OverTheWire's Bandit

March 29th 2018

Last updated: January 12th 2021 (Formatting)

Play here: https://overthewire.org/wargames/bandit/

Writeup

Level 0-1:

The first level only asks us to SSH into the bandit0 user on the labs' server. This can be done using the OpenSSH utility:

```
 ssh bandit0@bandit.labs.overthewire.org -p 2220
cat readme
```

level 1-2:

This level is very similar, but instead of the flag being in the `readme` file, it's in a file called `-`, which we cannot directly cat, because bash interprets the `-` character as ssh into bandit1@bandit.labs.overthewire.org -p 2220 cat ./- password is found

level 2-3

`cat spaces\ in\ this\ filename` or `cat "space in this filename"`

level 3-4:

ssh into bandit2@bandit.labs.overthewire.org -p 2220

```
 cd inhere/
ls -la
```

We see there is hidden file.

```
 cat .hidden
```

level 4-5:

If we navigate to the inhere/ directory. There are a bunch of files. Of course we could check them one by one, but we don't learn anything by doing that. Instead, we can try to find which file to look for inparticular. We're probably looking for something that contains some readable ASCII text. `file` might be useful here for figuring out the contents of a file.

Notice also that each file is starting with a `-`, just like in the first challenge. We can bypass this by either using the `./` or just using `--` before running our filenames, so that the interpreter does not consider it as input.

```
 file -- -*
```

Returns:

```
 -file00: data
-file01: data
-file02: data
-file03: data
-file04: data
-file05: data
-file06: data
-file07: ASCII text
-file08: data
-file09: data
```

The seventh file thus inevitably contains the flag.

level 5-6

When going into the `inhere/` directory, we first see a bunch of folders all titled `maybehere##`, with ## ranging from 0 to 20. If we go into `maybehere00`, we see a bunch of files as such:

```
drwxr-x---  2 root bandit5 4096 May  7  2020 .
drwxr-x--- 22 root bandit5 4096 May  7  2020 ..
-rwxr-x---  1 root bandit5 1039 May  7  2020 -file1
-rwxr-x---  1 root bandit5  551 May  7  2020 .file1
-rw-r-----  1 root bandit5 9388 May  7  2020 -file2
-rw-r-----  1 root bandit5 7836 May  7  2020 .file2
-rwxr-x---  1 root bandit5 7378 May  7  2020 -file3
-rwxr-x---  1 root bandit5 4802 May  7  2020 .file3
-rwxr-x---  1 root bandit5 6118 May  7  2020 spaces file1
-rw-r-----  1 root bandit5 6850 May  7  2020 spaces file2
-rwxr-x---  1 root bandit5 1915 May  7  2020 spaces file3
```

If we use the `file -- * .file*` We can see what each of them contain:

```
-file1:       ASCII text, with very long lines
-file2:       ASCII text, with very long lines
-file3:       PGP\011Secret Key -
spaces file1: ASCII text, with very long lines
spaces file2: ASCII text, with very long lines
spaces file3: data
.file1: ASCII text, with very long lines
.file2: ASCII text, with very long lines
.file3: data
```

This pattern is repeated for all the other directories. Judging from the previous flags, we are looking for a file that contains `ASCII text` This would take an absurd amount of time to do manually. Fortunately, anything is possible in bash:

In the `inhere/` directory running this will show all of the files and grep them to only show the files containing `ASCII text`.

```
for f in *; do file $f/* $f/.file*; done | grep -vie "ascii text, with very long lines" | grep -vie "data"
```

This however, did not yield any flags, because after checking through the 6 or so files that validated our conditions, none of them contained a valid flag. After checking back on the previous flag formats, I established that they were all going to be 33 characters long.

So instead of searching for ASCII Files, let's search for files that contain 33 characters. In the `inhere/` directory:

```
for d in */; do for f in $d*; do wc "$f"; done; for f in $d.file*; do wc $f; done; done | grep 33;
```

Yields the following:

```
   1    1 3385 maybehere03/spaces file2
  20   95 3362 maybehere07/-file3
   1    1 1033 maybehere07/.file2
   1    1 1133 maybehere17/-file1
   1    1 3387 maybehere17/spaces file2
   1    1 7334 maybehere18/spaces file1
```

Only two of these are interesting to me, the `maybehere07/.file2` and `maybehere17/-file1`. Sure enough, the flag was inside the first, as well as 1000 spaces to make it sneak past.

