CSE 543: Computer Security

Fall 2024
Project 1: Buffer Overflows and Return Oriented Programming
Due: 11:59 pm (eastern time), Sept. 23, 2024

September 10, 2024

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 the hard coded strings in the code etc). Then you will try out other advanced attacks including heap overflows, shell code injection and return oriented programming attacks.

In the attacks that demand return oriented programming, we will use a tool called `ROPgadget` to extract gadgets from target binaries. The ROP attack combines shared library functions and ROP gadgets from the executable code to invoke more powerful and robust attack.

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 review on the basics of stack frame, memory layout of program, use of GDB Debugger and big-endian vs little-endian byte ordering. To quickly brush through basics of GDB debugging, I'd recommend watching this GDB Debugger Tutorial - https://youtu.be/-pKu42v_opk?si=Ndzg97MCqZxLJa6k.

3 Project Platforms

For this project, we will use a Linux Virtual Machine (VM) provided as follows:

If you have a machine (laptop or desktop) with an `x86_64` CPU, download the pre-configured VM from the Google Drive link: https://drive.google.com/file/d/1HFcn7oDkfzraws4LpZIsnPy1Nfqos8NT/view?usp=sharing For this VM, you will have to install the Oracle VirtualBox software and then use the .vbox file to run the VM.

For Apple Silicon platforms, download and decompress the VM from: https://drive.google.com/file/d/1aY-t11Di5rsyd-LXEtQrQGhWCkwhwYk5/view?usp=sharing. For this VM, you will have to install the UTM Mac software and then use the `.utm` file to open an existing VM in UTM.

The exploits in this project have been tested on these VMs, 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 will be available in CANVAS.

4 Background

In the VM 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 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.

pwndbg 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. You can learn about `pwndbg` here: https://pwndbg.re/. You are free to use any of these commands for help during your attacks.

Return-oriented Programming. The paper by Roemer et al. describes the principles and capabilities of return-oriented programming - https://doi.org/10.1145/2133375.2133377. The critical feature of return-oriented programming is that once adversaries gain control of the stack pointer and/or the memory around the stack pointer (and if the binary has enough code), then the adversary has a Turing-complete environment from which to craft exploits.

ROPgadget. `ROPgadget` is an open-source tool for extracting gadgets from binaries. For your convenience, `ROPgadget` has already been installed in the given VM. The basic use is below, but there are several options. We can generate gadgets from any binary, but we will use gadgets from the executable (victim).

```
./ROPgadget --binary victim1-binary > gadgets
```

The result is a collection of the possible gadgets available in the executable's code. The `--binary`, `--badbytes`, `--multibr`, and `--ropchain` flags will also be particularly helpful in some project attack(s).

5 Code and Compiling

Once you download your project tarball from the link provided in Section 3, you will have to copy the files into your VM. You can use the `rsync` command to copy the files from your local machine into the VM or vice versa.

The provided files contains all the victims' compiled binaries and their corresponding source code. The source code is provided to help you gain a better understanding of the attack targets. You do not need to compile any of them.

```
victim1.c, victim1-binary
victim2-binary
victim3.c, victim3-binary
victim4.c, victim4-binary
victim5.c, victim5-binary
victim6.c, victim6-binary
victim7.c, victim7-binary
victim8.c, victim8-binary
victim9.c, victim9-binary
```

NOTE: You are supposed to write 9 attack files to generated attack payloads for each task. For task 2 and task 8, write attack2.py and attack8.py respectively, such that running each file against its corresponding victim binary results in a successful attack. For example, ./victim2-binary ‘python3 attack2.py‘ (with python3 attack2.py enclosed in two backticks). For the remaining task, you should write the attack*.py files such that they generate the proper payload attack*-payload to used like ./victim1 attack1-payload.

6 Exercise Tasks

The project consists of **nine** tasks in total. Every task/attack follows similar execution flow at your end. Primarily, each victim*-binary has at-least one buffer overflow vulnerability which you will take advantage of in each attack to generate unexpected and interesting results. To analyse these vulnerabilities, we have provided you most of the victim's source code files!

All the tasks follow a subtle **Harry Potter** theme which would interest most of you. However, you don't need any background as such to understand what the task expects you to do. The following descriptions of each task should be self-explanatory.

The tasks are as follows.

1. In Task 1, you will wield your first wand by invoking the function called get_wand in the first victim. Observe that the method get_wand is invoked through the enter_academy 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 enter_academy and create a payload by packing enough garbage characters at the beginning of your string. Find the location of the local variable key in the program stack and set it to the required value calculated from your student ID using this overflow.

   To run victim1-binary, you must set you student ID in the SID environment variable and provide a file name as the first argument to read input from. With no buffer overflow, the value of key would be 2 and the binary output would look like the following -

```
$ SID=919191919 ./victim1-binary attack1-payload
Welcome to the Magical Academy - CSE543
Task 1 :- Procure your wand, Wizard!
WRONG KEY for procuring your wand. Keep Trying.
```

   However, a successful attack will look like the following -

```
$ SID=90xxxxxxx ./victim1-binary attack1-payload
Welcome to the Magical Academy - CSE543
Task 1 :- Procure your wand, Wizard!
Congratulations Young Wizard! You wield your first wand.

Wand ID = 11383
```

2. In Task 2, you will have to influence the sorting_hat method in the second victim to print your favourite House of Witchcraft and Wizardry. You may choose any one of the houses - Gryffindor, Hufflepuff, Ravenclaw or Slytherin. In task 2, unlike other tasks, you are not provided with the victim source code. However you are encouraged to analyze the victim2-binary with Ghidra. It is recommended that you install Ghidra on your host operating system rather than the VM.

