**Lab Report #4**

**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. DEFEATING DASH’S COUNTERMEASURE 3
2. DEFEATING ADRESS RANDOMIZATION 7
3. EXPERIMENTING WITH OTHER COUNTERMEASURES 10
4. SUMMARY 12

# 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. Defeating *dash*’s Countermeasure
   * For this task, I attempted to defeat the /bin/dash countermeasure that drops privileges when it detects that the EUID of the program does not equal its RUID (such as when we are using a Set-UID program). To combat this, I utilized the code within the call\_shellcode.c file which had the shellcode to change the RUID of the program to zero, which is the UID of root, since we are using a root-owned Set-UID.
   * I first compiled the call\_shellcode.c file using the “make setuid” command which generated the a32.out and a64.out files. I then ran both of these files and found that they both generated a root shell.
   * I then attempted to run the buffer overflow attack from Task 3 of the previous lab using this updated shellcode to see if it would still be successful with this /bin/dash shell in place.
   * I used the debugger to find the $ebp and &buffer values of the stack file so that I can update the exploit.py file. I also updated this python file with the shellcode for “setuid(0)” and placed that at beginning before invoking the shell command. I then ran the exploit.py file followed by the vulnerable stack program and was returned a root shell.
   * Lastly, I ran the command provided within the instructions to verify that the /bin/sh was pointing to the /bin/dash shell.

|  |  |
| --- | --- |
| **Screenshots as Evidence** | |
| Switching to /bin/dash |  |
| Compiling call\_  shellcode.c  and testing  a32.out |  |
| Obtaining ebp and buffer values |  |
| Updating exploit.py with new shell code, ebp, and buffer vaues |  |
| Running exploit.py | Graphical user interface, text, application  Description automatically generated |

1. Defeating Address Randomization
   * In this task, I attempted to defeat the Address Randomization countermeasure that randomizes the starting address of the stack every time the code is loaded into memory. This makes it difficult to guess the $ebp values as well as the address of the malicious code. To test this, I first enabled the full Address Randomization using the command shown in the first screenshot. I then compiled the stack.c file and converted it to a root-owned Set-UID program.
   * To combat the Address Randomization feature, I utilized a brute-force approach to guessing the starting location of the stack and $ebp values. I ran the brute-force.sh shell script provided within the LabSetup files. This shell script simply ran the stack-L1 program an infinite number of times until the address of the stack and $ebp values matched the ones within the exploit.py file. Since the address location changes each time the stack-L1 program is loaded into memory (executed), it was just a matter of waiting until the address chosen was the same as the one used within the exploit.py file. After running the brute-force shell script for one minute and 22 seconds I was returned a root shell.

|  |  |
| --- | --- |
| **Screenshots as Evidence** | |
| Enabling Address Randomization and Attempting Attack from previous Task |  |
| Compiling stack.c and Creating Set-UID |  |
| Running the brute-force.sh shell script | Graphical user interface  Description automatically generated |

1. Defeating Other Countermeasures
   * For task 9.A, I attempted to defeat the StackGuard countermeasure that is specifically meant for the detection and prevention of buffer overflow attacks. I first tested the success of my attack by turning off the address randomization and running the exploit.py file and the vulnerable Level-1 stack program. After verifying the functionality of my buffer overflow attack, I then compiled the stack.c again, however this time without the -fno-stack-protector flag which disables StackGuard during compilation. I then attempted to run my attack on this new stack program and was returned with the error message “stack smashing detected”. This means that the compiler detected a buffer overflow attack and thus terminated the execution of the stack program.
   * For task 9.B, I attempted to defeat the Non-Executable Stack Protection which is a feature of the compiler that determines if the stack of the running program will be executable, *gcc* automatically makes the stack non-executable. I then modified the Make file to explicitly turn on the Non-Executable Stack Protection flag during the compilation of the call\_shellcode.c program. I then attempted to execute the a32.out and a64.out program files that where generated from the new Makefile, and was returned a Segmentation Fault. This means that the shellcode to generate a new root shell was not executed since the stack of the running program was not executed.

|  |  |
| --- | --- |
| **Screenshots as Evidence** | |
| 9.A: Turning off Address Randomization and Verifying Success of Attack | Graphical user interface, text  Description automatically generated |
| 9.A: Compiling stack.c with StackGuard enabled and testing attack | Graphical user interface, text  Description automatically generated |
| 9.B: Modifying Makefile to enable the Non-Executable Stack Protection and testing the shellcode |  |

1. Summary
   * The tasks within this Lab served as a good hands-on view of the countermeasures in place to defeat the buffer overflow attack and the ways to overcome them. It was interesting how easily the change dash countermeasure was beaten by simply invoking the “setuid(0)” shellcode. Additionally, I believe that it is important to examine the faults within these countermeasures so that we can implement better systems and not blindly rely on protections given by a company or organization. The final task within the Lab, serves as a good gateway into further attacks as we can see how each countermeasures gets progressively more difficult to defeat and we must rely on other methods (or a combination of techniques) to make an attack successful.