Level 6-7

For this level, we're looking for a file that meets certain conditions. Prompt:

Flag is located in a file that respects the following criterias: - owned by user bandit7 - owned by group bandit6 - 33 bytes in size

Immediately, we can think of the `find` command, but if this was not evident, some commands are also given at the level page.

```
find / -readable -user bandit7 -group bandit6 -size 33c 2>/dev/null
```

- `-readable` filters for all the readable files
- `-user bandit7` looks for files owned by the user bandit7
- `-group bandit6` looks for files that belongs to group bandit6
- `-size 33c` looks for a file that is 33 bytes long
- `2>/dev/null` will redirect standard error (2) to /dev/null. It will essentially remove any unwanted output onto the console.

This will leave us with one file that contains the flag.

Level 7-8

This one is very simple. Using grep with or without any flags will give us the output we are looking for:

```
cat data.txt | grep -ie millionth
```

Level 8-9

Here we are looking for the only line of text that occurs only once. We can make use of the `sort` and `uniq` commands for that.

```
cat data.txt | sort -bh | uniq -u
```

- `sort -bh` will return all the lines in human-readable numeric sort while also ignore leading blanks
- `uniq -u` will only return unique lines.

Level 9-10

We're looking for the password in the data.txt file once again, but this time it's the only line that is a human readable string and that is preceded by a bunch of = signs. Human-readable in a file of a bunch of non human-readable strings will immediately make us think of the `strings` command.

Running the following:

```
cat data.txt | strings | grep -ie =====
```

Will return a few outputs, but only one of which has a flag-like 33 character string.

Level 10-11

Another very simple one, if you've done other CTFs or puzzles, base64 is a very popular way to encode data. Fortunately, it's also very easy to decode too.

The `base64 -d` command will decode base64 input (given the -d flag of course). You could also use an online base64 decoder.

```
cat data.txt | base64 -d
```

Level 11-12

The prompt for this one is similar to the previous level. We must transform the output of the data.txt file into the format that we want. In this case the letters were shifted 13 characters, such that it's a ROT13 transformation. We can use an online converter or run the `tr` command:

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

The tr command takes the output and **transforms** it, using the various given sets as arguments to the command. In this case, the first set contains all of the letters, meaning we want to operate on all letters of the output of data.txt. The second set states that we want to replace all letters of the first output with N-ZA-M and the same for the lowercase letters. For example, if we have "ABC", it will be changed for a "NMP", shifting the letter 13 counts forward. Likewise if we have "NMP" instead.

Level 12-13

This level is very common in beginner CTFs. It's also very good for practicing your archive extraction speed on the command line!

The first thing to do is to create your own directory under the `/tmp` directory. This will be useful to create files. After that, copy the data.txt file under `~` and paste it in the newly created directory. The data.txt file contains a hexdump, as stated by the prompt. The `xxd` allows us to revert a hexdump into its original state with the `-r` flag.

```
xxd -r <filename> <output-file>
```

This will give us new file and when inspecting it with the `file` command, it is stated that it's a gzip archive. We thus have to rename it to `<filename>.gz`. This will need to be the case for the rest of the decompressions as well.

There are a total of 9 decompressions to do. I will not go through each of them, but here is the gist of it:

- If the compression format is POSIX tar, use the following:

```
tar -xf <filename>
```

- If the compression format is bzip2, use the following:

```
bunzip2 <filename>
```

- If the compression format is gzip, use the following:

```
gunzip <filename>
```

It's also important to know that the suffix used for the filenames must be changed to correspond to its correct format.

In the end, there will be a total of 10 extracted files:

```
 bandit12@bandit:/tmp/strixsc$ ls -la
total 908
drwxr-sr-x    2 bandit12 root   4096 Jan 24 21:27 .
drwxrws-wt 4649 root     root 847872 Jan 24 21:27 ..
-rw-r--r--    1 bandit12 root     49 Jan 24 21:27 data-10.txt
-rw-r-----    1 bandit12 root   2582 Jan 24 21:10 data-1.txt
-rw-r--r--    1 bandit12 root    606 Jan 24 21:11 data-2.gz
-rw-r--r--    1 bandit12 root    573 Jan 24 21:16 data-3.bz2
-rw-r--r--    1 bandit12 root    431 Jan 24 21:11 data-4.gz
-rw-r--r--    1 bandit12 root  20480 Jan 24 21:19 data-5.tar
-rw-r--r--    1 bandit12 root  10240 May  7  2020 data-6.tar
-rw-r--r--    1 bandit12 root    222 May  7  2020 data-7.bz2
-rw-r--r--    1 bandit12 root  10240 Jan 24 21:26 data-8.tar
-rw-r--r--    1 bandit12 root     79 May  7  2020 data-9.gz
```

The flag is in the last ASCII Text file.

Level 13-14

For this level, we don't need to find a flag. A private key file is listed and the contents must be copied to be used to log into the bandit14 user.

