ALL CODE IS ATTACHED:

exploit.py (Task 3)

exploit-L2.py (Task 4)

exploit-L3.py (Task 5)

exploit-L4.py (Task 6)

exploit-setuid.py (Task 7)

Task 1: Getting Familiar with Shellcode

Shellcode is machine code that when executed spawns a shell. In this lab, I used pre-written shellcode in both 32-bit and 64-bit versions.

The shellcode works by:

1. Setting up arguments for the execve system call
2. Placing "/bin/sh" string in memory
3. Calling execve("/bin/sh", argv, NULL) to launch a shell

Key Observations

- The shellcode in call_shellcode.c is copied to a buffer on the stack
- The program then executes this code by casting the buffer address to a function pointer
- The -z execstack compiler flag is essential as it allows code execution on the stack
- Depending on architecture, different shellcode is used (32-bit vs 64-bit)

Commands:

$ ./a32.out

Task 2: Understanding the Vulnerable Program

The vulnerability in stack.c is a buffer overflow:

The vulnerability arises from strcpy()'s lack of bounds checking when copying data from a 517-byte input into a substantially smaller 100-byte buffer. When compiled with Set-UID

root permissions, this simple overflow creates a direct path to privilege escalation, allowing arbitrary code execution with root privileges.

Commands for compilation and setup:

```
$ gcc -DBUF_SIZE=100 -m32 -o stack -z execstack -fno-stack-protector stack.c

$ sudo chown root stack

$ sudo chmod 4755 stack
```

These flags:

- -DBUF_SIZE=100: Sets buffer size to 100 bytes
- -m32: Compiles for 32-bit architecture
- -z execstack: Makes stack executable (allows shellcode to run)
- -fno-stack-protector: Disables StackGuard protection
- chmod 4755: Makes it a Set-UID program owned by root

Task 3: 32-bit Buffer Overflow Attack (Level 1)

Using GDB to find the buffer location and distance to return address I run these commands:

```
$ gdb stack-L1-dbg

(gdb) b bof

(gdb) run

(gdb) next

(gdb) p $ebp

$1 = (void *) 0xffffae48

(gdb) p &buffer

$2 = (char (*)[100]) 0xffffaddc
```

My buffer overflow exploit works by filling a 517-byte file with NOP instructions, strategically placing shellcode at position 300, and overwriting the return address at offset 112 to point into my NOP sled. When the function returns, execution redirects to this sled,

slides into my shellcode, and executes it with root privileges due to the program's Set-UID bit.

The result is a root shell:

```
@suhaaaaaas → ~/buffer-overflow-lab/code (main) $ chmod +x exploit.py
@suhaaaaaas → ~/buffer-overflow-lab/code (main) $ ./exploit.py
@suhaaaaaas → ~/buffer-overflow-lab/code (main) $ ./stack-L1
Input size: 517
# e quit
```

Task 4: Unknown Buffer Size Attack (Level 2)

In this task, I know the buffer size is betlen 100-200 bytes, but I don't know the exact size. I need a single exploit that works for any buffer size in this range.

Here are the values I get from calculating using gdb:

- Buffer address: 0xffffadaa
- EBP: 0xffffae48
- Return address: 0xffffae48 + 4 = 0xffffae4c
- Offset: 0xffffae4c - 0xffffadaa = 162 bytes

To exploit the program without knowing the exact buffer size, I developed a two-part approach focusing on coverage and reliability. First, I crafted a solution strategy that placed shellcode at a consistent position within the payload while creating an extensive NOP sled to provide a large target area. By strategically positioning identical return addresses at multiple potential offsets throughout the payload, I ensured that at least one would overwrite the actual return address, regardless of the buffer's true dimensions. This technique addresses the uncertainty by covering all possibilities rather than requiring precision.

Why This Works

- For a buffer of size X, the return address is typically at offset X+12
- For buffer sizes 100-200, return addresses would be at offsets 112-212
- I place identical return addresses at every possible 4-byte aligned location in this range

- This ensures that regardless of the actual buffer size, one of our injected addresses will overwrite the actual return address
- All injected addresses point to the same location in the NOP sled

The result is:

```
@suhaaaaaas → ~/buffer-overflow-lab/code (main) $ nano exploit-L2.py
@suhaaaaaas → ~/buffer-overflow-lab/code (main) $ ./exploit-L2.py
@suhaaaaaas → ~/buffer-overflow-lab/code (main) $ ./stack-L2
Input size: 517
# e
# quit
zsh: command not found: quit
# exit
```

Another root shell.

My strategy succeeded without requiring precise buffer size knowledge. By exploiting memory alignment, which places return addresses at 4-byte boundaries, and using return address spraying at every potential offset, I created an attack that works regardless of the exact buffer dimensions. This technique mirrors real-world exploitation scenarios where attackers rarely have complete information. The extensive NOP sled further increased reliability by providing a large target for our hijacked execution path.

