CS4235/6035 Information Security
Project #1 Buffer Overflow

The goals of this project:

- Understanding the concepts of buffer overflow
- Exploiting a stack buffer overflow vulnerability
- Having some thoughts about how to mitigate code reuse attacks (advanced buffer overflow attacks)

Students should be able to clearly explain: 1) what is buffer overflow; 2) why buffer overflow is dangerous; 3) how to exploit a buffer overflow. With the knowledge about buffer overflow, students are expected to launch an attack that exploits a stack buffer overflow vulnerability in a provided toy program. Finally, students are encouraged to have some thoughts about mitigating code reuse attacks. Some research papers are provided for reading.

1. Understanding Buffer Overflow

Note: For this task, you may use online resources to show a program with these vulnerabilities, but please cite these online sources. The diagrams should be your own (**not** copied from the online resources).

1) **Stack buffer overflow** (15 points)
   Write a testing program (**not** sort.c from task 2) that contains a stack buffer overflow vulnerability. Show what the stack layout looks like and explain how to exploit it. You are not required to write the real exploit code, but you may want to use some figures to make your description clear and concise.

2) **Heap buffer overflow** (15 points)
   Write a testing program that contains a heap buffer overflow vulnerability. Show what the heap layout looks like and explain how to exploit it. Again, you do not need to write the real exploit code, but you may want to use some figures to make your description clear and concise.

Deliverable: a pdf file containing your vulnerable programs (paste your code into the pdf directly) and your explanations.

2. Exploiting Buffer Overflow (60 points)

The following C code contains a stack buffer overflow vulnerability. Please write an exploit (e.g., Python script) to open a shell on Linux. The high level idea is to overwrite the return address with the address of function *system()*, and pass the parameter *"sh"* to this function. Once the return instruction is executed, this function will be called to open a shell.

We have provided you with a virtual machine image for this project. We do not recommend you use your own VM image. Our VM's image link is at the following: evan.gtisc.gatech.edu/cs6035/project1/Project1.ova. Our image is also hosted on Torrent. The magnet

link is: magnet:?xt=urn:btih:bef1714c0410ce390eb91070fa6e250a790e6b59&dn=Project1.ova. The original torrent file can be found at evan.gtisc.gatech.edu/cs6035/project1/Project1.ova.torrent.

Steps

0) Import. the OVA file to VirtualBox. Username: ubuntu Password: 123456
1) Compile the provided C code (which you will be exploiting): *gcc sort.c -o sort -fno-stack-protector*.
2) To run this program, put some hexadecimal integers in the file: *data.txt*, and execute *sort* by: *./sort data.txt*
3) When you put a very long list of integers in *data.txt*, you will notice *sort* crashes with memory segfault, this is because the return address has been overwritten by your data.
4) Now you can craft your shellcode in *data.txt*. Again, your goal is to overwrite the return address with the address of function "system()" and pass it with the address of string "sh". Do not use environment variables to store these addresses and then access those environment variables. Use the library addresses of "system()" and "sh" explicitly. GDB can be used to find these library addresses and test/debug your exploit. However, it should be noted that your final exploit (i.e., the final version of your *data.txt*) should work **outside** of GDB. Just running "./sort data.txt" should spawn a shell for you.
5) Provide a screenshot of you exploiting sort.
6) Have fun.

Deliverables: the *data.txt* file you craft and a screenshot of the exploit contained in a PDF file (the same from task1).

3. Open Question (10 Points)

First, if you are not familiar with code reuse attacks, please read the following papers:

1) The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)
2) On the Effectiveness of Address-Space Randomization
3) Code-pointer Integrity
4) Control-Flow Bending: On the Effectiveness of Control-Flow Integrity
5) ASLR-Guard: Stopping Address Space Leakage for Code Reuse Attacks

Two general detections of mitigating code reuse attacks are code diversification and control flow integrity. Interestingly, both directions have their own limitations, and have been shown to be still vulnerable. Write on your thoughts about code reuse attacks or defenses. Note that this is an open question. Any answer will be credited as long as it is reasonable in some sense.

Deliverable: write down your answer in the same pdf file of tasks 1 and 2.

The final deliverables: one pdf file (containing a screenshot of sort being exploited) and one data.txt file