After copying the files, put it in a file and use that file to ssh into the bandit14 user.

```
 chmod 0400 <private-key-file>
ssh -i <private-key-file> bandit14@bandit.labs.overthewire.org -p 2220
```

The `-i` uses an identity file to authentify. The `chmod 0400` is often necessary, since if the permissions on the private key file are too broad, and thus OpenSSH will ignore that file.

Once in, the password of bandit14 is in `/etc/bandit_pass/bandit14`

Level 14-15

We now just need to send a packet to the specified port on localhost, as stated by the prompt. The `nc` command will allow us to do that:

```
cat /etc/bandit_pass/bandit14 | nc localhost 30000
```

Level 15-16

For this level, we need to send the bandit15 password to localhost at port 30001 using SSL encryption. The `OpenSSL s_client` utility is very useful for that:

```
$ openssl s_client -connect :30001
```

Then just send the password and you'll receive the next level's password back.

Level 16-17

The level's prompt is very specific. We need to scan the localhost to find open ports between 31000 and 32000. Then we need to see if any of them accept SSL. The only one that does will have the password for the next level once we submit the current level's password to it.

Use your previously created directory under tmp to create an nmap directory. Then run nmap for an initial scan.

```
 mkdir /tmp/<dirname>/nmap
nmap -sC -sV -oA /tmp/<dirname>/nmap/ -p31000-32000 localhost
```

The nmap flags are important for accurately indentifying the services behind each ports.

- `-sC` will use default scripts (-sC: equivalent to --script=default)
- `-sV` will probe open ports to determine service/version info
- `-oA` will output all formats to the given directory (This is mostly important if the nmap is a wide scan. Usually scanning takes a long time and it's necessary to keep the information instead of having to rescan.)
- `-p31000-32000` will scan all ports in the given range

This yields the following:

```
bandit16@bandit:/tmp/strixsc16/nmap$ nmap -sC -sV -p31000-32000 -oA /tmp/strixsc16/nmap/ localhost

Starting Nmap 7.40 ( https://nmap.org ) at 2021-01-24 22:50 CET
Stats: 0:01:01 elapsed; 0 hosts completed (1 up), 1 undergoing Service Scan
Service scan Timing: About 80.00% done; ETC: 22:51 (0:00:15 remaining)
Nmap scan report for localhost (127.0.0.1)
Host is up (0.00048s latency).
Not shown: 996 closed ports
PORT      STATE SERVICE     VERSION
31046/tcp open  echo
31518/tcp open  ssl/echo
| ssl-cert: Subject: commonName=localhost
| Subject Alternative Name: DNS:localhost
| Not valid before: 2020-11-04T15:24:27
|_Not valid after:  2021-11-04T15:24:27
|_ssl-date: TLS randomness does not represent time
31691/tcp open  echo
31790/tcp open  ssl/unknown
| fingerprint-strings:
|   FourOhFourRequest, GenericLines, GetRequest, HTTPOptions, Help, Kerberos, LDAPSearchReq, LPDString, RTSPRequest, SIPOptions, SSL
|_    Wrong! Please enter the correct current password
| ssl-cert: Subject: commonName=localhost
| Subject Alternative Name: DNS:localhost
| Not valid before: 2020-12-03T12:25:02
|_Not valid after:  2021-12-03T12:25:02
|_ssl-date: TLS randomness does not represent time
31960/tcp open  echo
1 service unrecognized despite returning data. If you know the service/version, please submit the following fingerprint at https://n
SF-Port31790-TCP:V=7.40%T=SSL%I=7%D=1/24%Time=600DEBB3%P=x86_64-pc-linux-g
SF:nu%r(GenericLines,31,"Wrong!\x20Please\x20enter\x20the\x20correct\x20cu
SF:rrent\x20password\n")%r(GetRequest,31,"Wrong!\x20Please\x20enter\x20the
SF:\x20correct\x20current\x20password\n")%r(HTTPOptions,31,"Wrong!\x20Plea
SF:se\x20enter\x20the\x20correct\x20current\x20password\n")%r(RTSPRequest,
SF:31,"Wrong!\x20Please\x20enter\x20the\x20correct\x20current\x20password\
SF:n")%r(Help,31,"Wrong!\x20Please\x20enter\x20the\x20correct\x20current\x
SF:20password\n")%r(SSLSessionReq,31,"Wrong!\x20Please\x20enter\x20the\x20
SF:correct\x20current\x20password\n")%r(TLSSessionReq,31,"Wrong!\x20Please
SF:\x20enter\x20the\x20correct\x20current\x20password\n")%r(Kerberos,31,"W
SF:rong!\x20Please\x20enter\x20the\x20correct\x20current\x20password\n")%r
SF:(FourOhFourRequest,31,"Wrong!\x20Please\x20enter\x20the\x20correct\x20c
SF:urrent\x20password\n")%r(LPDString,31,"Wrong!\x20Please\x20enter\x20the
SF:\x20correct\x20current\x20password\n")%r(LDAPSearchReq,31,"Wrong!\x20Pl
SF:ease\x20enter\x20the\x20correct\x20current\x20password\n")%r(SIPOptions
SF:,31,"Wrong!\x20Please\x20enter\x20the\x20correct\x20current\x20password
SF:\n");

Service detection performed. Please report any incorrect results at https://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 92.21 seconds
```

