**Lab Report #3**

**Name: Nicolas Rios CIS 495 + 595**

Students need to submit a detailed lab report to describe what they have done, what they have observed, and how they interpret the results. Reports should include evidences to support the observations. Evidences include packet traces, screenshots, etc.

[ENVIRONMENT SETUP 1](#_TOC_250001)

1. TURNING OFF COUNTERMEASURES 1

[LAB TASKS 3](#_TOC_250001)

1. GETTING FAMILIAR WITH SHELLCODE 3
2. UNDERSTANDING THE VULNERABLE PROGRAM 5
3. LAUNCHING ATTACK ON 32-BIT PROGRAM (LEVEL 1) 7
4. LAUNCHING ATTACK WITHOUT KNOWING BUFFER SIZE (LEVEL 2) 10
5. SUMMARY 13

# Environment Setup

1. Turning Off Countermeasures
   * To have the proper setup, to exploit the vulnerability of the buffer with this attack, we must first disable the randomization of address spaces within the heap and stack. This will make guessing the exact address spaces more feasible for the buffer overflow attack. Additionally, we must change the symbolic link between the /bin/sh and the /bin/dash programs since these have countermeasures implemented within that prevent Set-UID programs from executing them. Thus, we change the link to a more vulnerable program /bin/zsh to increase the effectiveness of our attack.

|  |  |
| --- | --- |
| **Screenshots as Evidence** | |
| Disabling Address Space Randomization |  |
| Configuring /bin/sh |  |

# Lab Tasks

1. Getting Familiar with Shellcode
   * For this task, I examined the call\_shellcode.c file, which contains generated binary code from the assembly code used to invoke the execve() function to execute the /bin/sh program. For this lab, we have been instructed to center our focus on the code for the 32-bit machine. I then utilized the Makefile to compile this program using the execstack option which allows for code to be executed from the stack. This produced the a32.out file, which I then ran using the sudo command, resulting in the root shell of the program.

|  |  |
| --- | --- |
| **Screenshots as Evidence** | |
| Call\_  shellcode.c |  |
| Compiling and Invoking Shellcode |  |

1. Understanding the Vulnerable Program
   * In this task, I inspected the vulnerable program, *stack.c*, and noticed that the program takes in user input through *badfile*, which can have a size of 517 bytes. This number is much larger than the size of the buffer within the *bof* function (size of 100 bytes). This will cause a buffer overflow to occur when a large of data is entered by the user into this function, since it is allowed by the strcpy() function that does not check string boundaries. I then complied the stack.c program and disabled the Stack Guard and nonexecutable stack protections countermeasures. I also verified the values of L1thorugh L4, within the Makefile as outlined by the instructor. Finally, I made the compiled stack program a root Set-UID program so that a normal user may invoke this program and gain root shell access for this attack.

|  |  |
| --- | --- |
| **Screenshots as Evidence** | |
| Stack.c file |  |
| Compiling *stack.c* and making *stack* a Set-UID program |  |
| Confirming values for *Makefile* (L1- L4) |  |

1. Launching Attack on 32-bit Program (Level 1)
   * For this task, I first ran the make command to compile the stack program with the debugging flag to add the debugging information to the binary. I then created the badfile using the touch command, since this will store my shellcode within its payload. I then open the debugger for the stack-L1-dbg program in order to know the address of the $ebp (points to stack frame of bof function) and &buffer (start of the buffer) values. With these we can then calculate the distance and determine our return addresses for exploit file.
   * In the exploit.py file, I chose to change the value of start to 517 – the length of the shell code in order to place the shellcode at the end of the payload and fill the rest of the file with NOPs, this will give me a greater range of addresses to choose from so that the slope of NOPs will eventually reach my malicious shellcode. I then chose the return address to be the value of $ebp plus 112 since this shall hopefully compensate for any environment data add by the debugger which manipulates the address value of $ebp. This can then point to some point with in my file so that the malicious code can be run. I then chose the offset of 108+4 since this is the distance between the $ebp and &buffer and again adding 4 to compensate for the added bytes by the debugger.

|  |  |
| --- | --- |
| **Screenshots as Evidence** | |
| Compiling debug/stack files and entering debugger for *stack-L1-dbg* |  |
| *Run* and *next* commands |  |
| Getting *$ebp* and *&buffer* values |  |
| Modifying exploit.py file with new values |  |
| Running exploit.py and stack-L1 |  |

1. Launching Attack without Knowing Buffer Size (Level 2)
   * In this task, I attempted to perform the buffer overflow attack again, however, this time without knowing the size of the buffer ($ebp value). I still utilized the gdb debugger tool to determine the start of the buffer (&buffer) and implemented this into the new values for my exploit-2-class.py file.
   * For the exploit-2-class.py file, I kept the shellcode and array of NOPs the same for the badfile and placed the malicious shell code at the end of the payload, to have room for the return address values within the file. I then changed the value of the return address to the start of the buffer (&buffer) plus 400 bytes in order to make sure that the size of the buffer is surpassed (ranges from 100 to 200) and also moves ahead to either one of the NOPs in the badfile or the start of the malicious shellcode. I then used a for loop to essentially spray the return address within the first 240 bytes (240/4 = 60) of the badfile to increase the likely hood of the return address being pointed to and then proceeding to the malicious shellcode. I then ran this exploit file and then the stack-L2 program and was returned a root shell.

|  |  |
| --- | --- |
| **Screenshots as Evidence** | |
| Running debugger on *stack-L2-dbg* program |  |
| Getting &buffer value |  |
| Modifying exploit file with new return address value and range for offset |  |
| Running exploit-2-CLASS.py and stack-L2 |  |

1. Summary
   * Personally, I found this lab to be quite challenging as it was difficult for me to understand the nature of how the stack and pointers worked. However, once I understand how the stack frame changed with each function, it was easier for me to determine the necessary values for the exploit files. This lab taught me the importance of first visualizing how an attack will be implemented that way if the attack is unsuccessful, it is easier to understand your mistake and change your code to be more effective. Overall, it was interesting to see how a buffer overflow attack could escalate the privileges/access of a normal user by exploiting miscalculations in user input and buffer size made by the developers of a software/program.