CyberSec Project 3

Zack McKevitt (zamc2229@colorado.edu), Connor Radeloff (cora0945@colorado.edu), Sahib Bajwa (saba9473@colorado.edu)

Basic Linux Exploits: Zack McKevitt

Level 0

This level was to connect to the overthewire servers using SSH. To do this, I ran the command:
ssh bandit0@bandit.labs.overthewire.org -p 2220
With the password: bandit0 (which was given)

This command allowed me to connect to bandit.labs.overthewire.org on port 2220, with a username of bandit0 and a password of bandit0. This allowed me to successfully log in via SSH.

1 minute

Level 0-1

For this level, I had to find a password located in a readme. I used the command ‘ls’ to list the contents of the current directory, which resulted in a file called readme. Then, I used the ‘cat’ command to read this file, which contained the password. This password was:

boJ9jbbUNNfktd78OOpsqOltutMc3MY1

1 minute

Level 1-2

After logging into bandit1 over SSH with the password above, I could start the level. The password for the next level was stored in a file called ‘-’ (minus the quotes). At first, I simply tried the command ‘cat -’, but this did not work. I assume this is because flags are also denoted by the ‘-’ symbol, so the kernel was confusing the filename as a flag or something.

After some research, I realized I needed to use the relative path for the file. So, I used the command ‘cat ./-’ to locate the file using its relative path. This worked, and I received the password of:

CV1DtqXWVFXTvM2F0k09SHz0YwRINYA9

3 minutes

Level 2-3

After logging into bandit2 over SSH with the password above, I could start the level. I was looking for a password in a file named ‘spaces in this filename’. So, I again used the ‘cat’ command in combination with the relative path. This command looked like:

cat ./spaces\ in\ this\ filename

Here, the ‘\ ‘ denotes a space in the filename. This command allowed me to view the contents of the folder and find the next password:

UmHadQclWmgdLOKQ3YNgjWxGoRMb5luK

1 minute

Level 3-4

After logging into bandit3 over SSH with the password above, I could start the level. This level required me to find a password in a hidden file. I knew the file was in a directory called ‘inhere’, so I used the command ‘cd inhere/’ to switch to that directory. I tried using ‘ls’ to view the files in this directory, but since this file is hidden, nothing showed up. Then I used the ‘find’ command, which showed me the hidden file in the directory. Ifind was able to view the contents of this file with the command:

cat ./.hidden

Which resulted in the following password:

pIwrPrtPN36QITSp3EQaw936yaFoFgAB

2 minutes

Level 4-5

After logging into bandit4 over SSH with the password above, I could start the level. For this level, I once again had to navigate to the ‘inhere’ directory by using the command ‘cd inhere/’. The level stated that one of the 10 files is human readable, and this file contains the password. After checking the contents of each file using the ‘cat’ command, I found that ‘-file07’ was the

only file that was readable and contained a password. I extracted the password with the command:

cat ./-file07

Which produced the password:

koReBOKuIDDepwhWk7jZC0RTdopnAYKh

3 minutes

Level 5-6

After logging into bandit5 over SSH with the password above, I could start the level. This one required us to find a file that was human readable, 1033 bytes in size, and non executable. At first, I tried using the 'ls -l' command to view the file sizes for each file in each directory. After coming up short with no password, I decided to learn the 'find' command. This allowed me to search the directory for a file with a certain size and that was not executable. I used the following command:

find ./ -type f -size 1033c

This function allowed me to search the entire 'inhere' directory for all files of size 1033 bytes. One file was returned, a hidden file called .file2 in the mayberhere07 directory. This file gave the following password:

DXjZPULLxYr17uwoI01bNLQbtFemEgo7

10 minutes

Level 6-7

After logging into bandit6 over SSH with the password above, I could start the level. I did some digging around, and found myself in the '/var/' directory. From here, I could use the 'find' command. The specific command I used was:

find ./ -type f -size 33c

Since the file size was given again, that's what I searched for. Most of these files were locked and I did not have permission to view them. However, one file was not locked, and the file was bandit7.password. I then used the following command to open the file:

cat ./lib/dpkg/info/bandit7.password

Which yielded the following password:

HKBPTKQnIay4Fw76bEy8PVxKEDQRKTzs

5 minutes

Level 7-8

After logging into bandit7 over SSH with the password above, I could start the level. It provided a text file and told me that the password was located next to the word ‘millionth’. I then used the following grep command to find the password:

grep ‘millionth’ data.txt

Which gave the password:

cvX2JJa4CFALtqS87jk27qwqGhBM9plV

5 minutes

Level 8-9

After logging into bandit8 over SSH with the password above, I could start the level. Once again, I was given a text file to parse. The password is the only unique line in the text file, so I used the ‘sort’ command:

sort data.txt | uniq -u

Which resulted in the password:

UsvVyFSfZZWbi6wgC7dAFyFuR6jQQUhR

3 minutes

Level 9-10

After logging into bandit9 over SSH with the password above, I could start the level. I had to look into an unreadable file and find a password located next to many equals signs. So, I converted the document to string format and searched for all line starting with '=':

strings data.txt | grep '^='

And found the password:

truKLdjsbJ5g7yyJ2X2R0o3a5HQJFuLk

3 minutes

Level 10-11

The answer to this level was located in a base64 encoded file. So, I used the command:

cat data.txt | base64 -d

And got the following password:

IFukwKGsFW8MOq3IRFqrxE1hxTNEbUPR

2 minutes

Level 11-12

The password for this level was hidden behind a cipher, specifically ROT13. After some research on linux commands, I found the command 'tr' to be useful. I could translate all characters of the alphabet to their respective mappings and print the results with the following command:

cat data.txt | tr 'A-Za-z' 'N-ZA-Mn-za-m'

Which gave the password:

5Te8Y4drgCRfCx8ugdwuEX8KFC6k2EUu

15 minutes

Level 12-13

This level was very tricky. There was a hexdump that had been compressed many times and this dump contained the password. The first thing we needed to do was reverse the hexdump using the ‘xxd’ command. So, I used the command:

‘cat data.txt | xxd -r > data’

Which reversed the hexdump, hence the ‘-r’, into a file called data. I then used the following command to check the metadata of the file by typing:

‘file data’

Which showed that this is compressed with gzip. So, I used ‘mv data data2.gz’ to add a .gz extension and then used ‘gzip -d data2.gz’ to unpack it. After examining data2 with ‘file data2’, we could see that this file was now bzip2 compressed. Following a similar routine as above, we converted the file type to data3.bz, decompressed it with ‘bzip2 -d data3.bz’. There was a long sequence of unpacking different types of compressed files (tar, bz, gz), but after the 9th sequence, the final file was in ASCII format, to which we examined its contents to contain the password:

8ZjyCRiBWFYkneahHwxCv3wb2a1ORpYL

30+ minutes

Level 13-14

To find the password, we had to look in a file located in /etc/bandit_pass/bandit14. However, this file can only be accessed by the user bandit14. In this level, we are bandit13, which is not enough. So, we had to login again as user bandit14. I noticed that there was a private RSA key located in the home directory, so I used that key to login to bandit 14 with the command:
ssh bandit14@bandit.labs.overthewire.org -p 220 -i sshkey.private

From there, I could navigate to /etc/bandit_pass/bandit14 and get the following password:

4wcYUJFw0k0XLShlDzztnTBHiqxU3b3e

5 minutes

Level 14-15

This level required me to submit the password from the previous level to localhost on port 30000. At first, I tried unsuccessfully to use commands like ssh or telnet to login to localhost on port 30000, but I did not get very far. After some research, I discovered the ‘nc’ command, which will allow us to establish a connection to the desired localhost. I was also unsure about how to submit the previous password, but I found that it is possible to use the ‘echo’ command in conjunction with ‘nc’. So, my final command looked like:

echo 4wcYUJFw0k0XLShlDzztnTBHiqxU3b3e | nc localhost 30000

Which was me sending the password to the current level to the destination, which gave the following password:

BfMYroe26WYalil77FoDi9qh59eK5xNr

30+ minutes

Level 15-16

This level was similar to the last, but we are now submitting the data to localhost on port 30001 and encrypting it with SSL. Since I knew to use SSL, I followed a similar approach to the last problem, but using ‘openssl’ instead of ‘nc’. I also had to use ‘s_client’ in addition to ‘openssl’, which allowed us to test the connection. From there, I just had to set my destination and add the flag ‘ign_eof’ to ensure that the connection stayed open to read the data sent over. The command that ended up working for me was:

echo -e BfMYroe26WYalil77FoDi9qh59eK5xNr | openssl s_client -connect localhost:30001 -ign_eof

Which gave the following password:

cluFn7wTiGryunymYOu4RcffSxQluehd

15 minutes

Level 16-17

The key for this level is stored on a server open on localhost on some port between 31000-32000. To find out which of these ports was active, I used ‘nmap’:

nmap -p 31000-32000 localhost

Which gave me two ports: 31518 and 31790. I saw that 31790 was open while 31518 was filtered, so I tried that one first with the command:

echo -e cluFn7wTiGryunymYOu4RcffSxQluehd | openssl s_client -connect localhost:31790 -ign_eof

Which gave me a RSA private key. I saved this private key to a file on my computer, which I used to login to the next level. The key is here:

-----BEGIN RSA PRIVATE KEY-----
MIIEogIBAAKCAQEAvmOkuifmMg6HL2YPIOjon6iWfbp7c3jx34YkYWqUH57SUdyJ
imZzeyGC0gtZPGujUSxiJSWI/oTqexh+cAMTSMlOJf7+BrJObArnxd9Y7YT2bRPQ
Ja6Lzb558YW3FZl87ORiO+rW4LCDCNd2lUvLE/GL2GWyuKN0K5iCd5TbtJzEkQTu
DSt2mcNn4rhAL+JFr56o4T6z8WWAW18BR6yGrMq7Q/kALHYW3OekePQAzL0VUYbW
JGTi65CxbCnzc/w4+mqQyvmzpWtMAzJTzAzQxNbkR2MBGySxDLrjg0LWN6sK7wNX
x0YVztz/zbIkPjfkU1jHS+9EbVNj+D1XFOJuaQIDAQABAoIBABagpxpM1aoLWfvD
KHcj10nqcoBc4oE11aFYQwik7xfW+24pRNuDE6SFthOar69jp5RlLwD1NhPx3iBl
J9nOM8OJ0VToum43UOS8YxF8WwhXriYGnc1sskbwpXOUDc9uX4+UESzH22P29ovd
d8WErY0gPxun8pbJLmxkAtWNhpMvfe0050vk9TL5wqbu9AlbssgTcCXkMQnPw9nC
YNN6DDP2lbcBrvgT9YCNL6C+ZKufD52yOQ9qOkwFTEQpjtF4uNtJom+asvlpmS8A
vLY9r60wYSvmZhNqBUrj7lyCtXMIu1kkd4w7F77k+DjHoAXyxcUp1DGL51sOmama
+TOWWgECgYEA8JtPxP0GRJ+IQkX262jM3dEIkza8ky5moIwUqYdsx0NxHgRRhORT
8c8hAuRBb2G82so8vUHk/fur85OEfc9TncnCY2crpoqsghifKLxrLgtT+qDpfZnx
SatLdt8GfQ85yA7hnWWJ2MxF3NaeSDm75Lsm+tBbAiyc9P2jGRNtMSkCgYEAypHd
HCctNi/FwjulhttFx/rHYKhLidZDFYeiE/v45bN4yFm8x7R/b0iE7KaszX+Exdvt
SghaTdcG0Knyw1bpJVyusavPzpaJMjdJ6tcFhVAbAjm7enCIvGCSx+X3l5SiWg0A
R57hJglezIiVjv3aGwHwvlZvtszK6zV6oXFAu0ECgYAbjo46T4hyP5tJi93V5HDi
Ttiek7xRVxUl+iU7rWkGAXFpMLFteQEsRr7PJ/lemmEY5eTDAFMLy9FL2m9oQWCg
R8VdwSk8r9FGLS+9aKcV5PI/WEKlwgXinB3OhYimtiG2Cg5JCqIZFHxD6MjEGOiu
L8ktHMPvodBwNsSBULpG0QKBgBAplTfC1HOnWiMGOU3KPwYWt0O6CdTkmJOmL8Ni
blh9elyZ9FsGxsgtRBXRsqXuz7wtsQAgLHxbdLq/ZJQ7YfzOKU4ZxEnabvXnvWkU
YOdjHdSOoKvDQNWu6ucyLRAWFuISeXw9a/9p7ftpxm0TSgyvmfLF2MIAEwyzRqaM
77pBAoGAMmjmIJdjp+Ez8duyn3ieo36yrttF5NSsJLAbxFpdlc1gvtGCWW+9Cq0b
dxviW8+TFVEBl1O4f7HVm6EpTscdDxU+bCXWkfjuRb7Dy9GOtt9JPsX8MBTakzh3
vBgsyi/sN3RqRBcGU40fOoZyfAMT8s1m/uYv52O6IgeuZ/ujbjY=
-----END RSA PRIVATE KEY-----

10 minutes

Level 17-18

This level requires me to find the sole difference between two files, passwords.old and passwords.new. To do this, I used the ‘diff’ command:

diff passwords.new passwords.old

Which gave the following output:

42c42
< kfBf3eYk5BPBRzwjqutbbfE887SVc5Yd

> hlbSBPAWJmL6WFDb06gpTx1pPButblOA

Since I put the passwords.new file as the first one, its respective password was printed first, so our new password is:

kfBf3eYk5BPBRzwjqutbbfE887SVc5Yd

3 minutes

Level 18-19

When logging into this level, I get a byebye! This was very confusing at first, and I thought I encountered a bug. When I tried to skip this level and move on, I realized I needed to find the password before continuing. I finally realized that this logout was intentional, so I researched staying logged in. Luckily, the hint told me that the password was located in a readme in the home directory. I then learned that I can run commands directly from the ssh command, which would allow me to view the contents of the readme before getting the boot. So, I used the following command:

ssh bandit18@bandit.labs.overthewire.org -p 2220 ‘cat ~/readme’

The addition of ‘cat ~/readme’ at the end of the command would allow me to view the readme before being logged out, which resulted in the password:

IueksS7Ubh8G3DCwVzrTd8rAVOwq3M5x

15 minutes

Level 19-20

For this level, I had to change my uid to access the bandit20 password in /etc/bandit_pass/bandit20. There was a file in the home directory that allowed me to change my uid. So, I changed my uid to the uid associated with bandit20 (11020) and read the file using the 'cat' command. More specifically, the command I used was:

./bandit20-do uid=11020 cat /etc/bandit_pass/bandit20

To get the following password:

GbKksEFF4yrVs6il55v6gwY5aVje5f0j

5 minutes

Level 20-21

The first thing I had to do on this level was to set up a basic server on a port of my choice that contains the password we previously got in order to unlock this current level. Similarly to some of the previous challenges, I used the 'nc' command combined with 'echo':

echo 'GbKksEFF4yrVs6il55v6gwY5aVje5f0j' | nc -l localhost -p 53534 &

Here, I just chose a random port of 53534 to host the server. Then, I used the given file, suconnect, which takes an input for the port number, to read the line of data transmitted on this port. The '&' signifies that we want to run this as a background process. Since I echoed the previous password, this suconnect file should read that line and verify that it matches. It did match, and we are granted with our next password:

gE269g2h3mw3pwgrj0Ha9Uoqen1c9DGr

20 minutes

Level 21-22

The first thing I did to solve this problem was navigate to /etc/cron.d to see these 'cron' files. I then decided to check out a file called cronjob_bandit22, which I assumed would contain the password for level 22. When I checked this file with 'cat', I saw a shell file hidden inside. Again, I wanted to check out this file with 'cat', so I used the following command to view the shell file:
cat /usr/bin/cronjob_bandit22.sh

Inside this shell file were some commands pointing to another file. This file was located at '/tmp/t7O6lds9S0RqQh9aMcz6ShpAoZKF7fgv'. Again, I used 'cat' to view the contents of this file, which contained the password to the next level:

Yk7owGAcWjwMVRwrTesJEwB7WVOiILLI

15 minutes

Level 22-23

This level had a similar setup to the previous. However, when I tried the same exact process as last time, I was getting the same password as last time. So, I investigated this new shell script for cronjob_bandit23. There, I saw it was copying the password for the current level (it was using the command 'whoami', which resulted in bandit22. However, we need to be bandit23). The script then saved the user as a variable called 'myname'. Then, it created a new path in the '/tmp/' folder based off of the 'myname' variable. Without using the script, I was able to set the myname variable directly in the shell by typing:

'myname=bandit23'

From the shell script, I was able to find that the new path would be equal to $(echo I am user $myname | md5sum | cut -d ' ' -f 1). Instead of running this in the shell script, I copied and pasted it directly in the shell to get a new file path that contains the password. Since the variable 'myname' is now bandit23, we will get the correct file containing the password. After viewing its contents with 'cat', I was able to view the password:

jc1udXuA1tiHqjIsL8yaapX5XIAI6i0n

20 minutes

Level 23-24

Again, this setup is similar to the previous two. But when I check the shell script for bandit24, I can see that all files in /var/spool/bandit24 will be **executed** and then deleted. So, I know I can insert a shell file here that will be executed. So, I created a working directory in the /tmp/ location so that I can make a bash file. In the bash file, all I am doing is copying the contents of /etc/bandit_pass/bandit24 into a new file in my temp directory which I called password. Then, I copied my script into /var/spool/bandit24 because I know everything in there will execute and delete. After waiting for the file to execute, which I will know has happened when it is deleted,

we can navigate back to our temp directory and open the password file, which contains the password:

UoMYTrfrBFHyQXmg6gzctqAwOmw1IohZ

30 minutes

Level 24-25

This challenge required a 4 digit PIN to be submitted in addition to the previous password in order to get the next password. Since this PIN is not found anywhere, the only way to guess it is to brute force every combination. Since it is a 4 digit combination, there are 10000 combinations that need to be tried. So, a python script can be written to automate this process.
This script connects to the given daemon (localhost on port 30002). After connecting to the socket, we sent the PIN (displayed as 4 digits) + password to the server and analyzed the response. If it was "Wrong", we would increment the PIN by 1 and try again until the correct PIN was found. In our case, the correct PIN was 9403 (which took forever!), yielding the following password:

uNG9O58gUE7snukf3bvZ0rxhtnjzSGzG

30-45 minutes

Level 25-26

This level gives us a ssh key for bandit 26. When I tried logging in to bandit26 with this password using the command:

ssh bandit26@localhost -i bandit26.sshkey

I get kicked out almost immediately. But, I found that the only way to not get kicked out is to minimize the amount of text on screen. This will force the shell to display the 'more' command. But, when this command is displayed, we can actually force other commands. So, since I know all passwords are stored in '/etc/bandit_pass/', I decided I should check there for the next password. To do this, I would need to enter vim while the 'more' prompt is still showing. I was able to successfully enter vim by pressing 'v'. Once inside, I could use the ':e' command to view the contents of the /etc/bandit_pass/bandit26 file by entering this command into vim:

:e /etc/bandit_pass/bandit26

Which showed the following password:

5czgV9L3Xx8JPOyRbXh6lQbmIOWvPT6Z

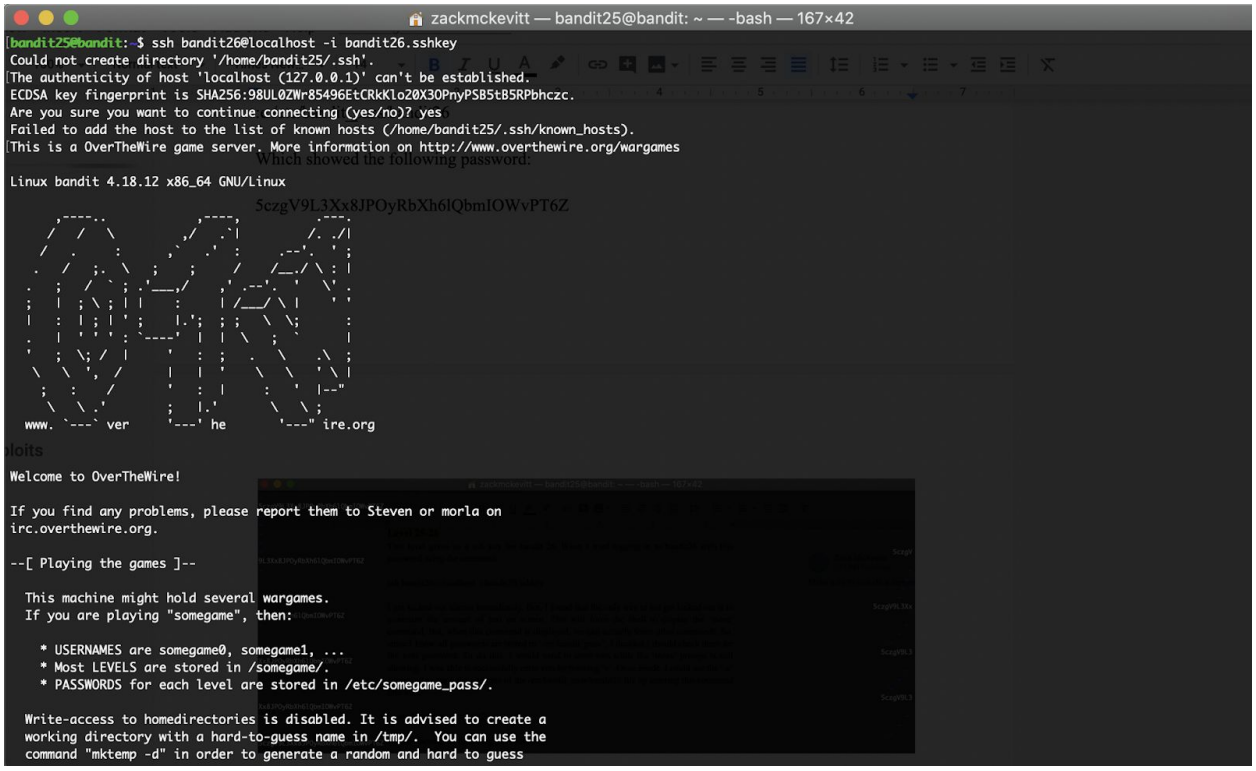
However, the shell for bandit26 is NOT /bin/bash. So, we will change it with the following command, which we will execute in VIM:

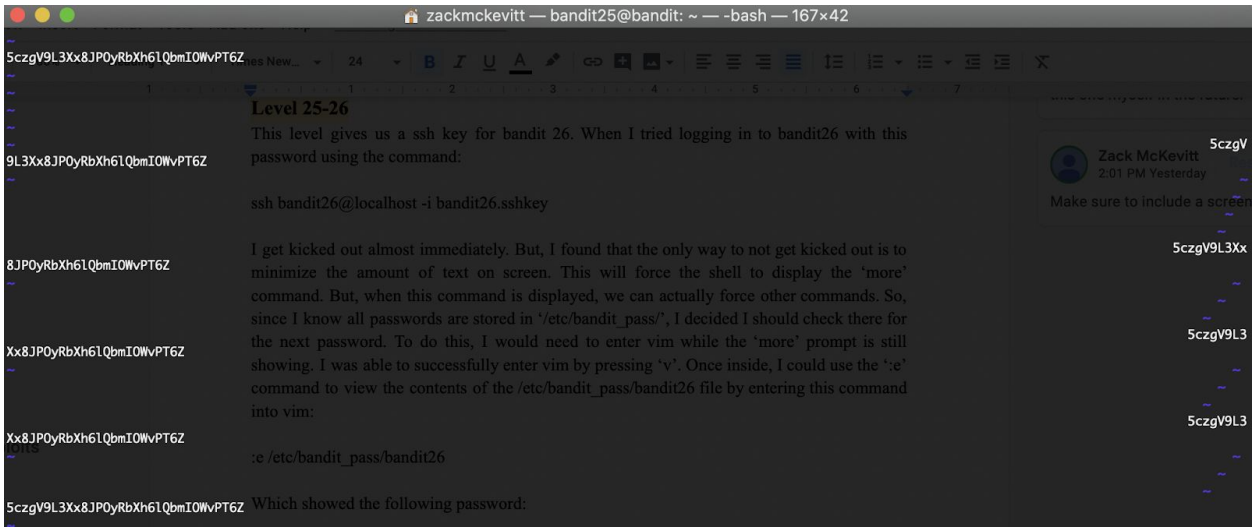
:set shell=/bin/bash

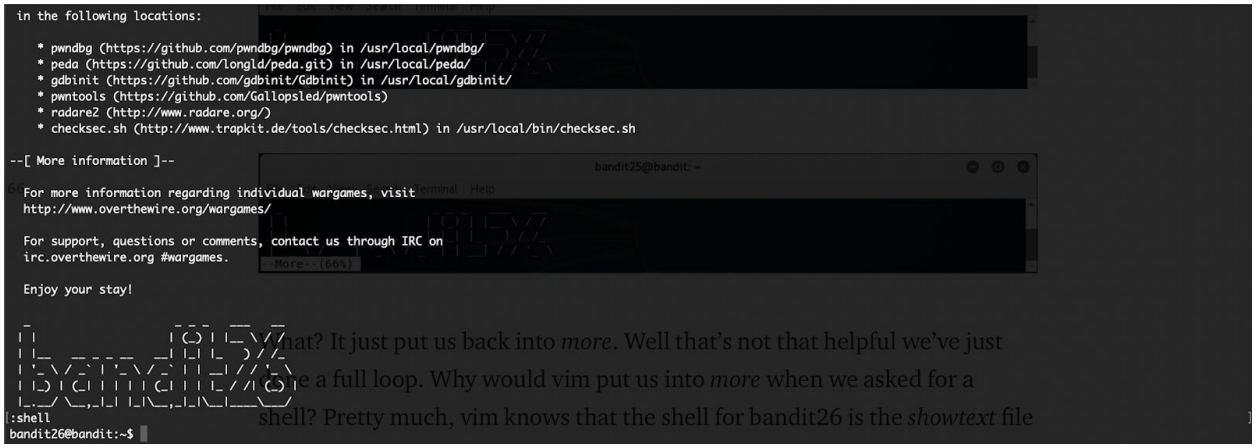
And afterwards, we can open the shell with the following VIM command:

:shell

20 minutes


zackmckevitt — bandit25@bandit: ~ — -bash — 167×42
[bandit25@bandit:~$ ssh bandit26@localhost -i bandit26.sshkey
Could not create directory '/home/bandit25/.ssh'.
The authenticity of host 'localhost (127.0.0.1)' can't be established.
ECDSA key fingerprint is SHA256:98UL0ZWr85496EtCRkKlo20X3OPnyPSB5tB5RPbhczc.
Are you sure you want to continue connecting (yes/no)? yes
Failed to add the host to the list of known hosts (/home/bandit25/.ssh/known_hosts).
This is a OverTheWire game server. More information on http://www.overthewire.org/wargames
Linux bandit 4.18.12 x86_64 GNU/Linux
Welcome to OverTheWire!
If you find any problems, please report them to Steven or morla on
irc.overthewire.org.
--[Playing the games]--
This machine might hold several wargames.
If you are playing "somegame", then:
* USERNAMES are somegame0, somegame1, ...
* Most LEVELS are stored in /somegame/.
* PASSWORDS for each level are stored in /etc/somegame_pass/.
Write-access to homedirectories is disabled. It is advised to create a
working directory with a hard-to-guess name in /tmp/. You can use the
command "mktemp -d" in order to generate a random and hard to guess


zackmckevitt — bandit25@bandit: ~ — -bash — 167×42
5czgV9L3Xx8JPOyRbXh6lQbmIOWvPT6Z
9L3Xx8JPOyRbXh6lQbmIOWvPT6Z
8JPOyRbXh6lQbmIOWvPT6Z
Xx8JPOyRbXh6lQbmIOWvPT6Z
Xx8JPOyRbXh6lQbmIOWvPT6Z
5czgV9L3Xx8JPOyRbXh6lQbmIOWvPT6Z
Level 25-26
This level gives us a ssh key for bandit 26. When I tried logging in to bandit26 with this password using the command:
ssh bandit26@localhost -i bandit26.sshkey
I get kicked out almost immediately. But, I found that the only way to not get kicked out is to minimize the amount of text on screen. This will force the shell to display the 'more' command. But, when this command is displayed, we can actually force other commands. So, since I know all passwords are stored in '/etc/bandit_pass/', I decided I should check there for the next password. To do this, I would need to enter vim while the 'more' prompt is still showing. I was able to successfully enter vim by pressing 'v'. Once inside, I could use the ':e' command to view the contents of the /etc/bandit_pass/bandit26 file by entering this command into vim:
:e /etc/bandit_pass/bandit26
Which showed the following password:
5czgV
Zack McKevitt
2:01 PM Yesterday
Make sure to include a screen
5czgV9L3Xx
5czgV9L3
5czgV9L3