The nmap result shows that two ports are using ssl. One of them is using `ssl/unknown` and the other `ssl/echo`. However, using the `-sC` flag quickly gives us the correct port to send the password to: 31790.

We can now send the password of the current level to the next using the same command as the one used for the previous level.

```
openssl s_client -connect :31790
```

Then submitting the current level's flag.

Level 17-18

To connect to level 17, we have to use the private key that we received from the previous level. Once that's done, we can connect to the bandit17 user and retrieve the password from `/etc/bandit_pass/bandit17`.

For this level, we are required to find the one different line between two files `passwords.new` and `passwords.old`. I did this in two different ways, and both are good methods:

- First method:

  ```
  cat passwords.old passwords.new | sort -bh | uniq -u
  ```

  This will output all the contents of both files, sort them human-numerically and only output unique lines. This gives us two outputs, one of which is the flag.

- Second method:

  ```
  diff passwords.old passwords.new
  ```

  The `diff` utility compares the given files and outputs the differences. This will give us the same two outputs as the first method.

Level 18-19

When connecting to the bandit18 user, we are instantly kicked out. We thus need to find a way to get the password without being directly connected. Thankfully, ssh allows us to send commands to execute on connection, and since we know the flag is in a file called readme in the home directory:

```
echo "cat readme" | ssh bandit18@bandit.labs.overthewire.org -p 2220
```

Level 19-20

This level is pretty straight forward, we are given a binary that executes a command as the bandit20 user. Knowing that each user can access their own passwords at /etc/bandit_pass/bandit##, with ## being the respective user's number, we can simply execute the binary as so:

```
./bandit20-do cat /etc/bandit_pass/bandit20
```

Level 20-21

For this level, we are given a binary that when executed makes a connection to localhost on the port you specify as a commandline argument. It will return the bandit21 password if we provide the bandit20 password to the port it's listening to.

There are a few ways of doing this. We could execute the binary and keep it running in the background, while sending the password with netcat on the same pane. However if there happens to be information that is outputed from the binary, it will be a hassle to keep toggling back to it. We could use two different ssh connections, but that also takes time. The right way is to use a terminal multiplexer, such as tmux.

Running tmux will allow us to have multiple bashs openned on the same terminal window.

In the first window, run the binary:

```
./suconnect 30000
```

In the second window, send the password with netcat and this will return the correct password for bandit21:

```
cat /etc/bandit_pass/bandit20 | nc localhost 30000
```

Level 21-22

Prompt: A program is running automatically at regular intervals from cron, the time-based job scheduler. Look in /etc/cron.d/ for the configuration and see what command is being executed.

This is a very useful challenge, because it introduces you to cronjobs, which are very popular and useful when using linux.

The prompt states that there is a cronjob that we should take a look at. Going into `/etc/cron.d`, we see that user bandit22 has a cronjob setup:

```
 bandit21@bandit:/etc/cron.d$ cat cronjob_bandit22
@reboot bandit22 /usr/bin/cronjob_bandit22.sh &> /dev/null
* * * * * bandit22 /usr/bin/cronjob_bandit22.sh &> /dev/null
```

It would seem that the cronjob executes `/usr/bin/cronjob_bandit22.sh` every minute (`* * * * *`). We can then take a look at that script and see what it does:

```
 bandit21@bandit:/etc/cron.d$ cat /usr/bin/cronjob_bandit22.sh
#!/bin/bash
chmod 644 /tmp/t7O6lds9S0RqQh9aMcz6ShpAoZKF7fgv
cat /etc/bandit_pass/bandit22 > /tmp/t7O6lds9S0RqQh9aMcz6ShpAoZKF7fgv
```

It would seem that it overwrites the content of the `/tmp/t7O6lds9S0RqQh9aMcz6ShpAoZKF7fgv` file with the contents of the `/etc/bandit_pass/bandit21` file. It also changes the permissions of the file and sets it to 644, which means that is is readable. Printing the file we can get the password for the next level.

