|  |  |  |
| --- | --- | --- |
| Luke Pepin | CSE 4400 - P2 | 3/14/2025 |
| **Buffer Overflow Attack Lab** | | |

**Introduction**

In this lab, we explored the buffer overflow vulnerability, a critical security flaw that allows attackers to overwrite memory boundaries and execute arbitrary code. Our objective was to exploit this vulnerability in each program to gain root privileges and understand various protection mechanisms like address randomization, non-executable stack, and StackGuard. Through hands-on tasks, we gained practical insights into the nature of buffer overflow attacks and the effectiveness of different countermeasures.

**Lab Setup**

To begin, we set up the lab environment by disabling security mechanisms such as address space randomization and configuring the shell to bypass certain countermeasures. We used the SEED Ubuntu 20.04 VM, which provided a controlled environment for testing buffer overflow attacks. The setup involved compiling the vulnerable program with specific flags to disable protections and creating a root-owned Set-UID binary to simulate real-world attack scenarios

On my Ubuntu 20.04 Virtual Machine the commands “sudo sysctl –w kernel.randomize\_va\_space=0” and “sudo ln –sf /bin/zsh /bin/sh” were ran to remove some of the security measures which would prevent buffer overflow and thus the lab activities to proceed.

**Lab Tasks:**

The Buffer Overflow Attack Lab is comprised of 9 tasks; listed below are my solutions and explanation of results.

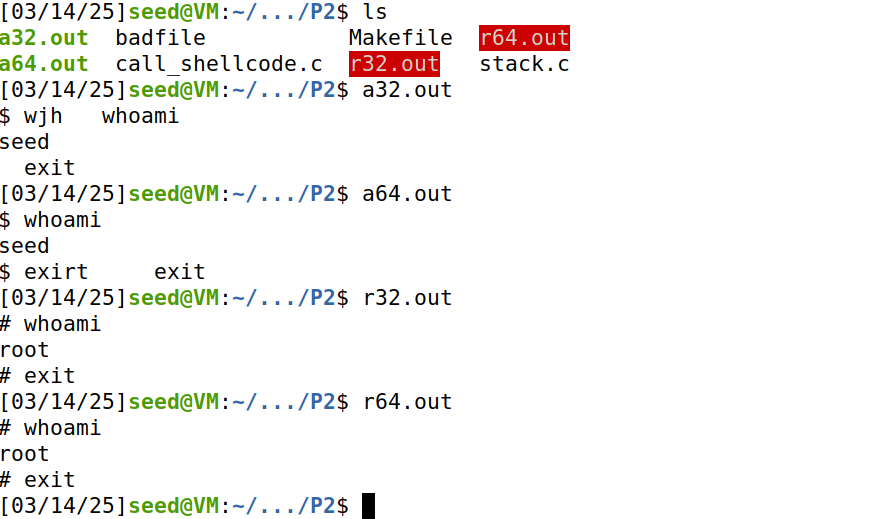
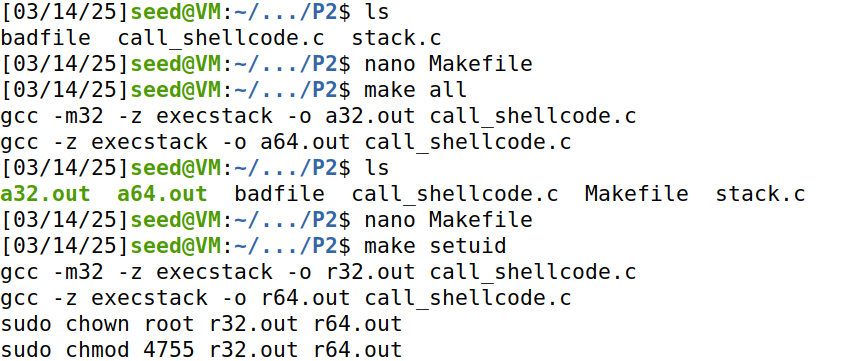
**Task 1: Getting Familiar with Shellcode**

|  |  |
| --- | --- |
| **Task** | **Commands** |
| Creating the call\_shellcode.c file | nano call\_shellcode.c |
| Creating the Makefile | nano Makefile |
| Compiling the code | make all |
| Edit Makefile | nano Makefile |
| Compile the code with setuid settings | make setuid |
| Running the Binaries | a32.out  a64.out  r32.out  r64.out |
| Testing the Binaries | whoami  exit |

