CS 6035 Introduction to Information Security
Project #1 Buffer Overflow

The goals of this project:

- Understanding the concepts of buffer overflow
- Exploiting a stack buffer overflow vulnerability
- Understanding 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 asked to read up on and write about code reuse attacks.

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. In particular, please include in your diagram: (1) The order of parameters (if applicable), return address, saved registers (if applicable), and local variable(s), (2) their sizes in bytes, (3) size of the overflowing buffer to reach return address, and (4) the overflow direction in the stack. 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. In particular, please include in your diagram: (1) each chunk of memory allocated by malloc(), (2) their sizes in bytes, (3) metadata of heap as it gets overwritten, (4) the sizes of this metadata in bytes, and (5) which metadata get overwritten and how the attacker controls which value can get written to any arbitrary location in memory. 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 attached C code (sort.c) contains a stack buffer overflow vulnerability. Please write an exploit (by modifying data.txt) 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, use the **latest** version of VirtualBox. We **do not recommend** using your own VM image. Our VM's image will be located at the following links which will allow you to download the .ova file

1. https://gtvault- my.sharepoint.com/personal/nyang38*gatech*edu/_layouts/15/ guestac-

   cess.aspx?docid=0ec8782993d054c0290d7251bb49

   e20dd&authkey=AbcoeZARVgqfNctJpA2tJOQ

2. https://www.dropbox.com/s/z90hv2qk3comhnq/Project1.ova ?dl=0

3. https://gtvault- my.sharepoint.com/personal/nyang38*gatech*edu/_layouts/15/ guestac-

cess.aspx?docid=0125ca32fb8754018921be11dd17

776c6&authkey=Ad9zDmzaLfJ9Vmuo43jkues

md5 hash: 0d8e71ed88646842f3dde4ab2e4e2b21

Steps:

1) Import the OVA file to VirtualBox. (Username: ubuntu, Password: 123456)
2) Compile the provided C code (which you will be exploiting): *gcc sort.c -o sort -fno-stack-protector*.
3) To run this program, put some hexadecimal integers in the file: *data.txt*, and execute *sort* by: *./sort data.txt*
4) 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.
5) 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 (if you're using GDB for the first time, we recommend checking out GdbInit) 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.
6) Provide a screenshot of you exploiting sort.
7) Have fun.

Deliverables: the *data.txt* file you craft and a screenshot of the exploit. The screenshot should be put into the 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

Task 2 is successful only when two major countermeasures, viz. stack canaries (which was disabled using the flag -fno-stack-protector) and ASLR (which has been disabled system wide on the VM – hint: check out /proc/sys/kernel/randomize_va_space) are disabled. DEP is another major counter measure which can be disabled using the "-z execstack" flag during compilation. However, in the real world both of these counter measures will not be serendipitously turned off. Explain the techniques used to defeat ASLR and DEP respectively.

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

The final deliverables: A pdf file (containing the answers to all of the questions above) and the modified data.txt file which exploits sort.c