Level 22-23

A program is running automatically at regular intervals from cron, the time-based job scheduler. Look in /etc/cron.d/ for the configuration and see what command is being executed.

Once again, we're dealing with cronjobs.

Going into `/etc/cron.d`, we see that user bandit23 has a cronjob setup:

```
 bandit22@bandit:/etc/cron.d$ cat cronjob_bandit23
@reboot bandit23 /usr/bin/cronjob_bandit23.sh  &> /dev/null
* * * * * bandit23 /usr/bin/cronjob_bandit23.sh  &> /dev/null
```

It would seem that the cronjob executes `/usr/bin/cronjob_bandit23.sh` every minute (`* * * * *`). We can then take a look at that script and see what it does:

```
 #!/bin/bash

myname=$(whoami)
mytarget=$(echo I am user $myname | md5sum | cut -d ' ' -f 1)

echo "Copying passwordfile /etc/bandit_pass/$myname to /tmp/$mytarget"

cat /etc/bandit_pass/$myname > /tmp/$mytarget
```

The script creates a file under the `/tmp` with a special name based on the user executing the script. We can find out the name of the file if we manually execute the command providing the mytarget variable value. The $myname variable represents the user that is executing the script. In this case it would be bandit23. We can manually replace it in the command below:

```
 echo "I am user bandit23" | md5sum | cut -d ' ' -f 1
```

This returns `8ca319486bfbbc3663ea0fbe81326349`, which if we print out `/tmp/8ca319486bfbbc3663ea0fbe81326349`, we will get the password for the next level.

Level 23-24

Once again, we're dealing with cronjobs and shell scripts. We can go see the cronjob by going to the `/etc/cron.d/` directory and printing the contents of the `cronjob_bandit24` file:

```
 bandit23@bandit:/etc/cron.d$ cat cronjob_bandit24
@reboot bandit24 /usr/bin/cronjob_bandit24.sh &> /dev/null
* * * * * bandit24 /usr/bin/cronjob_bandit24.sh &> /dev/null
```

We can see the script that is being ran too:

```
 #!/bin/bash

myname=$(whoami)

cd /var/spool/$myname
echo "Executing and deleting all scripts in /var/spool/$myname:"
for i in * .*;
do
    if [ "$i" != "." -a "$i" != ".." ];
    then
        echo "Handling $i"
        owner="$(stat --format "%U" ./$i)"
        if [ "${owner}" = "bandit23" ]; then
            timeout -s 9 60 ./$i
        fi
        rm -f ./$i
    fi
done
```

If we manually go through this:

`myname` will become `bandit24`

It script then goes to /var/spool/bandit24 and deletes everything in the directory, but before that executes it. Which is where we can exploit the cronjob.

```
owner="$(stat --format "%U" ./$i)"
if [ "${owner}" = "bandit23" ]; then
    timeout -s 9 60 ./$i
```

In this section, it will check if the owner of the script is bandit23, if so, then it will timeout for 60s with signal 9 then execute the script.

Thus, in order to get the flag, we need to create a bash script that will execute and send us the contents of the `/etc/bandit_pass/bandit24` file into a file under a directory that we create.

So let's first create a new directory under the `/tmp/` that we can use to create a new file.

After that, we can use any text editor to create a new script:

```
#!/bin/bash

cat /etc/bandit_pass/bandit24 &>> /tmp/<directory>/bandit24
```

The script is very simple, since the bandit24 user will execute all scripts in the `/var/spool/bandit24/` directory, we just have to make it read the `/etc/bandit_pass/bandit24` and output the content into a file inside of the newly created directory under `/tmp`.

The important thing to do here is to change the permissions of the both the output file and the script, so that the bandit24 user can execute it and then is allowed to write into the output file.

```
chmod +rw <output-file>
chmod +rwx <script-name>
```

Then just copy the script into the `/var/spool/bandit24/` directory

```
cp <script-name> /var/spool/bandit24/<script-name>
```

We can then monitor the contents of the output file with an infinite while loop:

```
while true; do cat <output-file>; done
```

And after a minute, the output file should contain the password.

Level 24-25

Prompt: A daemon is listening on port 30002 and will give you the password for bandit25 if given the password for bandit24 and a secret numeric 4-digit pincode. There is no way to retrieve the pincode except by going through all of the 10000 combinations, called brute-forcing.

