CS 6035 Introduction to Information Security
Project #1 Buffer Overflows
Fall 2019

Suggestions/Warnings:

- **Report Length –** The report should **AT MOST** be 7 pages. Anywhere between 1-7 pages is acceptable as long as you answer all the questions.
- **Read Piazza** – Lots of questions are answered there daily. Be sure to check there before asking a question.
- **Plagiarism will not be tolerated**! Anything more than 10% of work that is not yours, will be considered plagiarism, and will be given a 0% and not graded. Everything that is not yours **BE SURE TO CITE**.
  - We will be using anti-cheating software, so if you copy you will be caught and reported to OSI.
  - You must include a Works Cited/Bibliography page in MLA format.
    - You can use easybib or citationmachine or anything else to help cite.

The goals of this project:

- Understanding how stack and heap memory are used
- Understanding the concepts of buffer overflow
- Exploiting a stack buffer overflow vulnerability
- Understanding code reuse attacks (advanced buffer overflow attacks)
- With the knowledge about buffer overflow, students are expected to launch an attack that exploits a stack buffer overflow vulnerability in a provided sort.c program.
- Students are asked to read up on and write about code reuse attacks.
- Should be able to explain what a buffer overflow is and why are they dangerous?
- Should be able to explain how an actual buffer overflow works on both the stack and the heap.

A Helpful Suggestion:

It is natural for everyone to want to go complete the exploit first. **However**, if you follow the project flow and answer the first two questions before moving on to the exploit, it will make the process a whole lot easier to understand.

Helpful Hint/Question:

- **What is the EASIEST way to bypass a sorting function?** Figure this out and it will make the task much simpler. Reminder: You are **NOT** allowed to change any code.
- If you are struggling, use the #define to comment out the sort, recompile, and try and get the shell to execute. Once satisfied, uncomment the #define, recompile, and ensure the shell still executes.
- Do not wait until the last minute - this exploit does take significant time, especially if you are new to C and debugging.
- No C or assembly programming is required in this project.

Information/Hints:

Some information you could probably use throughout this project will be listed here. Understand these are only helpful topics and will not give you the answers to solve the problems but could help guide you to finding better information.

- When debugging using GDB, compile with: *gcc -g sort.c -o sort -fno-stack-protector*
- Useful tips for GDB debugging: Useful Commands for GDB or Tricky Useful Commands for GDB
- More useful commands for GDB: Most Tricky Useful Commands for GDB Debugger
- What is a stack or the heap? Information about Stacks / Information about Heaps
- What is a buffer overflow? Buffer Overflow
- Setting up the VM? Try this one or check piazza for the thread about setting up your VM.
- Add a shared folder via the VM: Getting Files off the VM
- A good video about understanding ROP: Understanding ROP -- YouTube
- Do not worry about endianness!
- Addresses in *data.txt* should be 12345A, **NOT** 0x12345A or /x12/x34/x5A. The file is read as hex data.

Understanding Buffer Overflow (40 points)

Note: For this task, you may use online resources to **learn** about a program with these vulnerabilities, but please **cite** these online sources. The diagrams you use can be copied from these online resources, but if they are, **explain the diagram thoroughly in your own words**. Review "Suggestions/Warnings above about how to cite and the percentage allowed to be copied.

1. Stack Buffer Overflow **(25 points)**

a. **(15 points) Memory Architecture.** (Diagram(s) would be helpful, but are not required)

i. Describe the stack in the address space of the VM (in generalities).

ii. Addresses where in memory the stack would be located (specifically).

- Which direction, relative to overall memory, does a stack consume memory when it grows?

iii. Explain how program control flow is implemented using the stack.

iv. 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)? [Also known as 'non-binary']

v. Create a diagram that includes the following

- What does the stack structure looks like when data is pushed onto the stack and popped off the stack?
- Show what register values are placed onto and used with the stack.
- Where would arguments be placed on the stack?
- Where are local user variables placed on the stack?

b. (10 points) Testing Program – Stack Buffer Overflow

i. **(4 Points)** Write a testing program (in C) that contains a stack buffer overflow vulnerability. (You **cannot** use *sort.c* from task 2) . You are not required to exploit it.