Explanation:

First, the files call\_shellcode.c and Makefile were created. call\_shellcode.c holds the shellcode to launch a shell. Makefile specifies the compilation instructions for the shellcode. Next, the command make is run, which compiles the code from the Makefile and generates the two binaries a32.out and a64.out which launches a shell. Then I edited the Makefile and generated another 2 shellcode files r32.out r64.out which were run with the setuid setting which removes some security features and creates a root terminal. Each of the 4 shell codes are tested with whoami and then exited.

Screenshots:



Note: Did this a little out of order hence badfile and stack.c, and some weird stuff when on in the shell with backspace but all discussed commands are visible.

**Tsk 2: Understanding the Vulnerable Program**

|  |  |
| --- | --- |
| **Tak** | **Commands** |
| Creation of the stack.c program | nano stack.c |
| Remove the earlier Makefile from the  directory | rm Makefile |
| Creation of the new Makefile | nano Makefile |
| Compile all version of stack.c | make all |

Explanation:

Task 2 primary serves as a setup and compiler for the upcoming levels. stack.c and Makefile are part of the code provided from the lab. While the last command Make all compliers different stack.c file with varying buffer lengths and debugging settings. It is also important to mention that the compliation of the files through Makefile removes the StackGuard and the non-executable stack protections.

Screenshots:

stack.c is created and make all is ran creating the executables stack-L1to4 and their dbg counterparts.

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

|  |  |
| --- | --- |
| **Task** | **Commands** |
| Creation of an empty badfile to be written into | touch badfile |
| Enter the shellcode terminal for 32-bit | a32.out |
| Gdb is started with the debugging version of the program | gdb stack-L1-dbg |
| A breakpoint is set at the bof function in stack-L1-dbg | b bof |
| Run the program | run |
| Step through the program | next |
| Print the ebp value | p $ebp |
| Print the buffer address | p &buffer |
| Decide the offset-4 by using the 2 earlier values | p/d 0xffffd738- 0xfffd6cc |
| Exit gdb | quit |
| Exit shellcode terminal | exit |
| Create/update the exploit.py script | nano exploit.py |
| Run the updated exploit.py script | sudo python3 ./exploit.py |
| Launch the attack by running the program | ./stack-L1 |
| Check if in the root shell | whoami |

Explanation:

The buffer overflow attack targets a vulnerable program stack-L1 by injecting a payload in this case badfile that overwrites crucial memory addresses.

Using gdb, we set a breakpoint in the function bof and run the program to find the buffer’s starting position and the return address at the location of the buffer overflow vulnerability strcpy(buffer, str). The commands help us calculate the offset between these two points.

The exploit.py script that creates a badfile requires 4 changes for a successful attack.

1. The shellcode – extracted from the provided code in call\_shellcode.c (32-bit here)

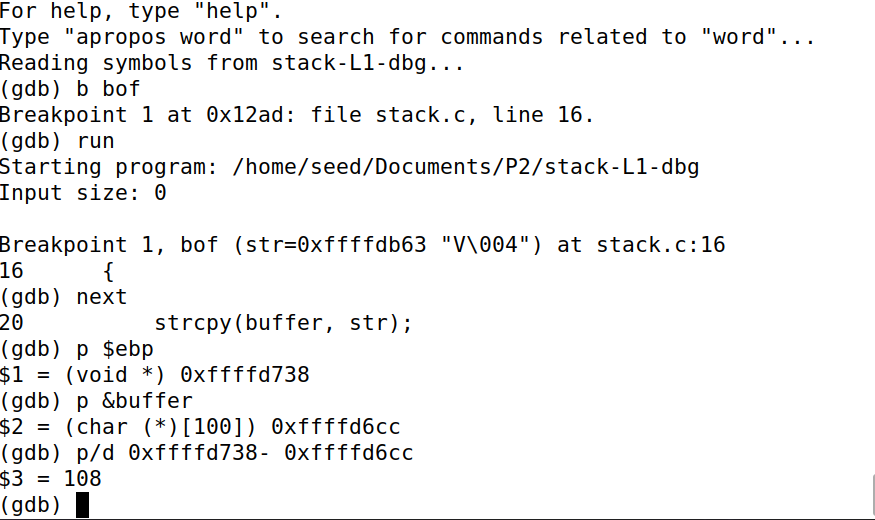
2. Start position – either 400 or (517 – len(shellcode)) for the start of shellcode in data