General Web Exploits: Sahib Bajwa

Level 0

To log into level 0, we were given the username and password. I used the username:

natas0

And the given password:

natas0

Level 0-1

For this level, the website states that we can find the password for level 1 on level 0's page. Inspecting the page's html code I when through the various tags looking for the password. When opening the <body> tag and then the tag, we can see the password displayed as a comment in the code. The password written in the comment was:

gtVrDuiDfck831PqWsLEZy5gyDz1clto

Level 1-2

On this level, right-clicking has been disabled (they did this by setting an alert to occur whenever right click is pressed). To get around this, I pressed F12 instead. This also brings up the same interface as if we had right-clicked and inspected the page. Similar to level 0, I went through the tags in the html code and was able to find the password in a comment after opening the <body> and tags. The password written in the comment was:

ZluruAthQk7Q2MqmDeTiUij2ZvWy2mBi

Level 2-3

This level starts out with it telling you that there is nothing on the page. Inspecting the page, and going through the tags, I saw that there was a link to an image that did not appear on the screen. The link to the image was contained in an image tag, <img src=”files/pixel.png”>. Going to this image, by adding “/files/pixel.png” to the end of the level url, ended up not giving me anything. The page is completely black, and inspecting the page gives nothing as to what the next password is. Considering that this page exists, even though it is blank, means that the “/files” page most likely exists as well. Going to the files page displays and index of files. pixel.png had nothing on it earlier so I ignored it. Opening users.txt, the password can be seen in a list of user password combinations. The password for level 3 (natas3) was:

sJIJNW6ucpu6HPZ1ZAchaDtwd7oGrD14

Level 3-4

This level starts the same as the last level, saying that there is nothing on the page. Again, inspecting the page and going through the tags results in a comment that says “No more information leaks!! Not even Google will find it this time...:”. My next step was to indeed search google for “google information leaks”. This ended up not resulting in anything useful. I then searched “how to remove something from google search”. This resulted in a google page where at the bottom, it talks about robots.txt. Finally remembering what robots.txt is and what it does, I appended the level url with “/robots.txt”. This resulted in a web page that showed “/s3cr3t/” was disallowed from showing up. Appending “/s3cr3t/” to the end of the level url showed another users.txt file which contained the password. The next password given was:

Z9tkRkWmpt9Qr7XrR5jWRkgOU901swEZ

Level 4-5

Figuring out exactly how to get around the roadblock in this level took a little bit of time. Upon loading the page we are told that users visiting from “http://natas5.natas.labs.overthewire.org/” are authorized, but since we are visiting from “http://natas4.natas.labs.overthewire.org/”, we are not authorized. First I inspected the page. In the html code, I noticed a link to “index.php”. This turns out to be nothing since it is just the same page and is used when we use the refresh tool on the page. I then decided to take a look at the network tab instead of the html code. Inspecting the network tab and refreshing the page showed a few items being loaded on refresh. Clicking on index.php, which was simply the page’s id, I saw in the request header that there was a referer from “http://natas4.natas.labs.overthewore.org/index.php”. Unfortunately, I could not simply

change it from natas4 to natas5, so I needed to use a referer control plugin to accomplish this. Changing the referer to "http://natas5.natas.labs.overthewore.org/index.php" and reloading the page displayed the next password. The next password given was:

iX6IOfmpN7AYOQGPwtn3fXpbaJVJcHfq

Level 5-6

This level was much simpler than the last one. Upon opening this level, we are told that access is disallowed since we are not logged in. Inspecting the page's html code gave me nothing, so I moved on to the application tab. Here, I went to the cookies subsection and saw that there was a cookie for this page that was named "loggedin" and had a value of 0. Editing this value to 1 and refreshing the page caused the password to be displayed. The next password given was:

aGoY4q2Dc6MgDq4oL4YtoKtyAg9PeHa1

Level 6-7

On this level we are told to input a secret. Entering an incorrect value tells us we have inputted the wrong secret. Pressing the "View sourcecode" button shows us the sourcecode for the page. Inside this sourcecode I can see a line that says "include "includes/secret.inc";" appending "/includes/secret.inc" takes me to a page that says the "secret="FOEIUWGHFEEUHOFUOIU"". Entering "FOEIUWGHFEEUHOFUOIU" on the main level page and submitting grants me access to the password. The next password given was:

7z3hEENjQtflzgnT29q7wAvMNfZdh0i9

Level 7-8

On this level, we are given two buttons. One labeled "Home", and the other one labeled "About". Going to either of these pages was not useful. Inspecting the "Home" page, there is a comment in the html code that says "hint: password for webuser natas8 is in /etc/natas_webpass/natas8". Going to the url "http://natas7.natas.labs.overthewire.org/etc/natas_webpass/natas8" does not give us anything. Going back to the "Home" page, I can see that the url ends with "/index.php?page=home". Removing "home" and adding "/etc/natas_webpass/natas8" took me to a new page and displayed the password. The next password given was:

DBfUBfqQG69KvJvJ1iAbMoIpwSNQ9bWe

Level 8-9

This level was another “Input secret:” level. Clicking the “View sourcecode” button, I could see that there was a line that said “$encodedSecret = "3d3d516343746d4d6d6c315669563362";”. Right below this was a function that was used to encode the secret. The function encoded it using base64, then reversed it (as in reversed the string of characters), then changed it from binary to hex. So in order to find the secret, I decided to do all of these steps in reverse. Searching for base64_decode, results saying that there is a PHP function that does exactly that. I created a PHP file that contained the command:
“print base64_decode(strrev(hex2bin('3d3d516343746d4d6d6c315669563362')));”
Running the file gave me “oubWYf2kBq”, which was correct when entered for the secret. After entering this as the secret I was given the password. The next password given was:

W0mMhUcRRnG8dcghE4qvk3JA9lGt8nDl

Level 9-10

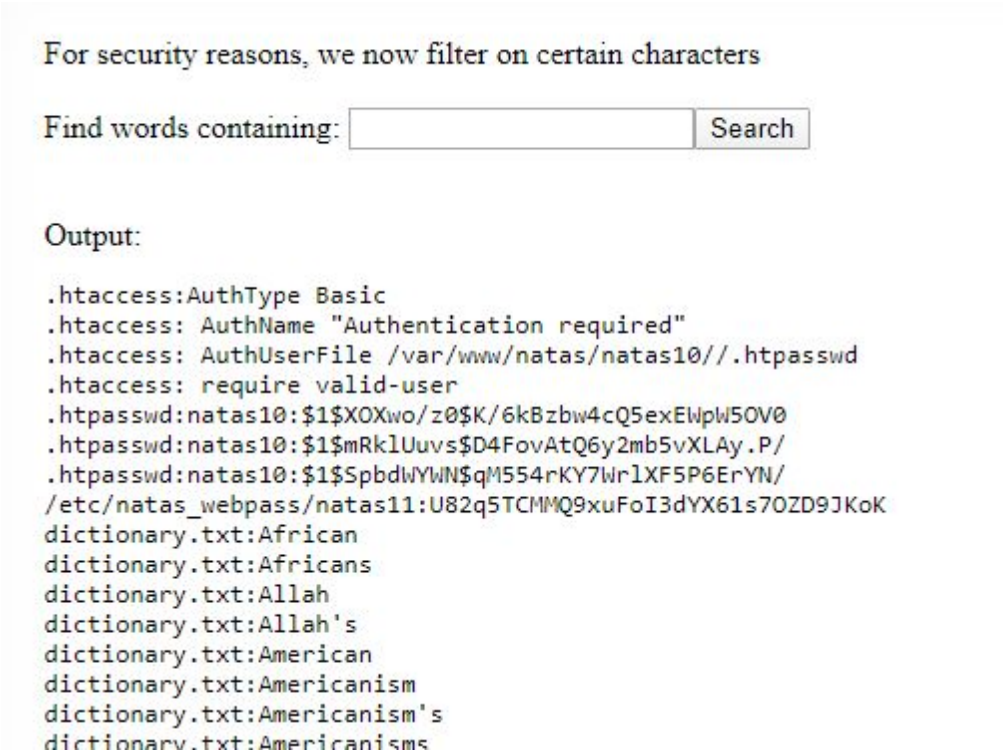
When opening this level, I was presented with a search box that seemed to search a dictionary for any word that starts with whatever I entered in the search box. Pressing the “View sourcecode” button, I could see exactly how the search worked. The search would work by taking the word entered, $key, and using that to search “dictionary.txt”. The command they used to do this was “passthru("grep -i $key dictionary.txt");”. Seeing this, I knew that I could alter what the command did and enter the password file by cutting off the command early and entering code that would display the password to me. All passwords are stored in “/etc/natas_webpass”, so I searched for the password by entering “; cat /etc/natas_webpass/natas10”. ‘;’ ended the command early and let me enter whatever code I wanted executed. Doing this resulted with the password as well as the entire dictionary. The next password given was:

nOpp1igQAkUzaI1GUUjzn1bFVj7xCNzu

Level 10-11

This level is similar to the previous one, but we cannot use ‘;’ to change the command like the last level. For this one, I had to check on multiple ways to use grep in order to pass a new command in the search box. Looking up on how to enter multiple fields in a grep command, I found a post that talked about using ‘.*’ in grep to find multiple patterns. The new search that I entered was: “.* /etc/natas_webpass/natas11”. Inserting this into the search resulted with the entire password file as well as the entire dictionary. The next password given was:

U82q5TCMMQ9xuFoI3dYX61s7OZD9JKoK



PortSwigger Attacks: Connor Radeloff

SQL Injection: (note, I spent a bit of extra time getting BurpSuite setup for these SQL challenges, but aside from that these took about 5-10 min each)

SQL injection UNION attack, determining the number of columns returned by the query:
I spent a bit of time searching for some search bar or login to enter queries into, but there was none, so I intercepted and modified the outgoing request to get the category, and unioned with NULL columns until the site returned successfully, about 10 min.

SQL injection UNION attack, finding a column containing text:
Same sort of attack as the previous attack, except we need to figure out which of the three columns contains text, so just make one of the columns a text parameter until one returns a good response, which turned out to be the second column of the three.

SQL injection UNION attack, retrieving data from other tables:

Same attack as the two above, and we were given the users table's schema, so it's just a simple matter of creating a union query to select the usernames and passwords. So, I added a simple union with select username, password from users like in the A Real Challenge section (which I completed before this section) and it returned a list of usernames and passwords.

SQL injection vulnerability in WHERE clause allowing retrieval of hidden data:
Just like in the A Real Challenge section, we can get all the products by adding an OR tag that's always true to return data we can't normally access, so I added an OR 1=1 to the query and got all the entries in the products table.

SQL injection vulnerability allowing login bypass:
The query behind this was most likely a simple database lookup and check, so i just set the username to administrator and in the request I sent I added an '-- to the end of the username parameter to close and comment out the rest of the request commenting out the password check in the database and allowing me to log in.

XSS:

Reflected XSS into HTML context with nothing encoded:
A quick search on the site shows the text of your request is put into the results field, and a quick test of adding a script tag with alert(1) displays an alert, which solves the challenge!

Reflected XSS into HTML context with most tags and attributes blocked:
This challenge is a bit more complex than the previous, as the site has a Web Application Firewall to make our lives harder. Trying the previous attack doesn't work, as the webpage will return a "Tag is not allowed" error. After trying some other common tags that execute javascript like img which failed, I decided to take a look at the hint for this challenge (given its title is "most tags and attributes blocked" it seems like there was a better way than blindly guessing) and this taught me about using BurpSuite's Intruder feature, which lets you test tags and attributes on those tags for xss vulnerabilities. After testing vulnerable tags and attributes, the <body> tag and "onresize" attribute are not blocked by the server, and a simple closing tag and adding a body tag with an alert(document.cookies) solved the challenge.

Stored XSS into HTML context with nothing encoded:
The challenge says there is a xss vulnerability in the comment functionality, so visiting a post and adding a script tag to the comment box and posting it makes it so when a user visits the post an alert is displayed. Hardest part of this challenge was trying to figure out the format for the Website field.

DOM XSS in document.write sink using source location.search:

After reading the portswigger write up on DOM XSS attacks, we see that when we type our parameters in search, we see that our search appears in the image src attribute, so we can close the current tag with an "> add another malicious image tag with an onerror attribute which will display an alert, solving the challenge.

DOM XSS in innerHTML sink using source location.search:
We can see that the function doSearchQuery sets the innerHTML attribute of the searchMessage span, so we can do another simple xss just by adding a script tag with alert to solve the challenge.

CSRF

CSRF vulnerability with no defenses:
After reading into what CSRF attacks are, we need to create a malicious HTML document that we can redirect users to and perform the attack. The lab description wants us to change the viewers email address, so we need to try and change our own email on the account to create an attack. After attempting to change the email and intercepting the request, we can build a CSRF attack with the information in the request, and upload our attack to the exploit server to solve the challenge. It's just like the example from the PortSwigger CSRF topic.

CSRF where token validation depends on request method:
In this challenge, we need to perform another CSRF attack, but if the CSRF parameter is different, then the server denies the request. After doing some more reading, many times the CSRF token validation is only done on POST methods, and not on GET methods. So, just like before, we can create a CSRF attack using the change email request and intercepting it, but this time we convert the request to a GET request and not a POST request. Uploading this GET version to the server completes the challenge.

A Real Challenge: Connor Radeloff

Session: 32e1bbc3-352b-4a7f-9e2f-0ed0f73642df

SQL Login:

For this challenge we were told to log in as a user, in my case, the user 'fry'. Because the challenge is called SQL Login and the only thing I'm given is a text box, this must be some sort of sql injection attack. I figured the query would look something like "select user from table where user= 'fry' and password = 'user-entered-password' " so I simply put a ' mark as the query to see if the field is vulnerable and got an incorrect password error and a response:

'SELECT username, password FROM users WHERE username=fry AND password="" - unrecognized token: """"

This gave me the exact query the site is using, so we know this is vulnerable to a SQL Injection attack! I tried adding another query on the end, and it gave me a SQLite 3 error about how it can only execute one query at a time. So, how can I get this query to return usernames and passwords? If you can add an OR query after the password = validation part, we can get this query to check if the password for fry is blank, OR all results. So, I input the following:
' OR '1'='1
And it worked! It returned all the usernames and passwords in the database. I could then enter the password "w4ts-w/-th3-17-dungbeetles" to complete the challenge.

