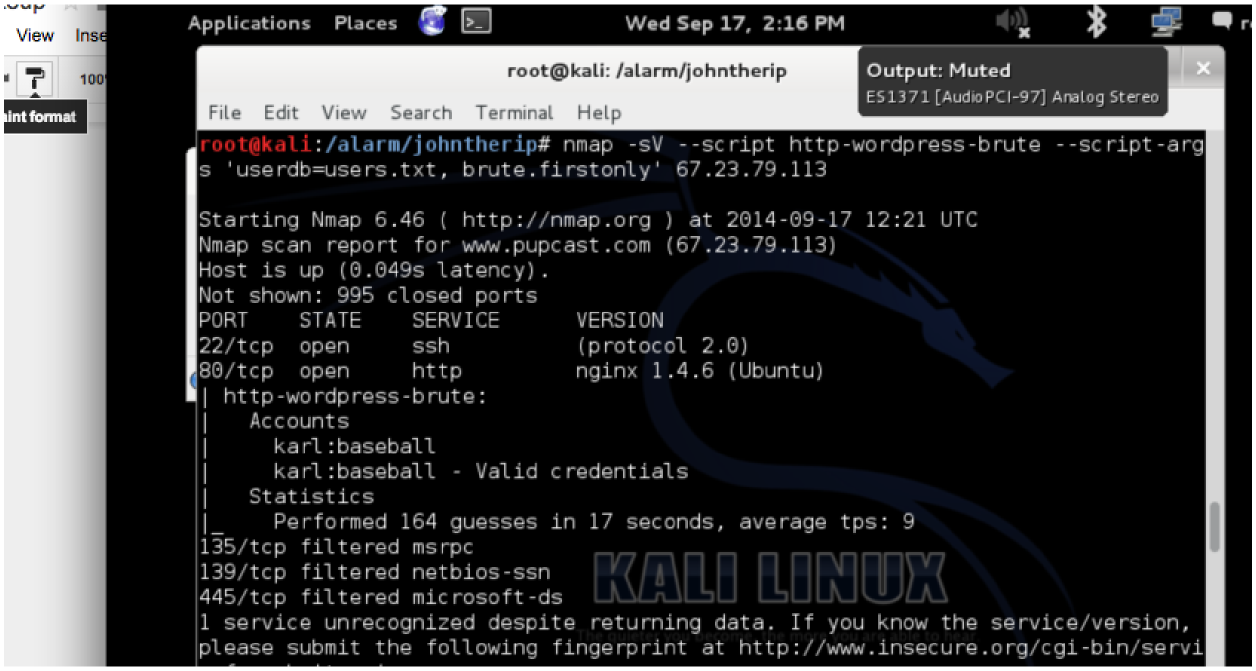
```
a2V5e2U2NmUyODU1MmU3ZTcxYzNjYTVmYTA0NTQwMjEzYml3YzA3NDRmMD
N9
```

(You may also select input charset.)

```
key{e66e28552e7e71c3ca5fa04540213bb7c0744f03}
```

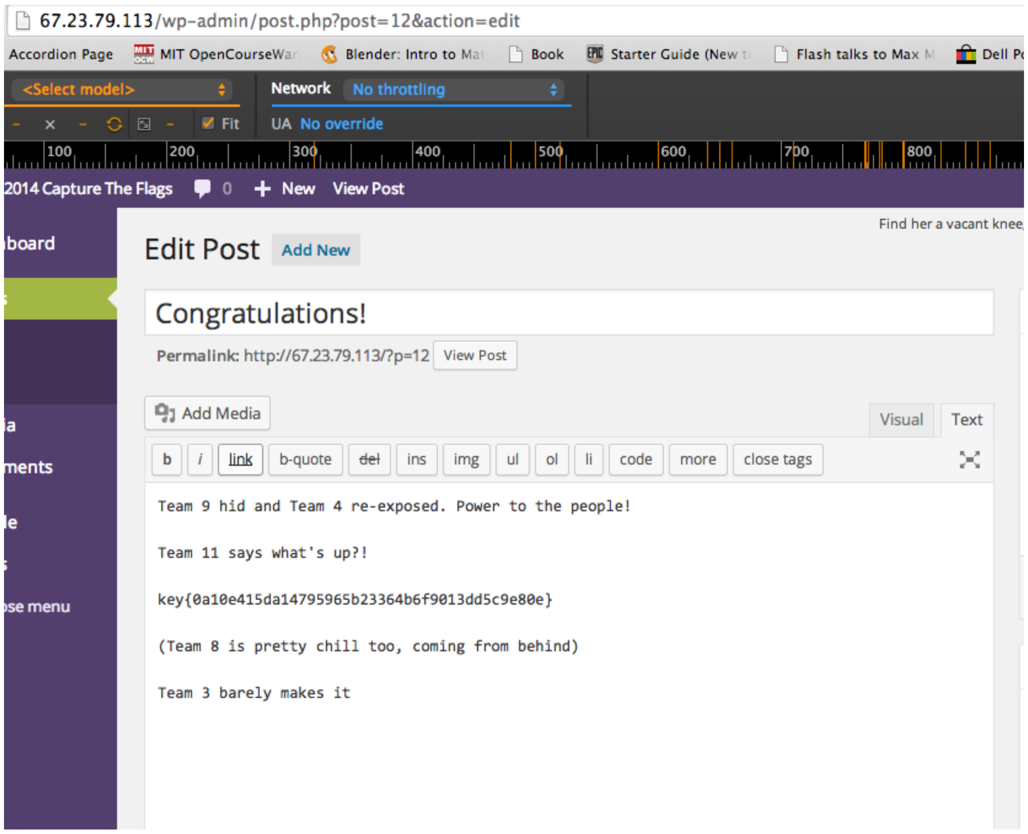
10) About my friend Karl...