We thus need to send 10000 different strings to the listener. We could do this in a number of ways, ideally you could do it with a bash script, but it's not necessary, as a one liner in the command line will do just fine:

```
for i in {0000..9999}; do echo "UoMYTrfrBFHyQXmg6gzctqAwOmw1IohZ $i"; done | nc localhost 30002
```

Executing this as bandit24 will give an output that looks like this: h

```
[...]
Wrong! Please enter the correct pincode. Try again.
Wrong! Please enter the correct pincode. Try again.
Wrong! Please enter the correct pincode. Try again.
Wrong! Please enter the correct pincode. Try again.
Wrong! Please enter the correct pincode. Try again.
Wrong! Please enter the correct pincode. Try again.
Wrong! Please enter the correct pincode. Try again.
Wrong! Please enter the correct pincode. Try again.
Wrong! Please enter the correct pincode. Try again.
Wrong! Please enter the correct pincode. Try again.
Wrong! Please enter the correct pincode. Try again.
Wrong! Please enter the correct pincode. Try again.
Wrong! Please enter the correct pincode. Try again.
Correct!
The password of user bandit25 is uNG9O58gUE7snukf3bvZ0rxhtnjzSGzG
```

Level 25-26

Sidenote: This level was the one that took me the longuest to figure out. I would've never figured it out had it not been for the list of commands that the challenge page suggests.

Prompt: Logging in to bandit26 from bandit25 should be fairly easy… The shell for user bandit26 is not /bin/bash, but something else. Find out what it is, how it works and how to break out of it.

So, when logging into bandit25 and listing the files in the home directory, we are met with `bandit26.sshkey`, containing the necessary RSA Private Key to ssh into bandit26@localhost.

```
ssh -i bandit26.sshkey bandit26@localhost
```

The following is then outputted:

```
[...]
--[ More information ]--

  For more information regarding individual wargames, visit
  http://www.overthewire.org/wargames/

  For support, questions or comments, contact us through IRC on
  irc.overthewire.org #wargames.

  Enjoy your stay!


  _                     _ _ _   ___   __
 | |                   | (_) | |__ \ / /
 | |__   __ _ _ __   __| |_| |_   ) / /_
 | '_ \ / _` | '_ \ / _` | | __| / / '_ \
 | |_) | (_| | | | | (_| | | |_ / /| (_) |
 |_.__/ \__,_|_| |_|\__,_|_|\__|____\___/
Connection to localhost closed.
```

So we are getting successfully authentified, but we are getting kicked out the second we log in. It does mention in the prompt that the shell is not bash. This is a good hint, so lets go see exactly what we are getting prompted with on login. We can do that by looking at the `/etc/passwd` file. The last line of the `passwd` file gives you the commandline/shell to prompt the user with on login. Of course, this doesn't have to be a shell, it could be any script/binary.

In our case:

```
bandit25:x:11025:11025:bandit level 25:/home/bandit25:/bin/bash
bandit26:x:11026:11026:bandit level 26:/home/bandit26:/usr/bin/showtext
bandit27:x:11027:11027:bandit level 27:/home/bandit27:/bin/bash
```

It's a script called `showtext`.

Let's see what it is:

```
#!/bin/sh

export TERM=linux

more ~/text.txt
exit 0
```

There we go, something we can work with. Here is where the `more` command really proved useful and I would've never thought about it had it not been for the list of commands that the challenge page suggests. This is a good thing to keep in mind: always read your challenge pages thoroughly.

The `more` command works a little bit like the `less` command in that it prints the text, but takes into account the screen realestate. When using `less` to show the contents of a file, we can navigate through the output incrementally. The `more` command allows us to do the same thing, but only if the screen is too small to print the whole text. In my case, this *took a really long time to figure out*, because I am using a window manager where I don't really have window sizes. After a lot of tinkering with different things, I figured out how to lower the screen size by creating a bunch of smaller panes using tmux. This allowed me to get the following output:

```
  _                     _ _ _   ___   __
 | |                   | (_) | |__ \ / /
 | |__   __ _ _ __   __| |_| |_   ) / /_
--More--(50%)****
```

Finally, I was able to navigate. This is necessary because we thus have the ability to use commands. `more` has this great feature where it allows you to go into editor mode. The editor it uses is `vi`, and if you have any experience with vi, you know that you can execute commands from within the editor. We can go into the editor mode by pressing "v" when in `more`. In editor mode, we can use

```
: vi /etc/bandit_pass/bandit26
```

To finally get the password for the user.

This was by far the most original challenge in the Bandit wargame, and one of the most original out there too.

Level 26-27

For this level, we need to do the same hacky thing with the `more` command. ssh into the bandit26 user with minimal space on the screen to trigger the allowance of visual mode. Trigger the editor mode and now you're able to input commands.

