CS 6035 Introduction to Information Security
Project #1 Buffer Overflow
Spring 2019

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). You can use Purdue OWL format for citing, as well as http://www.easybib.com/website-citation.

1) **Stack buffer overflow** (25 points)

a. (15 points) Memory Architecture. Describe the stack in the address space of the VM, in generalities. Specifically, address where in memory the stack would be located, what the stack structure looks like when data is pushed onto the stack and popped off the stack. Discuss what register values are placed onto the stack, where user variables are placed within the stack, where arguments would be placed in the stack relative to pertinent register storage within the stack, and finally how program control flow is implemented using the stack. How does the stack structure get affected when a buffer of size 'non-binary' is allocated by a function (ie – buffer size which causes misalignment within the stack)? When a stack grows, in which direction, relative to overall memory, does a stack consume memory?

b. (10 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 (5) What locations within the stack are actually overwritten with your target data to exploit a stack to cause the routine you want to execute to be invoked? *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)

a. (5 points) Memory Architecture. Where is the Heap located in a machine's memory map, in general. Contrast this to Stack memory allocation.

b. (10 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.

Address the data structure implemented in a heap memory. How are allocated and non-allocated chunks structured? Is heap memory contiguous within memory architecture? Again, you do not need to write and test the real exploit code, but you may want to use some figures to make your description clear and concise.

2. Exploiting Buffer Overflow (50 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 *"/bin/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. VirtualBox is a general-purpose full virtualizer for x86 hardware, targeted at server, desktop and embedded use. You can download it here: https://www.virtualbox.org/wiki/Downloads

We **do not recommend** using your own VM image. Our VM's image link for SPRING 2019 is at the following:
https://drive.google.com/file/d/1B391eKMzaooouNRPBEKKvH5K9gbLGQHE/view?usp=sharing .

As a general suggestion for the rest of the course (since we will be using virtual machines in 3 of the 4 projects) you will need to figure out how to share files between your VM and your computer, and how to take a screen shot of your VM.

DO NOT UPDATE THE APPLIANCE IN YOUR VM. DO NOT EDIT SETTINGS, RAM, etc. FAILURE TO FOLLOW THESE INSTRUCTIONS WILL LIKELY RESULT IN A LOWER GRADE.

Steps:

1) Import the OVA file to VirtualBox. (Username: ubuntu, Password: 123456)
2) Immediately CLONE the original OVA for use in the video portion of this project. The cloned OVA is not to be used for anything other than video proof of your exploit.
3) Compile the provided C code (which you will be exploiting): *gcc sort.c -o sort -fno-stack-protector. (DO NOT use other options)*
4) To run this program, put some hexadecimal integers in the file: *data.txt*, and execute *sort* by: *./sort data.txt*
5) 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. If you first answer step 1, part 1 above, you should understand the goal of this exploit and why a seg fault occurs. Pay attention to the 'non-binary' allocated buffer and what it does to the stack structure (and you can see this in GDB).
6) 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 "bin/sh". Do not use environment variables to store these addresses and then access those environment variables. Use the library addresses of "system()" and "bin/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. This needs to be a clean shell, with NO segfaults.

7) You can verify the exploit has occurred because you will get a new, clean command prompt. But, how do we know it is not the same bash shell that invoked the sort program? To verify you have successfully caused a buffer overflow, issuing echo $$ will give the process ID of the current process. Issuing: echo $$ (shows PID of current bash shell), then run ./sort data.txt, (then after sort completes to a clean shell) issue echo $$ to show the PID of the current exploited shell. (noted – echo $0 instead of or in addition to echo $$ will return the name of the current shell)
8) Provide a screenshot of you exploiting sort.
9) Have fun.

Deliverables: the *data.txt* file you craft, a video, and a screenshot of the exploit. The screenshot should be put into the PDF file (the same from task1). Also, in your report with the screenshot, please identify which OS, Version, and bus width you used which contains your VM (for example, PC, Windows 10, 64 bit).

VIDEO INSTRUCTIONS:
Submit a video using the following steps to document your exploit on your machine. These steps must be precisely followed AFTER you upload your data.txt file to Canvas:

Now, showing each step on video, lets demonstrate the exploit from scratch:

1. Open the CLONED OVA in Virtual Box.
2. Compile the sort.c file in your VM.
3. echo $$ which will show the PID of the current shell.
4. Run the sort on the initial supplied data.txt file, (not your, data.txt file) showing the original sample data sorted.
5. Rename the supplied sample data.txt file to original_data.txt
6. In your VM, log into Canvas and download your submitted data.txt file. Do not edit this file.
7. Issue the sort command on the downloaded data.txt file. (note – depending on the number of submissions, your data.txt file may have been renamed by Canvas. That is OK –we simply need to see the downloaded file used in the video.)
8. echo $$ to show the different PID of the new shell.

DO NOT repeat the exploit multiple times to eventually get it to work. First time only will be counted. It must be clear the file you submitted was the data.txt run in your video sort. This video should be succinct and fairly short. DO NOT post this video to any public domain sites to be viewed.

Fine print: there will be no regrade requests accepted on the exploit if it fails the autograder, regardless of whether the exploit works in your machine. This video will serve as proof of your exploit should the autograder fail (which it does occasionally). Failure to supply this video may mean you receive a zero on this portion of the project should your submission unfortunately fail the autograder. We want everyone to receive full credit – especially since this exploit takes significant time.

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

Then read the paper - Jump-oriented programming: a new class of code-reuse attack
and explain the similarity and difference between Jump-Oriented Programing and Return-Oriented Programing.

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

4. The final deliverables:

A pdf file (containing the answers to all of the questions above) named *your_gt_studentid_project1.pdf* and the modified *data.txt* file (named data.txt) which exploits sort.c. Also, upload your video as *your_gt_studentid_video.mov.*

Put your studentID on the top of each page of your deliverable report, to help us when grading. To clarify, if my gt student id is lbrown318, I would submit lbrown318_project1.pdf, lbrown318_video.mov and data.txt.

In Canvas, you will upload your project1.pdf, data.txt, and your video file in assignment: **Project 1 - Understanding Buffer and Heap Overflows.**

Your gradebook will show the report and exploit in separate assignments under project 1.

DO NOT ZIP THE FILES!

Failure to follow the instructions will cause unnecessary point loss.