This last task involved pulling several clues together to break into the wp-admin account. We noticed from the hint and from the “author” tag scattered throughout the site that karl was a user. We also knew that since this was a wordpress account the admin login page would be located at ‘/wp-admin’ Once we reached the login page we confirmed that karl was a username when we tested ‘karl’ and a SQL injection as a password and received a message along the lines of “The password for karl is not correct.” With a username in hand and with SQL injection seemingly ineffective, we decided to try to brute force our way into the account. We googled the most popular wordpress brute force techniques and came across an nmap scan run with a script. Since we had used nmap in the past we thought we would try it out. The technique also allowed you to pass a list of known usernames which could expedite the process and tested initially with a general set of password wordlists (although you could add your own as well). The script was ‘http-wordpress-brute’ and you can see the command we ran below in the screenshot provided. We used the users.txt wordlist that only contained the usernames ‘admin’ and ‘karl’. We also ran with the brute.firstonly boolean set to true which meant the scan would stop once one set of credentials was validated. In only 17 seconds and 164 guesses we were able to validate the username password combination of ‘karl’ and ‘baseball’. We then simply had to login and we had access to the entire wordpress blog. Once we had gained access we just had to poked around to find a private post containing the key. The central idea behind this challenge was that a weak password is a flaw in your site. We could not use SQL injection or alter cookies to hack into the account but all we had to do was a guess a (relatively) few common passwords and we were able to gain access. This challenge also showed some common vulnerabilities of wordpress including the fact that all admin logins occur at the /wp-admin page and that often wordpress blogs will tell you that you have not entered the correct password for that username (and thereby confirming the existence of that username).


Applications
Places
Wed Sep 17, 2:16 PM
Output: Muted
ES1371 [AudioPCI-97] Analog Stereo
root@kali: /alarm/johntherip
File Edit View Search Terminal Help

```
root@kali:/alarm/johntherip# nmap -sV --script http-wordpress-brute --script-arg
s 'userdb=users.txt, brute.firstonly' 67.23.79.113

Starting Nmap 6.46 ( http://nmap.org ) at 2014-09-17 12:21 UTC
Nmap scan report for www.pupcast.com (67.23.79.113)
Host is up (0.049s latency).
Not shown: 995 closed ports
PORT     STATE    SERVICE      VERSION
22/tcp   open     ssh          (protocol 2.0)
80/tcp   open     http         nginx 1.4.6 (Ubuntu)
| http-wordpress-brute:
|   Accounts
|     karl:baseball
|     karl:baseball - Valid credentials
|   Statistics
|_    Performed 164 guesses in 17 seconds, average tps: 9
135/tcp filtered msrpc
139/tcp filtered netbios-ssn
445/tcp filtered microsoft-ds
1 service unrecognized despite returning data. If you know the service/version,
please submit the following fingerprint at http://www.insecure.org/cgi-bin/servi
```



Edit Post

Congratulations!

Permalink: http://67.23.79.113/?p=12 View Post

```
Team 9 hid and Team 4 re-exposed. Power to the people!

Team 11 says what's up?!

key{0a10e415da14795965b23364b6f9013dd5c9e80e}

(Team 8 is pretty chill too, coming from behind)

Team 3 barely makes it
```