In vi, we can execute commands by inputing `:!` in `Normal` mode. Of course, this uses the default user shell. For this case, the default shell was `showtext`. We thus need to first set the default shell that vi will use. In `Normal` mode, use:

```
:set shell=/bin/bash
```

Then, we can finally run commands through vi. Listing the home directory contents we see the following:

```
 drwxr-xr-x  3 root     root     4096 May  7  2020 .
drwxr-xr-x 41 root     root     4096 May  7  2020 ..
-rwsr-x---  1 bandit27 bandit26 7296 May  7  2020 bandit27-do
-rw-r--r--  1 root     root      220 May 15  2017 .bash_logout
-rw-r--r--  1 root     root     3526 May 15  2017 .bashrc
-rw-r--r--  1 root     root      675 May 15  2017 .profile
drwxr-xr-x  2 root     root     4096 May  7  2020 .ssh
-rw-r-----  1 bandit26 bandit26  258 May  7  2020 text.txt
```

There is a file `bandit27-do` that will allow us to run a command as the bandit27 user. Evidently, this will be used to print the contents of /etc/bandit_pass/bandit27.

To do that, run:

```
:! ./bandit27-do cat "/etc/bandit_pass/bandit27"
```

And we'll have the password for bandit27.

Level 27-28:

This level is very simple compared to the other ones. The prompt simply says that we need to clone the repo at the url `ssh://bandit27-git@localhost/home/bandit27-git/repo` and we'll be able to find the flag after that.

Git allows the use of ssh to clone a repo, if not done through http. Since we have the url and the password, it's only a matter of inputing the command. However, right before that, we need to be able to write in the directory we're cloning, which is not the case in the home directory. So, simply create a directory under the `/tmp` directory:

```
 mkdir /tmp/<directory-name>
cd /tmp/<directory-name>
```

The clone the repo:

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

The password is the same as the bandit27 user's password.

The password for the bandit28 user is located in the `README` located in the repo/ directory.

Level 28-29:

Once again, clone the repo into a newly created directory under the `/tmp` directory.

When printing the contents of the `README` file in the newly cloned `repo/` directory, this is what it yields:

```
 bandit28@bandit:/tmp/strix28/repo$ cat README.md
# Bandit Notes
Some notes for level29 of bandit.

## credentials

- username: bandit29
- password: xxxxxxxxxx
```

This doesn't tell us much. Let's try to poke through this repo and see for version control and logs.

```
 bandit28@bandit:/tmp/strix28/repo$ git log
WARNING: terminal is not fully functional
commit edd935d60906b33f0619605abd1689808ccdd5ee
Author: Morla Porla <morla@overthewire.org>
Date:   Thu May 7 20:14:49 2020 +0200

    fix info leak

commit c086d11a00c0648d095d04c089786efef5e01264
Author: Morla Porla <morla@overthewire.org>
Date:   Thu May 7 20:14:49 2020 +0200

    add missing data

commit de2ebe2d5fd1598cd547f4d56247e053be3fdc38
Author: Ben Dover <noone@overthewire.org>
Date:   Thu May 7 20:14:49 2020 +0200

    initial commit of README.md
```

Here we see that there was 3 commits. Let's checkout into the c086d11 commit to see what changed:

```
 bandit28@bandit:/tmp/strix28/repo$ git checkout c086d11a00c0648d095d04c089786efef5e01264
Note: checking out 'c086d11a00c0648d095d04c089786efef5e01264'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b <new-branch-name>

HEAD is now at c086d11... add missing data
bandit28@bandit:/tmp/strix28/repo$ ls -la
total 16
drwxr-sr-x 3 bandit28 root 4096 Jan 25 08:19 .
drwxr-sr-x 3 bandit28 root 4096 Jan 25 08:18 ..
drwxr-sr-x 8 bandit28 root 4096 Jan 25 08:19 .git
-rw-r--r-- 1 bandit28 root  133 Jan 25 08:19 README.md
```

Still only one file, what's inside of it?

```
 bandit28@bandit:/tmp/strix28/repo$ cat README.md
# Bandit Notes
Some notes for level29 of bandit.

## credentials

- username: bandit29
- password: bbc96594b4e001778eee9975372716b2

```

Level 29-30:

Another git level, that's fine, it helps us master our git usage. For this one, do the same thing and clone the repo into a directory under `tmp`.

After analyzing the master branch, it would seem like the `README.md` file contained the password for the bandit30 user, but was redacted says that its only available for development.

Typically with Git repos, there will be a master branch and a development branch, to not mix both environments. Features that are complete and tested, ready to be shipped will be pushed on the master branch, while some that are still under development will stay under the dev branch. Under these two branches, there are also different API keys sometimes for different databases for example. We would not want to be developing a feature and use the production database for instance.