- Provide this program in your PDF writeup. (copy/paste is fine. **No Screenshots**)

ii. **(6 Points)** Show what the stack layout looks like and explain how to exploit it. (**Include a diagram**)

- Include the following items:

a. The order of parameters (if applicable), return address, saved registers (if applicable), and local variable(s).

b. The sizes in bytes.

c. The overflow direction in the stack.

d. Size of the overflowing buffer to reach and overwrite the return address.

e. Overflow data that is meaningful for an exploit (this can be general).

2. **Heap Buffer Overflow (15 points)**
    a. **(5 points)** Memory Architecture. (A diagram would be useful here)
        i. Where is the Heap located in a machine's memory map?
        ii. Contrast this to Stack memory allocation (in general terms).
        iii. Describe the data structure implemented in a heap memory.
        iv. How are allocated and unallocated chunks structured? (Show a diagram)
        v. Is heap memory contiguous within the memory architecture? (Yes or No and why?)
    b. **(10 points)** Write a testing program (in C) that contains a heap buffer overflow vulnerability. (Provide an example in the project. Copy/paste is fine. **No Screenshot**). Again, you do not have to exploit it.
        i. Show what the heap layout looks like and explain how to exploit it. **(Include a diagram)**
            - Include the following items:
                a. Each chunk of memory allocated by malloc() and their metadata.
                b. Their sizes in bytes.
                c. The overflow direction in the heap.
                d. The size of the overflowing buffer to reach and overwrite the metadata.
                e. Overflow data that is meaningful for an exploit (this can be general).

3. Exploiting Buffer Overflow (50 points)

- First things to do:
  - ➢ 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 Fall 2019 is at the following:
    - https://drive.google.com/file/d/1UVpZAlpDHGrI9lh-4tasO9NovkhKE0dR/view?usp=sharing
    - The appliance can also be downloaded from the instructions on Canvas.
    - This download will take significant time, depending on your internet speed (maybe 15-60 minutes possibly). So if it does, don't be surprised
  - ➢ **DO NOT UPDATE THE APPLIANCE IN YOUR VM. DO NOT EDIT SETTINGS.**
    - Failure to follow these instructions will likely result in a lower grade.
    - (**3D acceleration and VRAM are the exceptions to this guideline**)
      - You can enable 3D Acceleration and allocate more VRAM.

❖ Once everything is downloaded and installed, you will notice that there is an attached C code file (*sort.c*), the main text file (*data.txt*) holds examples of hex values. Your job is to craft *data.txt* in order to get the buffer overflow to open a shell on Linux and then exit cleanly.

- ➢ In order to do this, you will need to find three addresses.
  - The address of the function *system()*
  - The address of the function parameter for *system()* which is */bin/sh*. */bin/sh* parameter is what will spawn a shell
  - The address of a function that can exit the shell.
- ➢ Once you have these three addresses, you will need to put them in *data.txt* in such a way that they are put into the correct locations in memory. This will cause a shell to spawn and allow you to exit from the shell cleanly.

❖ **General suggestion:** for the rest of the course (We will be using VM's for 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 screenshot of your VM. (**Check the hints portion above for help on this**)

Helpful Hints:

- Inside sort.c you will see "#define SORT_ME 1". If you comment this line out and **RECOMPILE** then you will notice that your *data.txt* file no longer gets sorted. Doing this could be a possible help to execute a shell and ensure you have the right addresses and format. Once you get the shell to spawn, uncomment that line and **RECOMPILE**, your values will now be sorted.
  - ➢ When we grade your *data.txt* file, our code **will** sort your values, so ensure that your *data.txt* works when being sorted.
- Make sure that your data.txt works **outside** of the GDB debugger. Often it will work inside but not outside. This has to do with addresses. If you have this error, make sure you understand the differences.

What is required:

- **Provide a screenshot of you exploiting sort. (**Put the screenshot in your write up)
  - ➢ If you fail to provide a screenshot, you will lose points. **(5 points)**
- **Give an explanation about how you figured out the exploit**. (Put this in your write up)
  - ➢ If you fail to provide an explanation, you will lose points. **(5 points)**
- **Create a video of your exploit working.** (Explanation below in Final Deliverables)
  - ➢ This is vital. There will be no regrades, this video will be your proof if you exploit fails in our auto-grader.
  - ➢ Video must be .mov or .mp4 format

Steps:

- Import the OVA file to VirtualBox. (Username: ubuntu, Password: 123456)
- 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.
- Compile the provided C code: *gcc sort.c -o sort -fno-stack-protector*. *(DO NOT use other options for the final submission)*
- To execute *sort* on the *data.txt* file, run the command: *./sort data.txt*

