CMPSC 443: Introduction to Computer Security

Spring 2022
Project 2: Buffer Overflows
Due: 11:59 pm (eastern time), March 14, 2022

February 22, 2022

1 Introduction

In this assignment, you will produce Buffer overflow attacks. First, you learn some attacks that invoke shared functions with arguments obtained from different places in memory (injected by you, or from environment variables, or from the hard coded strings in the code etc). Successful completion of this project heavily relies on correct understanding of stacks, heaps, program memory layout and a function's stack frame.

2 Prerequisite

Before attempting this project, it is advisable to brush on the basics of stack frame, memory layout of program, use of GDB Debugger and big-endian vs little-endian. To quickly brush through basics of GDB debugging, I'd recommend watching this GDB Debugger Tutorial - `https://www.youtube.com/watch?v=J7L2x1ATOgk&t=319s`.

3 Project Platforms

For this project, we will use the Linux virtual machine (VM), which is available at `https://drive.google.com/file/d/1OqvEfiIkl4DUrU_ZlBfVdPwRCnossri-/view?usp=sharing`. To get the VM running on your host machine, you will need **Oracle VM VirtualBox** downloaded and installed. The tar provided in the URL above consists of all necessary files (*.vbox, *.vmdk etc.) for Oracle VirtualBox to start the Ubuntu VM.
The exploits in this project have been tested on the same VM, therefore you must use the same environment for solving your tasks. Running the task binaries in a different VM or environment might not work.
Note: The password for the VM is posted on **CANVAS**. Once you login, all the necessary project files would be available on the Desktop itself.

4 Background

In the Virtual machine set-up, we have installed few tools and configurations that are essential for the completion of this project.

Address space layout randomization (ASLR) is set to Zero (Turned off) in the 32-bit Linux machine.

gdb-peda is a wrapper around the GDB debugger that has many features that help better visualize operational stack frames, variables, registers etc. when debugging C programs. This is already installed and set-up in the VM provided to you.

```
GDB command to show 100 lines of the stack starting from the $esp register is -
x/100xw $esp

GDB peda command provides a better visualization of the stack for the same purpose -
context stack 100
```

The exhaustive list of commands in gdb-peda is shown in this cheat sheet -
`https://github.com/kibercthulhu/gdb-peda-cheatsheet/blob/master/gdb-peda%20cheatsheet.pdf`
You are free to use any of these commands for help during your attacks.

GDB is a very popular and important GNU debugger that is used primarily to debug C programs. It is an essential tool used by computer science engineers. I highly recommend you to spend a couple of days to get hands-on with the tool if you haven't used it anytime in the past. Some of the most important commands that can come handy in this project are as follows -

```
print var OR p var
Prints the value of the local/global variable

p &var
Prints the address at which var is stored

p sample_function
Prints the pointer address to the method "sample_function"

p exit, p printf, p scanf etc.
Prints the pointer address to the standard C methods like printf, scanf, exit etc.

b 171
Adds a breakpoint at line no 171

run args
Starts a program within GDB with arguments

c
Continues the program until the next breakpoint

info locals
Gives information about all the local variables at the moment in the current frame.

info frame
Gives information about the current frame.
```

5 Code and Compiling

The initial code for the project is available at `https://drive.google.com/file/d/1AgZGNMczwECwAzxE9CQ8f8av8VSFBXG` `view?usp=sharing`. The same initial code is also present inside the VM on its Desktop. You have three groups of files given here -

The first group of files contains the **victim-binary** file which is compiled using its source code **cse443-victim-program.c**. Other files in this group contain utility functions, Makefile and README.txt to help you guide through the tasks. **You should NOT edit any of these files**.

```
victim-binary
cse443-victim-program.c
cse443-util-program.c
cse443-util-program.h
Makefile
README.txt
```

The second group of files correspond to each of the five tasks to be executed. They have some initial code written for your help. **You should edit these files** appropriately to successfully finish all the tasks.

```
cse443-task1-attack.c
cse443-task2-attack.c
cse443-task3-attack.c
cse443-task4-attack.c
cse443-task5-attack.c
```

The third group of files correspond to other intermediate files and payloads that are generated using the above two groups of files. For Example, the command "make task1-binary" will produce two intermediate files "task1-binary" and "cse443-task1-attack.o" from the source code "cse443-task1-attack.c".

```
"make task1-binary" produces task1-binary
"make task2-binary" produces task2-binary
"make task3-binary" produces task3-binary
"make task4-binary" produces task4-binary
"make task5-binary" produces task5-binary
"make victim-binary" produces victim-binary (This is not required as you wont make
    any changes to cse443-victim-program.c)
```

Similarly, running the task binaries should produce their corresponding payload files as follows.

```
"./task1-binary" produces task1-payload
"./task2-binary" produces task2-payload
"./task3-binary" produces task3-payload
"./task4-binary" produces task4-payload
"./task5-binary" produces task5-payload
```

NOTE: Remember! You are only supposed to edit the contents of files mentioned in Group2 to create corresponding attack binaries and payloads. Editing any file mentioned in Group1 might help you temporarily in your VM but we will evaluate your code with the original victim binary in a different setup. Then your code may not be creating successful attacks and this will lead to a 0 score in all tasks.