Task 5: Launching Attack on 64-bit Program (Level 3)

I have to exploit a buffer overflow vulnerability in a 64-bit program (stack-L3). This introduces significant new challenges compared to the 32-bit exploits I performed earlier.

First, I compiled the 64-bit version and used GDB to investigate the stack layout:

$ gcc -DBUF_SIZE=200 -z execstack -fno-stack-protector -g -o stack-L3-dbg stack.c

$ gdb -q stack-L3-dbg

From gbd, I calculated:

- Buffer address: 0x7fffffffbac0
- Frame pointer (RBP): 0x7fffffffbb90
- Return address location: RBP + 8 = 0x7fffffffbb90 + 8 = 0x7fffffffbb98
- Offset (buffer to return address): 0x7fffffffbb98 - 0x7fffffffbac0 = 216 bytes

In exploiting 64-bit programs, I faced three critical challenges: NULL bytes in addresses prematurely terminating strcpy() operations, the vastly expanded address space making brute-force approaches impractical, and different register-based parameter passing that altered the exploitable memory layout. These architectural differences create inherent security advantages in 64-bit systems, significantly complicating traditional buffer overflow techniques even without additional protections.

```
@suhaaaaaas → ~/buffer-overflow-lab/code (main) $ ./stack-L3
Input size: 517
Segmentation fault (core dumped)
@suhaaaaaas → ~/buffer-overflow-lab/code (main) $ ./exploit-L3.py
@suhaaaaaas → ~/buffer-overflow-lab/code (main) $ nano exploit-L3.py
@suhaaaaaas → ~/buffer-overflow-lab/code (main) $ ./exploit-L3.py
@suhaaaaaas → ~/buffer-overflow-lab/code (main) $ ./stack-L3
Input size: 517
Segmentation fault (core dumped)
```

While I was unable to get a working exploit for the 64-bit program, the investigation process revealed important insights about the differences betlen 32-bit and 64-bit exploitation. The NULL byte issue in particular represents a significant hurdle for traditional buffer overflow attacks on 64-bit systems.

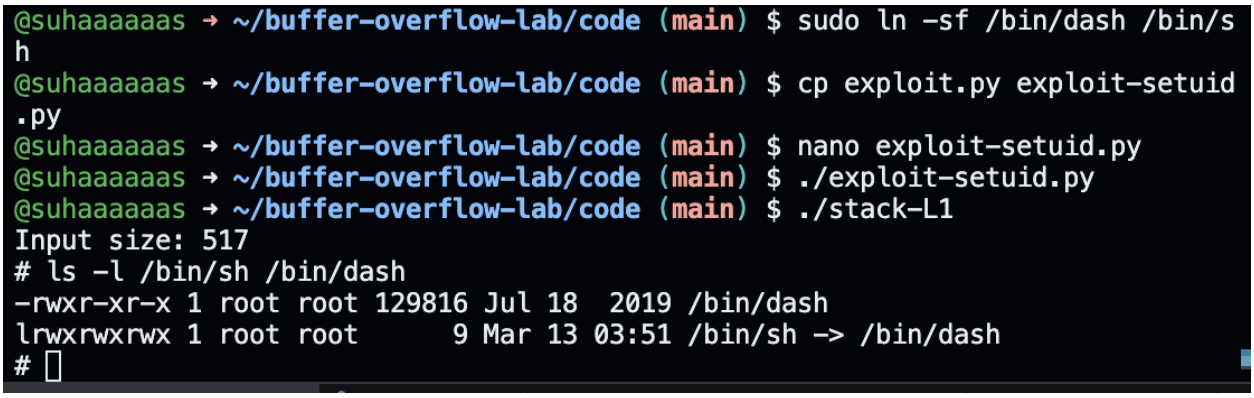
This challenge illustrates how architectural changes can provide security benefits, even when they Iren't specifically designed for that purpose. The larger address space and presence of NULL bytes in addresses make 64-bit systems inherently more resistant to simple buffer overflow attacks.