Cross Site Scripting:

I see that the SEC for the image is directly what's given in the URL parameter, so maybe we can load a local website from this?
Insering https://i.redd.it/if7w9jpx1dt41.jpg"/> adds a tag to the webpage HTML, so the site is vulnerable to XSS. I tried putting: test" onerror="alert();" as the image input and the webpage alerts! Next I inserted the following: test" onerror="alert(document.cookies);" into the image, but no alert popped up. Strange. However, after inspecting the source again, it's clear that the "anti XSS technology" failed rather spectacularly. This is what the inspect element looked like: https://puu.sh/FyUTe/a887cc2105.png

Answer: flag{blo8/dword/default/@ck}

SQL Credit Cards:

Just like the first challenge, checked if the field is vulnerable to SQL Injection attacks by simple using ' as the entry, and it returned the following error:
'SELECT username FROM credit_cards WHERE username="' COLLATE NOCASE' - unrecognized token: "'" COLLATE NOCASE"
So now we have the query the backend uses. Great! There's a problem however, this query does not return the column that contains the credit card numbers, and we dont know its name. After doing a bit of googling, I found a SQLite query to return the table information of the entire database, which is as follows:
SELECT name, sql FROM sqlite_master

Because the query used on the backend only returns one column, we'll perform a UNION attack, and concatenate the results together into one column if we need to return more than one column.

After doing some googling on concatenating results, we modify the above command to get tables to fit into a single column:

' UNION SELECT name || '~' || sql FROM sqlite_master--

Which returned:

credit_cards~CREATE TABLE credit_cards (username text, card text, cvv integer, exp text)
users~CREATE TABLE users (username text, password text)

This command closes the username “where” clause, then selects all the table names and schemas of the tables. From the result, we can see that the column name we need is called card inside credit_cards, and it also has additional information associated with the card, but we don't need that data for this challenge.

So, we make another union attack as such:

' UNION SELECT username || '~' || card FROM credit_cards--

Which returned all the usernames and associated credit card numbers, completing the challenge!
CC number for scruffy: 4987327898009549

Stack Overflow:

So this problem seemed fairly simple at first, but after getting to work on it became clear this was a bit harder of a problem than I anticipated. Just to preface, I put the code provided into onlinegdb.com/online_c++_compiler to run and easily debug my inputs.

First, it's pretty easy to tell that this will be some sort of Buffer Overflow attack. Doing a little research will tell you that “gets()” is notoriously unsafe if you do some googling. gets() does not check the size of the variable you’re copying into, so you can easily buffer overflow variables you copy into. So, I initially thought this would be an easy task. Password and checkpass are both char[12] and declared one after the other, so all I thought I would have to do is type 24 characters in, like “aaaaaaaaaaaaaaaaaaaaaaaa”, that way both arrays would contain the same content, the strncmp would pass and the program would falsely authenticate. Performing this is the OnlingGdb confirmed my hypotheses and printed success!

However, entering this solution into the answer did not work, it segfaulted! I spent the next two hours frantically trying to figure out why the challenge website was failing but my code in the test environment worked... After I got desperate, I decided to check the console output of the

website, and it gave the flags used to build the program, which included the flag "-fno-stack-protector" which made my original buffer overflow exploit not work. Turns out, I should have taken the hint when it asks what the value of authenticated is, because when you overflow the buffer by just 8 bytes, authenticated increases to 97 from zero, and because the authenticated variable is not reset to zero if the password check fails, the program authenticated check passes.

So the answers I used were:
aaaaaaaaaaaaa (13 a's)
97

Extra Credit: Go Further

Linux Exploits (bandit continued): Zack McKevitt

Level 26-27

The hard part of this challenge is logging into the shell, since it is not a /bin/bash shell at first. Once we are in, it is quite easy to get the password for bandit27. We can see the file 'bandit27-do', which is a filetype we have seen in previous challenges that allows users to change their uid. So, I changed the current uid from bandit26 to bandit27, and viewed the password in the file /etc/bandit_pass/bandit27. The commands and password are below.

./bandit27-do uid=11027 cat /etc/bandit_pass/bandit27

3ba3118a22e93127a4ed485be72ef5ea

5 minutes

Level 27-28

For this level, I had to clone a git repo on a remote machine. The game gave me the ssh info and path to the file, so I was able to easily clone this remote repository to a local directory. I did this with the following command:

git clone bandit27-git@localhost:/home/bandit27-git/repo

After examining its file contents on my local machine, I got the password:

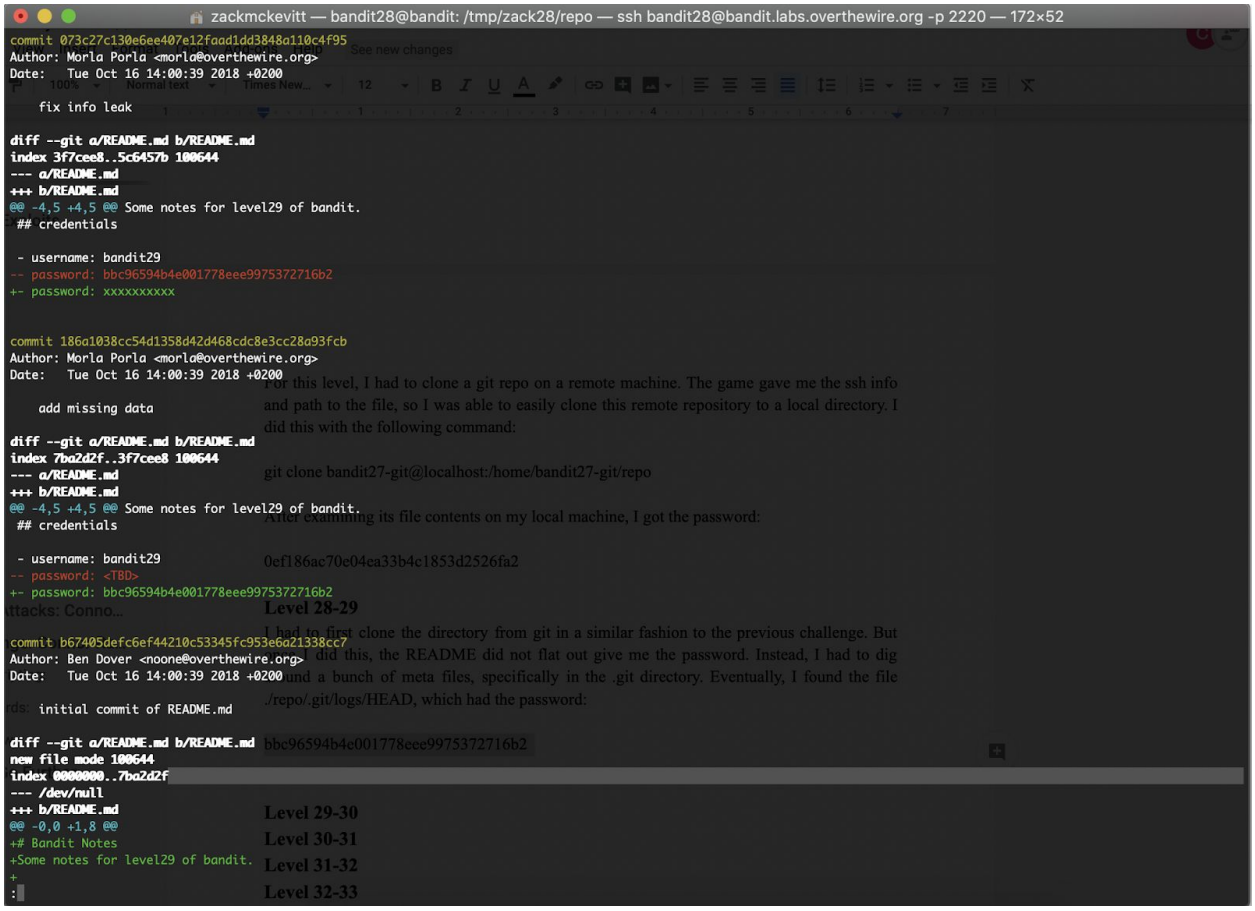
0ef186ac70e04ea33b4c1853d2526fa2

5 minutes

Level 28-29

I had to first clone the directory from git in a similar fashion to the previous challenge. But once I did this, the README did not flat out give me the password. Instead, I had to look at the git logs. At first, I was poking around metafiles and also saw long encrypted strings, which I thought were the password. However, these were not passwords, but commit numbers, and I examined the git logs instead. Inside, I found the password:
Bbc96594b4e001778eee9975372716b2

20 minutes



Level 29-30

For this level, I had to clone another git repo. This time, I had to use the ‘git diff’ command to find the password after failed attempts at checking the logs. I first checked the ‘HEAD’ and ‘master’ files, but had no results. Next, I tried ‘git diff origin/dev’, which contained the password:

5b90576bedb2cc04c86a9e924ce42faf

10 minutes

Level 30-31

In this git repository, the README was not at all helpful in obtaining the password. While I was playing around with different git commands, I came across a file called secret. I then used ‘git show secret’ to reveal the password:

47e603bb428404d265f59c42920d81e5

5 minutes

Level 31-32

This level required me to push a key file to the repository. When I checked the README, it told me the filename must be ‘key.txt’ and the contents must be ‘May I come in?’. I created this file and wrote to it using the ‘touch’ and ‘echo’ commands. I also realized that there was a .gitignore file, which ignored ‘.txt’ files, so I removed that file as well. I then added ‘key.txt’ and pushed my code to the master branch. I then received the password for the next level:

56a9bf19c63d650ce78e6ec0354ee45e

230+ minutes

Level 32-33

This level had a shell that only interpreted uppercase commands. This is annoying, and we wanted to get to a bash shell, so I entered the command ‘$0’ which invokes a bash shell. From there, I tried entering the /etc/bandit_pass/bandit33 file which actually worked! It gave the following password:

c9c3199ddf4121b10cf581a98d51caee

5 minutes

Level 33-34

Bandit Level 33 → Level 34

At this moment, level 34 does not exist yet.

:(

Web Exploits (natas continued): Sahib Bajwa

Level 11-12

This level took a very long time to figure out. Upon login we are greeted with a message that says “Cookies are protected with XOR encryption”. Other than that we have a text box that we can use to submit color references. Looking into the page’s html code results with nothing. The next step was to press the button for viewing the source code. Taking a look at the source code, I can see the xor_encrypt function and how it takes a key and uses that to encrypt the $in. Looking into this function, the function uses a $key that we cannot see to encrypt the string. Above this, I can also see the $defaultdata array that has “showpassword” set to “no” and “bgcolor” set to “#ffffff”. This is similar to the text box on the landing page. Next, I can see a function called loadData that seems to mess around with a cookie ($global $_COOKIE; is the first line in the function). The saveDAta function below sets the cookie to the encrypted string. Going back to the landing page, I inspect the cookies and see a cookie with the name “data” and a value “ClVLIh4ASCsCBE8lAxMacFMZV2hdVVotEhhUJQNVAmhSEV4sFxFeaAw%3D”. I assume that the value is the encrypted text from the xor_encrypt function. Seeing all of these things, I understand that I might need to figure out the $key used in the xor_encrypt function in order to change “showpassword” to yes. To figure out this key, I need to do the reverse operation to the value found in the cookie. In the saveData function, I can see a base64_encode(xor_encrypt(json_encode($d))));. My first step was to then run a base64_decode on the value found in the cookie using php. Doing so resulted with the string: “UK"H+O%pSWh]UZ-T%UhR^,^h7”. I then copied over the entire xor_exrypt function into my php code. To continue reversing the function, I had to place the “json_encode($d)” into the key and change the “$d” to “"showpassword"=>"no", "bgcolor"=>"#ffffff"” because that was in the original code as the $defaultdata array. I then ran the code with the previously mentioned json_encode() set to the $key and xor_encrypted the previously base64_decoded cookie. This gave me “qw8Jqw8Jqw8Jqw8Jqw8Jqw8Jqw8Jqw8Jqw8JqL”. Using this as the $key and setting $text to “json_encode(array("showpassword"=>"yes", "bgcolor"=>"#ffffff"), I ran the

function and base64_encoded the output of xor_encrypt (also note I had to remove $in from the input or the function would crash). The value I recieved when running the function was: “ClVLIh4ASCsCBE8lAxMacFMOXTlTWxooFhRXJh4FGnBTVF4sFxFeLFMx”. I then took this value and replaced the cookie on the landing page from earlier with this value. Pressing “Set color” on the home page resulted with the next password. The next password given was:

EDXp0pS26wLKHZy1rDBPUZk0RKfLGIR3

Level 12-13

This level was much (very much) simpler than the last level (to me). Upon getting into level 12, we are told to choose a jpeg to upload and we have two further options. The first is to choose a file and the second is to upload the file. Choosing the file allows me to upload a file directly from my computer. My first thought was that this is where the vulnerability for this level would come into play. Inspecting the pages html code resulted with nothing useful. The next step was to look into the source code. Going through the functions, the first function simply generated a random string. The second function made a random path in the directory. The third function made a random path from a filename in the directory. The next section in the source code was not a function but an if statement. This if statement dictated what would happen to a file if it was successfully uploaded or if it could not be uploaded. There was a section in this if statement that checked to see if the file was too big, otherwise it tried to upload the file. Since it otherwise tried to upload the file, I could upload a file that would contain any code I wanted as long as it is under the maximum file size. I then created a php file that I could upload that would access the pages password file, /etc/natas_webpass/, and give me the next password. The php file I created was pretty basic, it contained one line that read:

readfile('/etc/natas_webpass/natas13');

Note, I did try to get this to work with fopen, but did not end up figuring it out.

I expected the password to get displayed after doing this, but nothing happened. The file was uploaded, but got changed to a .jpg file due to the first three functions generating a random name for the file. To change the type of file being uploaded, I used Burp Suite to see and edit the in between process. In Burp Suite, I can see this:

```
-----------------------------163767420834589823385465942 74
Content-Disposition: form-data; name="filename"

hlpdf4cyjz.jpg
-----------------------------163767420834589823385465942 74
```

I can see in this request that the file name is going to be “hlpdf4cyjz.jpg”. I changed the “.jpg” to “.php” and forward the request on. The file is then uploaded successfully as a php file. Opening the file results with the next password. The next password given was:

jmLTY0qiPZBbaKc9341cqPQZBJv7MQbY

Here is me being successfully logged-in to level 13: