**Lab Report #5**

**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.

[LAB TASKS](#_TOC_250001) 1

1. FINDING OUT THE ADRESSES OF *LIBC* FUNCTIONS 1
2. PUTTING THE SHELL STRING IN THE MEMORY 3
3. LAUNCHING THE ATTACK 6
4. DEFEAT SHELL’S COUNTERMEASURE 10

# Lab Tasks

1. Finding Out the Addresses of *libc* Functions
   * For this task, I first modified the value of N within the Makefile to 48 instead of the default 12. This is to ensure that the retlib.c vulnerable program is compilied with the right flags to provide the correct answer for the other tasks within this lab.
   * For the next step, I utilized the debugger (gdb) on the Set-UID program retlib.c in order to find the address of the system and exit functions within the library. These addresses are necessary for our attack since we must jump to these addresses within our arbitrary code since the Non-executable stack feature will not allow for these kinds of commands to be executed directly from our code. Thus, we will jump to the address of the system function within the trusted library that is loaded with the Linux program. The address of the system funciton and exit function are shown in the second screenshot.

|  |  |
| --- | --- |
| **Screenshots as Evidence** | |
| Altering value of DBUF\_SIZE to 48 in the Makefile |  |
| Running gdb on the retlib program and getting address for system and exit |  |

1. Putting the Shell String in the Memory
   * In this task, I first created the MYSHELL environment variable and initialized it with the “/bin/sh” string. This was to place the shell string within memory so that we can determine its address and use this within our system call funciton to return a shell for our attack in the later tasks.
   * I then created the prtenv.c file which essentially printed the address of the “bin/sh” string stored within the MYSHELL environment variable that was inherited by the child process that will be created when the program is executed.
   * Next, I compiled this file into a binary named “prtenv” which is the same number of characters as the name of the vulnerable program “retlib” so that the memory address matches when used in our attack. I also used the -m32 flag during compilation to ensure that it was the same as the retlib program. I then executed the prtenv binary and obtained the address of “/bin/sh”.

|  |  |
| --- | --- |
| **Screenshots as Evidence** | |
| Creating MYSHELL environment variable |  |
| Creating prtenv.c file to print address of “/bin/sh” |  |
| Compiling and executing prtenv and obtaining the address of “bin/sh” |  |

1. Launching the Attack
   * For this task, I first began by modifying the exploit.py file given within the lab setup files by adding the address values for “/bin/sh”, system(), and exit(), which were found in the previous tasks. I then utilized the debugger on the retlib program once again to find the address of the ebp and then used this to determine the values of X, Y, and Z. These values are based upon the word size of the variables and their location within the function. The 60 bytes referenced in the assembly code means that the pointer must jump over 60 bytes (the local variables section) to reach the bottom of the function where we can place our libc functions each separated by intervals of 4 bytes.
   * I then tested the attack with this exploit.py file and executed the retlib program and was returned a root shell. To test the first attack iteration, I commented out the exit function that would populate the badfile and reattempted the attack. I found that the attempt was still successful. This may be due to the need of only the system funciton and “/bin/sh” string and not the need of exit() function which terminates the process. This process may still continue without affecting the root shell.
   * The second attack iteration was the renaming of the retlib vulnerable file to newretlib and attempting the attack once again. However, to my surprise the attack failed when the file was renamed. This may be due to the number of bytes associated with the file name being increased from its original one. This would cause the addresses and X,Y, and Z values to be slightly different then the original retlib program to compensate for the added bytes.

|  |  |
| --- | --- |
| **Screenshots as Evidence** | |
| Finding ebp value and modifying exploit.py |  |
| Executing exploit.py and retlib to test the attack | Graphical user interface, text  Description automatically generated |
| Commented out the addition of the exit() function and attempted attack again | Graphical user interface, text  Description automatically generated |
| Renamed retlib program to newretlib and attempted attack again |  |

1. Defeating Shell’s Countermeasure
   * For task, I utilized the same techniques as in the previous tasks to obtain the address for the “/bin/dash” and “-p” shell strings using environment variables to be passed to the execv function. I then used the debugger to get the address of the execv function and modify the exploit file. However, I was not successful in having this attack perform properly with this feature.

|  |  |
| --- | --- |
| **Screenshots as Evidence** | |
| Modifying prtenv and obtaing address for “/bin/dash” and “/-p” |  |
| Using debugger to obtain address for execv function |  |
| Modifying exploitT4.py file and attempting attack |  |