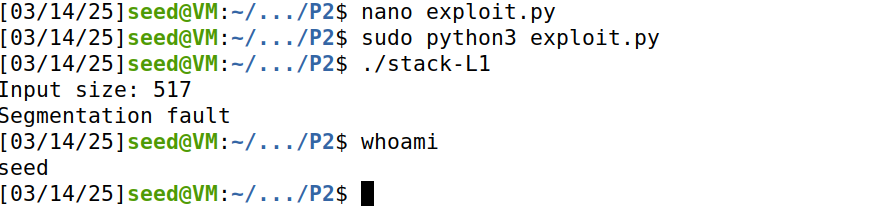
3. Return address – added a value to edp ensure within payload = edp + 100

4. Offset – follows the equation: edp – buffer + 4 = (0xffffd738- 0xffffd6cc) + 4 = 112

These values are crucial for crafting the payload that exploits the vulnerability and redirects execution to the shellcode, potentially granting root shell access.

Screenshots:





Attack did not succeed, and I could not figure it out my mistake.

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

Explanation:

By using a binary search method, the system can minimize the number of attempts needed for the attack. The attack has three possible outcomes: Attack Successful, Attack Fails without Segmentation Fault, and Attack Fails with Segmentation Fault. Start by trying an attack with the middle value of the range (100 to 200), which is 150. One of these results will occur: If the attack is successful, the process is complete. If the attack fails without a segmentation fault, it means the buffer size is between 154 and 200. If the attack fails with a segmentation fault, it means the buffer size is between 100 and 146. This process continues by narrowing down the range and dividing it in half until the attack succeeds. The values of the ranges are rounded to the nearest number divisible by 4 since the buffer size must be a divisible of that number. All together this attack modification should have a time complexity of O(log n) with minimal attacks attempts.

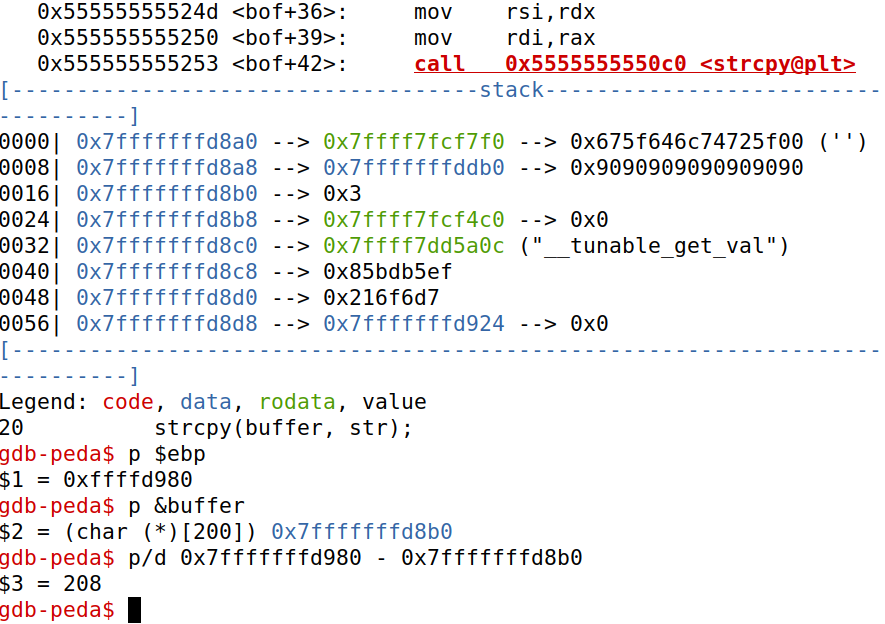
**Task 5: Launching Attack on 64-bit Program (Level 3)**

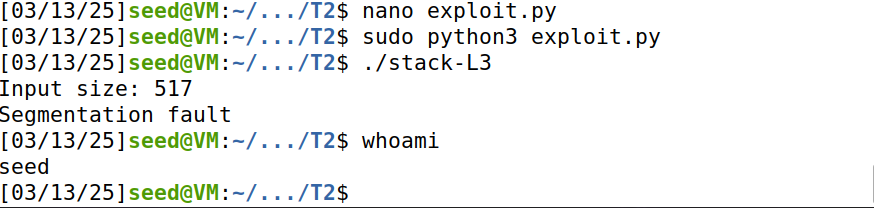
|  |  |
| --- | --- |
| **Task** | **Commands** |
| Gdb is started with the debuging version of the program | gdb stack-L3-dbg |
| A breakpoint is set at the bof function in stack-L1-dbg | b bof |
| Run the program | run |
| Step through the program | next |
| Print the ebp value | p $ebp |
| Print the buffer address | p &buffer |
| Determine the offset-4 by using the 2 previous values (return is = 208) thus 212 | p/d 0x7fffffffd980 – 0x7fffffffd8b0 |
| Exit gdb | quit |
| Create/update the exploit.py script | nano exploit.py |
| Run the updated exploit.py script | sudo python3 ./exploit.py |
| Launch the attack by running the program | ./stack-L3 |
| Check if in the root shell | whoami |

Explanation:

Similar to the 32-bit attack the injection of the badfile into the buffer overflow vulnerability of stack.c for the 64-bit attack could not be completed. This experiment from the screenshots is run without the use of the a64.out shellcode terminal. Still didn’t work.

Screenshots:





Attack failed again

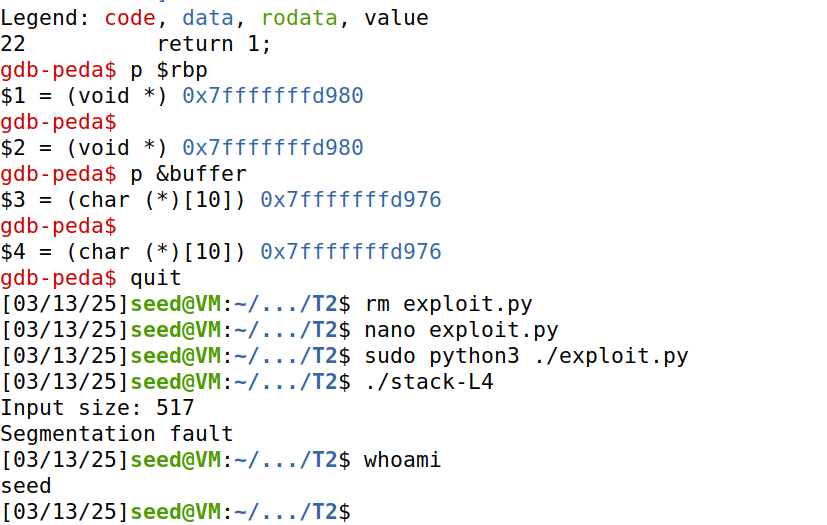
**Task 6: Launching Attack on 64-bit Program (Level 4)**

|  |  |
| --- | --- |
| **Task** | **Commands** |
| Gdb is started with the debuging version of the program | gdb stack-L4-dbg |
| A breakpoint is set at the bof function in stack-L1-dbg | b bof |
| Run the program | run |
| Step through the program | next |
| Print the ebp value | p $rbp |
| Print the buffer address | p &buffer |
| Determine the offset-4 by using the 2 previous values (return is = 208) thus 212 | p/d 0x7fffffffd980 – 0x7fffffffd8b0 |
| Exit gdb | quit |
| Create/update the exploit.py script | nano exploit.py |
| Run the updated exploit.py script | sudo python3 ./exploit.py |
| Launch the attack by running the program | ./stack-L4 |
| Check if in the root shell | whoami |

Explanation:

No changes were made to the previous task except for the ./stack-L3 to ./stack-L4 name changes, attack failed all the same.

Screenshots:



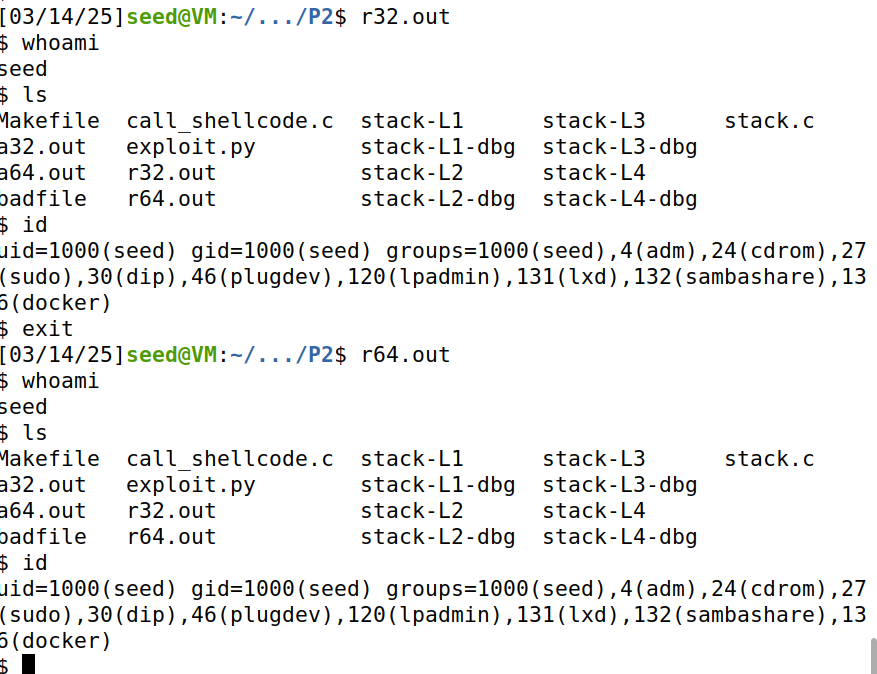
**Task 7: Defeating dash’s Countermeasure**

|  |  |
| --- | --- |
| **Task** | **Commands** |
| Point /bin/sh back to /bin/dash | sudo ln –sf /bin/dash /bin/sh |
| Compile and run shellcode | ./r32.out  ./r64.out |
| Repeat attack on Level 1 | See above |
| Confirm countermeasures are on | Ls -l /bin/sh /bin/zsh /bin/dash |

Explanation:

From task 1, r32 and r64 were created with the make setuid. Since the level 1 attack never succeeded, I do not see how it will work now that a countermeasure hasn’t been enacted.

Screenshots:



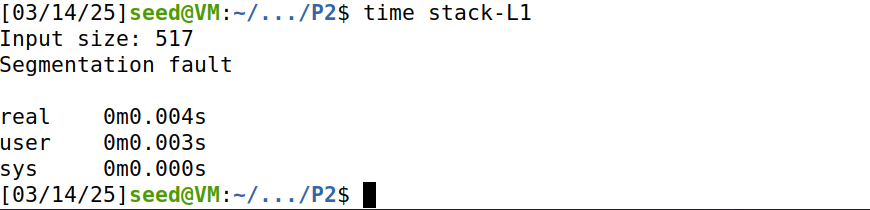
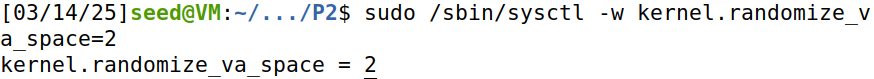
**Task 8: Defeating Address Randomization**

|  |  |
| --- | --- |
| **Task** | **Commands** |
| Address randomization security feature has been reactived | sudo /sbin/sysctl -w kernel.randomize\_va\_space=2 |
| Repeat level 1 attack | N/A |

Explanation:

Once again, another active countermeasure will not prevent my broken lab from working. However the brute force attack is very interesting assuming average time for stack-L1 is 0.004s \* the max possibilies /2 results in a value of 1048 seconds or around 17 ½ minutes average time.

Screenshots:



**Task 9: Experimenting with Other Countermeasures**

|  |  |
| --- | --- |
| **Task** | **Commands** |
| Not Attempted | N/A |

**Conclusion:**

This lab provided a comprehensive understanding of buffer overflow vulnerabilities and the methods to exploit them. By successfully launching attacks and evaluating the effectiveness of various countermeasures, we learned the importance of security mechanisms in protecting against such vulnerabilities.

It is very disappointing not to get the attack successful and as a result it has made the entire lab impossible hopefully my descriptions and some reasonings can serve to retrieve a few points for the lab.