So let's see all the branches that we are dealing with:

```
bandit29@bandit:/tmp/strix29/repo$ git branch -a
* dev
  master
  sploits-dev
  remotes/origin/HEAD -> origin/master
  remotes/origin/dev
  remotes/origin/master
  remotes/origin/sploits-dev
```

There we go, we have a dev branch. We can switch to it using `git checkout dev`, and the password for the bandit30 user is in the `README.md` file.

Level 30-31:

Yet another git challenge. For this one, we're going into it knowing that we've already done a branch-based challenge and a commit based challenge, so it will probably not be the same.

As usual create a repo under tmp and clone the given repo. We can then take a look inside the repo and we'll only see one `README.md` file:

```
bandit30@bandit:/tmp/strix30/repo$ cat README.md
just an epmty file... muahaha
```

Let's see if there are any logs we can analyze.

```
bandit30@bandit:/tmp/strix30/repo$ git log
commit 3aefa229469b7ba1cc08203e5d8fa299354c496b
Author: Ben Dover <noone@overthewire.org>
Date:   Thu May 7 20:14:54 2020 +0200

    initial commit of README.md
```

So that's the only commit that was done on this branch. Let's see the other branches.

```
bandit30@bandit:/tmp/strix30/repo$ git branch -a
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/master
```

Only one branch. Let's check out the tags.

```
bandit30@bandit:/tmp/strix30/repo$ git tag
secret
```

Running a `git show secret` will reveal the flag.

Level 30-31:

YAGC.

Clone the repo into a newly created directory under `/tmp/`. Then analyzing the contents, we have the following:

```
bandit31@bandit:/tmp/strix31/repo$ ls -la
total 20
drwxr-sr-x 3 bandit31 root 4096 Jan 25 18:02 .
drwxr-sr-x 3 bandit31 root 4096 Jan 25 18:02 ..
drwxr-sr-x 8 bandit31 root 4096 Jan 25 18:02 .git
-rw-r--r-- 1 bandit31 root    6 Jan 25 18:02 .gitignore
-rw-r--r-- 1 bandit31 root  147 Jan 25 18:02 README.md
```

There seems to be a `.gitignore` this time. the .gitignore file in a git repo will take a list of file that git should ignore during file staging and commits.

Let's see the contents of `README.md`:

```
bandit31@bandit:/tmp/strix31/repo$ cat README.md
This time your task is to push a file to the remote repository.

Details:
    File name: key.txt
    Content: 'May I come in?'
    Branch: master
```

Ok, let's do that.

```
echo "May I come in?" > key.txt
```

Before pushing, let's see what the contents of the .gitignore file is:

```
bandit31@bandit:/tmp/strix31/repo$ cat .gitignore
*.txt
```

Nice catch. It's ignore all files with the `.txt` suffix, conveniently ignoring the `key.txt` we just had to create.

```
rm .gitignore
git add .
git commit -m "key file"
git push origin master
```

The flag will then be outputted through the use of a git server-side hook.

Level 32-33:

Sidenote: As of writing this, there has been two iterations. Levels 0-25 were done at the date mentionned in the title, but 26-33 were done in May 2020. This is because the challenges got introduced then. Level 33 is the last challenge, but there could always be more later on.

This time, we're back to escape. When logging in, we're greeted with the uppercase shell. This is an interactive shell that transforms all inputs to their uppercase format, making it very difficult to execute commands.

The actual binary for the shell resides in the home directory of bandit33, therefore we can't really analyze it. Furthermore, it's a binary so it would be hard to analyze without the appropriate tools, which we do not have.

Thankfully some quick thinking will allow us to figure it.

Trying to list everything will not work:

```
>> ls -la
sh: 1: LS: not found
```

All letters are turned to uppercase, however, symbols are not:

```
>> ..
sh: 1: ..: Permission denied
```

We can then find an exploit like this since symbols and numbers don't have an uppercase form. Here is a pretty quick combination that gives us a shell:

```
>> $0
$ whoami
bandit33
```

Evidently, `$0` will give us a shell, since $ cannot be capitalized, nor can 0. Put both of them together and they expand to the name of the default shell for the user, in this case `sh`.

We can then execute find the password for the user in the `/etc/bandit_pass/bandit33` file.

Level 33-34:

```
 bandit33@bandit:~$ cat README.txt
Congratulations on solving the last level of this game!

At this moment, there are no more levels to play in this game. However, we are constantly working
on new levels and will most likely expand this game with more levels soon.
Keep an eye out for an announcement on our usual communication channels!
In the meantime, you could play some of our other wargames.

If you have an idea for an awesome new level, please let us know!
```

:)