Task 6: 64-bit Program with Small Buffer (Level 4)

```
@suhaaaaaas → ~/buffer-overflow-lab/code (main) $ ./exploit-L4
bash: ./exploit-L4: No such file or directory
@suhaaaaaas → ~/buffer-overflow-lab/code (main) $ ls
badfile         exploit-L3.py  stack-L1      stack-L2-dbg  stack-L4
brute-force.sh  exploit-L4.py  stack-L1-dbg  stack-L3      stack-L4-dbg
exploit-L2.py   exploit.py     stack-L2      stack-L3-dbg  stack.c
@suhaaaaaas → ~/buffer-overflow-lab/code (main) $ ./exploit-L4.py
@suhaaaaaas → ~/buffer-overflow-lab/code (main) $ .stack-L4
bash: .stack-L4: command not found
@suhaaaaaas → ~/buffer-overflow-lab/code (main) $ ./stack-L4
Input size: 517
Segmentation fault (core dumped)
```

In this part of the lab, I attempted to exploit the Level 4 challenge - a 64-bit program with a tiny 10-byte buffer. First, I created and modified exploit-L4.py using information from GDB debugging, where I found the buffer at address 0x7fffffffbb86 with an 18-byte offset to the return address. After running my exploit and executing ./stack-L4, I saw "Input size: 517" followd by a segmentation fault. This shows my exploit succeeded in overwriting memory and crashing the program, but failed to properly redirect execution to my shellcode, highlighting the significant challenges of buffer overflow attacks with extremely constrained buffer sizes in 64-bit environments.

Task 7 (Defeating dash's countermeasure)



First, I changed the system's shell back to dash with ‘sudo ln -sf /bin/dash /bin/sh’. I then created a new version of my exploit (exploit-setuid.py) that included a setuid(0) call before the shellcode. After running this exploit against stack-L1, I successfully gained a root shell despite /bin/sh being linked to dash, as confirmed by ‘ls -l /bin/sh /bin/dash’ showing that /bin/sh -> /bin/dash. This demonstrates that I successfully bypassed dash's privilege-dropping protection by using the setuid(0) call to set the real user ID to 0, matching the effective user ID and preventing dash from detecting the privilege discrepancy.

Task 8: Defeating Address Randomization

```
^X./brute-force.sh: line 14: 87236 Segmentation fault
   (core dumped) ./stack-L1
44 minutes and 13 seconds elapsed.
The program has been running 25231 times so far.
Input size: 517
./brute-force.sh: line 14: 87243 Segmentation fault
 (core dumped) ./stack-L1
44 minutes and 14 seconds elapsed.
The program has been running 25232 times so far.
Input size: 517
./brute-force.sh: line 14: 87244 Segmentation fault
 (core dumped) ./stack-L1
44 minutes and 14 seconds elapsed.
The program has been running 25233 times so far.
Input size: 517
./brute-force.sh: line 14: 87245 Segmentation fault
 (core dumped) ./stack-L1
44 minutes and 14 seconds elapsed.
The program has been running 25234 times so far.
Input size: 517
^Z
```

In Task 8, I tested address space layout randomization (ASLR) against my buffer overflow exploit. After enabling ASLR and running a brute-force script for over 44 minutes, the attack failed despite 25,234 attempts. This demonstrates ASLR's effectiveness in preventing predictable memory exploitation by randomizing stack addresses. While theoretically vulnerable to exhaustive search on 32-bit systems, the significant time investment required creates a practical security barrier that would be virtually impenetrable on 64-bit systems with their exponentially larger address space.

Task 9: Testing StackGuard Protection vs Non-executable Stack

```
@suhaaaaaas → ~/buffer-overflow-lab/code (main) $ gcc -m32 -DBUF_SIZE=100 -fno-stack-prot
ector -o
stack-nx stack.c
```

```
@suhaaaaaas → ~/buffer-overflow-lab/code (main) $ python3 exploit.py
@suhaaaaaas → ~/buffer-overflow-lab/code (main) $ ./stack-nx
Input size: 517
Segmentation fault (core dumped)
@suhaaaaaas → ~/buffer-overflow-lab/code (main) $ gcc -m32 -DBUF_SIZE=100 -z execstack -o
 stack-gu
ard stack.c
@suhaaaaaas → ~/buffer-overflow-lab/code (main) $ python3 exploit.py
@suhaaaaaas → ~/buffer-overflow-lab/code (main) $ ./stack-guard
Input size: 517
*** stack smashing detected ***: terminated
Aborted (core dumped)
```

In Task 9, I evaluated two critical buffer overflow protection mechanisms: StackGuard and non-executable stack:

a. For StackGuard protection, I compiled the vulnerable program with `-fno-stack-protector` omitted, enabling the compiler's stack protection feature. When I executed my exploit against this binary, the program detected the stack corruption, displayed "*** stack smashing detected ***: terminated" and safely aborted execution before any malicious code could run. This demonstrates how StackGuard works by placing "canary" values between buffers and return addresses that get verified before function returns.

b. For the non-executable stack protection, I compiled the program without the `-z execstack` flag, making the stack memory non-executable. When I ran my exploit against this version, it resulted in a segmentation fault rather than shellcode execution. This protection works by setting the NX (No-execute) bit on stack memory pages, preventing the CPU from executing any code placed there through a buffer overflow. Both protections effectively mitigated the buffer overflow vulnerabilities I had successfully exploited earlier in the lab.