Information:

- Now you can put values in your *data.txt* file in such a way that a shellcode is executed.
    a. **Your main goal is to overwrite the return address with the address of *system()* and pass that the address of the string */bin/sh* as well as pass an *exit()* address.**
        i. Do not use environment variables to store these addresses. This is **NOT** allowed.
        ii. Use the library address of *system()*, */bin/sh*, and an *_exit()* function explicitly.
- To find the required information for this project, you will be using the GDB debugger. If you have never used the GDB Debugger before, you should view "helpful hints" on the first page.
    a. Again, **make sure** that your final exploit works **outside** of the GDB debugger. Simply running "*./sort data.txt*" should spawn a shell for you.

- **This must execute a clean shell, then exit cleanly. No segfaults are allowed.**
    a. Your screenshot should look like the one below. (Yours will include the values from *data.txt*)

Note: 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 (or *mis-aligned*) buffer and what it does to the stack structure (and you can see this in GDB).

4. Open Question (10 Points)

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

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

Then read the paper - Jump-oriented programming: a new class of code-reuse attack.

- **Explain the similarities and differences between Jump-Oriented Programing and Return-Oriented Programing.**
- **What protection measures do they overcome or are vulnerable to?**
- **Why might you use one over the other?**

Deliverable:

- Write down your answer in the same pdf file for tasks 1 and 2.

5. The final deliverables:

- A PDF answering all the above questions and a screenshot of the exploit.
  - This should be named *your_gt_studentid_project1.pdf*
  - Put your student ID on the top of each page of your deliverable report.
  - Underneath the screenshot of your exploit, please include the following:
    - Identify which OS, Version, and bus width you used which contains your VM
      - Example: PC, Windows 10, 64 bit
- A video showing your exploit working correctly.
  - This video should be named *your_gt_studentid_video.mov.*
  - **Suggested screen recording option:** download OBS from **https://obsproject.com/**
- The modified file *data.txt* which exploits the overflow in sort.c.
  - This should be named *your_gt_studentid_data.txt*
  - **Warning: Windows will convert your end of lines in your *data.txt* file to windows format. We are testing your *data.txt* files in Linux, so be sure it is not converted.**
    - Suggestion: Using notepad++ will stop this from occurring.

Clarification:

If my gt student id is lbrown318, in canvas I would submit: (and where I would submit)

- lbrown318_project1.pdf -- Submit to: **Project 1 - Understanding Buffer and Heap Overflows.**
- lbrown318_video.mov -- Submit to: **Project 1 - Understanding Buffer and Heap Overflows.**
- lbrown318_data.txt -- Submit to: **Project 1: Stack Buffer Exploit**

DO NOT ZIP THE FILES! – This will cause **point loss**. Use this naming format and uploading instructions. (either will cost 10 Points).

- Don't worry that canvas may insert a -# in the middle of your file name - this is done by the system to change a file name so prior submissions remain in the system. The autograder can handle this when grading the data.txt file.

VIDEO INSTRUCTIONS: (There does not need to be audio. No need to explain anything.)

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 Virtualbox.
  2. Compile the sort.c file in your VM.
  3. echo $$ and echo $0 which will show the PID and name of the current shell.
  4. Run the sort on the initial supplied data.txt file, (not your, data.txt file) showing the original sample data sorted.
  5. Rename the supplied sample data.txt file to original_data.txt
  6. In your VM, log into Canvas and download your last submitted data.txt file. **Do not edit this**.
  7. Issue the sort command on the downloadeded data.txt file. (**Note**: depending on the number of submissions, your data.txt file may have been renamed by Canvas. That is OK –we simply need to see the downloaded file used in the video.)
  8. Type "echo $$" and then "echo $0" to show the different PID and name of the new shell.
  9. Type "exit" to show that the shell is cleanly exited.

DO NOT repeat the exploit multiple times to eventually get it to work. Only the first time 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 short. **DO NOT** post this video nor any of your work to any public domain sites to be viewed (including public github repositories).

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 exploit actually work and your submission unfortunately fail the autograder. We want everyone to receive full credit – especially since this exploit takes significant time. And we review the submitted videos of every exploit which fails the autograder.

Please be concise in your responses. **Make sure to upload all 3 files in your final submission (at the appropriate location)**. Upload the .pdf and the .mov files in the same submission. Thank you.