Project I

CS 6238: Secure Computer Systems

Understanding Memory Protection

Learning Objectives:

The goal of this project is to help students become familiar with memory protection facilities provided by operating systems. In particular, you will learn how to limit access to a certain region of memory (e.g., only read, write, or execute access). The project has two parts. First, you will learn about **mprotect()**, a memory protection system call. In the second part, you will explore why it is a good idea to disable execution on the stack to safeguard a program's execution. Memory protection is not limited only to these two examples, there are many different methods to protect the stack and heap. You are encouraged to look at and familiarize yourself with other relevant memory protection mechanisms as well. However, to keep this project simple and of limited scope, we will focus only on the protection of the memory by the above two discussed mechanisms. Thus, the objectives of the project are as follows:

1. Understand how *mprotect()* works and how it can be implemented.
2. Understand the benefits of turning off the executable stack.

Project Setup:

Note: The link to the VM will be posted on an Ed Discussion pinned post.

For this project, you will be provided an Ubuntu-based Virtual Machine (VM). This VM was tested on Oracle VM Virtual box 6.1.34 and can be directly imported to it[1]. This VM has a default account "project1" setup with normal user access privileges. You do not need root account credential for completion of this project.

Password for the "project1" account is "CS6238". You should not include "" while entering the password. When you log into account "project1", you will find Project_1 folder on Desktop. This folder contains 2 additional folders named, Stack_Protection and M_protect. We are going to use the contents of folder M_protect for completion of Task 1, and similarly contents of Folder Stack_Protection for Task 2.

Background:

This project assumes that you know the basics of C programming language (or can learn it quickly). More specifically, you should be able to understand and modify a given C code snippet. Second, you should know how a basic buffer overflow works. For the understanding of basic buffer overflow, you can refer to course materials in CS 6035, Intro to Information Security. If

[1] By now, you should know how to import a VM. If not, please visit: https://docs.oracle.com/cd/E26217_01/E26796/html/qs-import-vm.html.

not, try some Google-fu. Project Details: For both tasks, you should first do some exploration and then conduct specified experiments or answer some questions based on the learning. The project should be straightforward and is more focused on exploring things and how they work.

Task 1: Protection of Memory via *mprotect()* (50% of grade):

This task is divided into two parts,

(A) Understanding *mprotect()*

(B) Experimenting with programs that use this call.

Part (A) Understanding Memory Protection (20% of grade):

For this task, students need to navigate to the M_protect directory in Desktop via terminal and compile both mprotect.c and without_mprotect.c via the gcc compiler [see https://www.wikihow.com/CompileaC-Program-Using-the-GNU-Compiler-(GCC)]. You may notice some warnings when you compile these two files via gcc compiler, which is due to usage of printf in the code. You can use -w option with gcc to stop displaying any warnings.

(NOTE: This is not usually recommended, if there are warnings in your C program, the best practice is to fix them. But for the sake of this project, carry on).

Run both the object files you generated and see the difference between their outputs. Do not worry if addresses are different on both the outputs. Look for SIGSEGV when executing the object generated by compiling mprotect.c program. You can refer to the man page of *mprotect* (http://man7.org/linux/manpages/man2/mprotect.2.html) to learn more about this call.

To answer:

1. Why do you get SIGSEGV when you execute the object code generated by compiling the mprotect.c program?
2. How does *mprotect()* protect memory ?
3. What is the minimum size requirement for this call?
4. What happens if you pass the value of the "len" argument as 1?
5. Sam is a programmer, and he needs to protect a 0x380 byte block starting at memory address 0x1234000 and ending at address 0x1234379. This block should be protected from overwriting by some other internal function. Can Sam use the *mprotect()* function in his C program to protect this memory space? Explain your answer.
6. Review and report what data is being protected in mprotect.c by the *mprotect()* function. Moreover, discuss if any function can perform read or write on the protected data. If the *mprotect* call is used in multiple instances, write your observations for each of them.

Part (B) Experimentation and Implementation (30% of grade):

To do: In folder M_protect, you can find the Exercise.c program file. We already have created a buffer of size 10*pages and assigned character 'A's to the whole buffer in the given code snippet. You are required to complete this C program based on instructions given below and then answer questions based on the execution of the completed code.

1. Write first *n* bytes of last two pages (9^{th} and 10^{th} page) with your first name where *n* is equal to number of characters in your first name (print to STDOUT).

2. Now, use *mprotect*() to allow read and write access on 7^{th} and 8^{th} page. Write your last name in first *n* bytes of 7^{th} and 8^{th} pages, where *n* is equal to number of characters in your last name and then try to read it. You should display it on output screen (print to STDOUT).
3. Now, use mprotect on 6^{th} and 5^{th} page and only give write access through *mprotect*() function. Now write your *n* character Georgia Tech user-id like (gpburdell3), where *n* equal to number of characters in your user-id. After writing try to read it. Can you read it?
4. Now, create a buffer of 2 pages and try to copy 7^{th} and 8^{th} page into it. Can you copy it, if not why?
5. Now try to copy 6^{th} page and 9^{th} page into the previously created buffer. Are you able to do so? If not, when your code hit SIGSEGV, is it copying 6^{th} or 9^{th} page? Explain your answer.

All the above steps should be followed in the same order, and updates should be made to the same script. Include in the report the screenshots of the print to STDOUT for all tasks 1-5.

Deliverables for Task 1:

1. Completed code (Exercise.c).
2. Report.pdf - Report.pdf should contain a section called Task 1 which should contain answers to the above asked questions for Part A and Part B.

Note: Do not zip up the deliverables. Upload them separately.

Task 2: Non-Executable Stack (50% of grade):

This task is also divided into two parts, (A) Understanding the importance of non-executable stack, and (B) Experimenting with vulnerable code with protected and unprotected stack.

Part (A) Understanding Stack Protection (30% of grade):

Review various mechanisms that are used to protect against buffer overflow exploits. More specifically, research the methods given below and write a brief explanation for each one of them. Each answer must be no more than a paragraph, and definitely no more than half a page of text (diagrams do not count to this limit).

1. Stack smashing via buffer overflow.
2. Stack canary.
3. NX (Non-Executable Stack).
4. Address space layout randomization (ASLR).

Part (B) Experiments with execution of code (20% of grade):

For this part of task 2, you need to locate the Stack_Protection directory via terminal, and there you will find vuln.c. You should review it carefully and then compile this source code into four different binaries with the following gcc options:

- gcc -g -O0 -fno-stack-protector -z execstack -o vuln-nossp-exec vuln.c
- gcc -g -O0 -fno-stack-protector -o vuln-nossp-noexec vuln.c
- gcc -g -O0 -z execstack -o vuln-ssp-exec vuln.c
- gcc -g -O0 -o vuln-ssp-noexec vuln.c

To answer:

(Each answer should be no more than a paragraph.)

1. Explaining the functionality of -fno-stack-protector and -z execstack options of gcc. Explain the differences you see in the binaries when the binaries are created with and without these options.
2. Which of the above four binaries can you exploit, by smashing the stack and overflowing a buffer?
3. Attempt to find the stack canary value in vuln-ssp-exec binary, by using a debugger like gdb. Does the canary value change or remain the same across multiple compilations/ executions? Post screenshots of your attempts.

Deliverables: Just a single “Report.pdf” for both Task1 and Task2.

Good luck!