6 Exercise Tasks

The project consists of **five** tasks in total. Out of theses tasks, Task no. 4 is a BONUS task and can be treated as optional. Therefore, tasks 1,2,3,5 are enough to provide you with the full grade. Every task/attack follows similar execution flow at your end. Primarily, the **victim-binary** has at-least 5 buffer overflow vulnerabilities which you will take advantage of in each attack to generate unexpected and interesting results. To analyse these vulnerabilities, we have provided you the victim's source code i.e. **cse443-victim-program.c**!

The tasks are as follows.

1. In Task 1, you will build your very first light-saber by invoking the method **make_lightsaber**. Observe that the method **first_lightsaber** is invoked through the main function. It has many local variables including the variable **key** that is set to the value of another argument **argc**. You need to find the buffer overflow vulnerability in **first_lightsaber** and create a payload by packing enough A's at the beginning of your string. Find the location of the local variable key and set it to 0 using this overflow.
   Observe that with no attack, the value of key is 2 (because argc is 2) and thus the function **make_lightsaber** can be never legally called. But with the right overflow attack, you need to invoke the function **make_lightsaber** with the right value of **key=0**.
   Complete the program **cse443-task1-attack.c** to build a payload **task1-payload** using which the victim-binary prints the message - `Congratulations! You have successfully built your lightsaber.` A successful attack will look like the following.

```
cse443student@cse443student-VirtualBox:~/Desktop/lab2-handout$ ./victim-binary
    task1-payload
Welcome to not a Jedi Academy for CMPSC443
TASK1: Try to make your first lightsaber!
Congratulations! You have successfully built your lightsaber.

This is your lightsaber ID := 12937
```

2. In Task 2, you will use the Force to get access to the Shell! Observe that the method **force_shell** is invoked through the main function. It has many local variables including the function pointer variable denoted by **functionPtr** that is set to point to a method called **get_this**. You need to find the vulnerability in **force_shell** and create a payload by packing enough A's at the beginning of your string. Find the location of the local variable functionPtr and set it to the method called **and_get_that** using this overflow.
   Observe that with no attack, the value of **functionPtr** is set to the address of the function **get_this** and thus the function **and_get_that** can be never legally called. But with the right overflow attack again, you need to invoke the function **and_get_that**.
   Complete the program **cse443-task2-attack.c** to build a payload **task2-payload** using which the victim-binary prints the message - `Young Jedi! You got the shell - and give access to a new shell.` A successful attack will look like the following.

```
cse443student@cse443student-VirtualBox:~/Desktop/lab2-handout$ ./victim-binary
    task2-payload
Welcome to not a Jedi Academy for CMPSC443
TASK1: Try to make your first lightsaber!
TASK2: Try to get to the Shell. May the force be with you!

This is your lightsaber ID := 11254
```

```
Young Jedi! You got the shell.
$
```

3. In Task 3, you will need to complete your Jedi Combat Training by successfully invoking the method **complete_training**. Observe that the method **combat_training** is invoked through the main function. You need to find the vulnerability in **combat_training** and create a payload by packing enough A's at the beginning of your string. In this attack, you need to spot the return address of this method and successfully change it to the method **complete_training** using the overflow.
   Observe that with no attack, the method **combat_training** will simply return back to the main method from where it was initially invoked. But with the right overflow attack again, you need to return to the function **complete_training**.
   Complete the program **cse443-task3-attack.c** to build a payload **task3-payload** using which the victim-binary prints the message - `Well Done. Been recognized, your hard-working has!` A successful attack will look like below.

```
cse443student@cse443student-VirtualBox:~/Desktop/lab2-handout$ ./victim-binary
    task3-payload
Welcome to not a Jedi Academy for CMPSC443
TASK1: Try to make your first lightsaber!
TASK2: Try to get to the Shell. May the force be with you!
Try with greater force!
TASK3: Not prepared, you are!

This is your lightsaber ID := 16380
TASK3 has been successfully completed!
Well Done. Been recognized, your hard-working has!
Segmentation fault (core dumped)
```

4. In Task 4, you will need to collect 5 lightsabers to successfully finish the attack. Observe that the method **collect_lightsabers** is invoked through the main function. You need to find the vulnerability in **collect_lightsabers** and create a payload by packing enough A's at the beginning of your string. In this attack, you need to spot the return address of this method and successfully craft a chain of calls to the functions **get_blue_lightsaber** and **get_green_lightsaber** using the overflow.
   Observe that with no attack, the method **collect_lightsabers** will simply return back to the main method from where it was initially invoked. But with the right overflow attack again, you need to carefully craft a sequence of 5 calls to collect **3 BLUE lightsabers** and **2 GREEN lightsabers**.
   Complete the program **cse443-task4-attack.c** to build a payload **task4-payload**. A successful attack will look like the following.

```
cse443student@cse443student-VirtualBox:~/Desktop/lab2-handout$ ./victim-binary
    task4-payload
Welcome to not a Jedi Academy for CMPSC443
TASK1: Try to make your first lightsaber!
TASK2: Try to get to the Shell. May the force be with you!
Try with greater force!
TASK3: Not prepared, you are!
TASK4: Collect five lightsabers to complete this task!

This is your lightsaber ID := 10310
```

```
You got a blue lightsaber!

This is your lightsaber ID := 6192
You got a blue lightsaber!

This is your lightsaber ID := 10109
You got a blue lightsaber!

This is your lightsaber ID := 17883
You got a green lightsaber!

This is your lightsaber ID := 17596
You got a green lightsaber!
Segmentation fault (core dumped)
```

5. In Task 5, you will need to follow the light side and use your skills to print your name to successfully finish the attack. Observe that the method **follow_the_light** is invoked through the main function. You need to find the vulnerability in **follow_the_light** and create a payload by packing enough A's at the beginning of your string. In this attack, you need to print your name at the end of the last print statement in this method using the overflow.
   Observe that with no attack, the method **follow_the_light** will simply return back to the main method from where it was initially invoked. So it will only print - **Your Jedi Name is :-** . But with the right overflow attack, you need to invoke the C library function **printf** with a custom argument string i.e. your name and then invoke the system function **exit**.
   Complete the program **cse443-task5-attack.c** to build a payload **task5-payload** using which the victim-binary invokes the native **printf** function using your NAME as an argument at the right place in the code. A successful attack will look like below.

```
cse443student@cse443student-VirtualBox:~/Desktop/lab2-handout$ gdb -q victim-
    binary
Reading symbols from victim-binary...done.
gdb-peda$ run task5-payload
Starting program: /home/cse443student/Desktop/lab2-handout/victim-binary task5-
    payload
Welcome to not a Jedi Academy for CMPSC443
TASK1: Try to make your first lightsaber!
TASK2: Try to get to the Shell. May the force be with you!
Try with greater force!
TASK3: Not prepared, you are!
TASK4: Collect five lightsabers to complete this task!
TASK5: Print your name !

Your Jedi Name is :- GOUTHAM
[Inferior 1 (process 3072) exited with code 0107]
Warning: not running
gdb-peda$
```

   NOTE : Task 5 is very different from other tasks where we need to send a custom argument like **GOUTHAM** to the printf function. In Tasks 1-4 we only change return addresses and values of local variables to achieve our goal. Your attack will be successful within GDB debugger, however, the same payload may not help in performing a successful attack outside the GDB debugger.

Explain this in your report (refer to Questions section).

7 Questions

1. Draw the function's stack frame in **Task 2** to demonstrate the overflow. Use tools like Paint, Excel or any other online tool to show the stack frame. Refrain from providing diagrams drawn using hand.
2. Why does **Task 5** fail to run from the command line, but succeed when run in GDB debugger?
3. Why do **Tasks 1-4** run both from the command line and GDB debugger the same ?
4. Briefly identify and explain a viable defense mechanism to prevent the attack in **Task 3**. Precisely explain how this would prevent the attack you have crafted.

8 Deliverables

Please submit a tar ball containing the following:

1. **cse443-task*-attack.c** files (4 or 5 files), respective binaries **task*-binary** (4 or 5 files), payload files **task*-payload** (4 or 5 files).
2. A report in PDF containing: (1) Trace of output printed (e.g., shell invocation) from your execution of each case (2) Screenshot of each completed task and (3) Answers to project questions

9 Grading

The assignment is worth 200 points (+30 BONUS) total broken down as follows.

1. Answers to four questions (40 pts, 10 points each).
2. Packaging of your attack programs, binaries, payloads and the report in the "tar" file you submit. Your attack programs build without incident. (20 pts).
3. Completeness of report (20 pts).
4. Task 1 (20 pts), Task 2 (30 pts), Task 3 (30 pts), Task 4 (30 BONUS pts) and Task 5 (40 pts).