   NOTE: The victim2-binary, unlike most targets in this project, requires a string argument instead of a payload file. This string is to be printed by the attack2.py python script appropriately.

With no buffer overflow, you will be allotted Slytherin and the output would look like the following -

```
$ ./victim2-binary `python3 attack2.py`
The Sorting Hat will now put you in one of the four houses!!!

I will randomnly place you in Slytherin.
```

However, a successful attack will look like the following -

```
$ ./victim2-binary `python3 attack2.py`
The Sorting Hat will now put you in one of the four houses!!!

Ravenclaw!!
```

3. In Task 3, you will open the Chamber of Secrets and get access to a shell by successfully invoking a method called `chamber()` in the third victim. Here, you do not want the program to execute `moaning_myrtle` function. Try exploring some of the local variables around the buffer overflow vulnerability to successfully call the chamber function.

   With no buffer overflow, you will be randomly allotted Slytherin and the failed attack would look like the following -

```
$ ./victim3-binary attack3-payload
This is Moaning Myrtle's bathroom.
Go away Wizard. There is nothing here!
```

   However, a successful attack will look like the following -

```
$ ./victim3-binary attack3-payload
This is Moaning Myrtle's bathroom.
You have opened the Chamber of Secrets. Well done!!
$
$ whoami
cse543-fa24
$
```

4. In Task 4, you will help Sirius Black escape without being caught by the Dementors with the help of the `buckbeak` method. In this task however, the method is not called from any other function. Therefore, you will need to figure out a way to use an overflow vulnerability to change the return address of a certain program stack frame to the desired value.

   With no buffer overflow, the failed attack would look like the following.

```
$ ./victim4-binary attack3-payload
Help Sirius Black escape without being caught by the Dementors.
Oh no! The dementors found you. Try once again.
```

   However, a successful attack will look like the following.

```
$ ./victim4-binary attack4-payload
Help Sirius Black escape without being caught by the Dementors.
Well done. You helped Sirius safely escape!
```

5. In Task 5, you will have to complete the Triwizard Tournament by successfully calling 3 methods in the correct order - `round_1()`, `round_2()` and `round_3()`! You should identify the vulnerability in the 5th victim and exploit it to chain the function calls according to the order specified. Clearly

follow how function frames are created in the program stack by adding necessary breakpoints in the GDB debugger.

A successful attack will look like the following -

```
$ ./victim5-binary attack5-payload
Welcome to the Triwizard Tournament.
Hurray! You slayed the dragon and found a golden egg!
Round Code = 11383

Well Done! You rescued your friend from the lake!
Round Code = 5886

Excellent! You completed the maze and found your ultimate champion cup!
Congratulations on winning the Triwizard tournament!
Round Code = 2777

[1]    27677 segmentation fault (core dumped)
```

6. In Task 6, the victim binary should print the message - `You are the chosen one.  Well done !!!` which is implemented by a `printf` call within the `prophecy` method. This task, however, needs better understanding of the way `malloc` works in C. The program variables now exist in both the stack and the heap.

   It would be helpful for debugging if you can use break points in `gdb` and look at what is being stored at any given address.

   A successful attack will look like the following.

```
$ ./victim6-binary attack6-payload
You are the chosen one. Well done !!!
```

7. In Task 7, you wield the Elder Wand to defeat the Dark Lord. To successfully complete this task, the victim7 should successfully print your name right after the usual execution of the binary.

   HINT: Your payload has to construct a whole new stack frame that calls the system `puts` function with the correct return addresses and arguments (your name).

   A successful attack will look like the following.

```
$ ./victim7-binary attack7-payload
Its time for you to wield the Elder Wand!
Congratulations. There is a new owner of the Elder Wand


Your Name is :-
ALI
[1]    29052 segmentation fault (core dumped)
```

8. In Task 8, you have to invoke a shell by identifying the right buffer overflow vulnerability in the `victim8-binary` and supplying the correct payload as a string. Observe that the victim source code never invokes system shell in any manner. **You should not use ROP Gadgets for this task**. Figure out a way to make the system call with the `/bin/sh` argument to successfully get the shell.

   HINT: The victim binary is compiled with the options `execstack` and `-fno-stack-protector`. This means that your program's stack may contain data that is treated as code and can be executed. Your shell code could invoke the system call.

NOTE: The `victim8-binary`, unlike most targets in this project, requires a string argument instead of a payload file. This string is to be printed by the `attack8.py` python script appropriately.

A successful attack will look like the following -

```
$ ./victim8-binary `python3 attack8.py`
Get Shell from here!
$
$ whoami
cse543-fa24
$
```

9. In Task 9, similar to the previous task, you have to invoke a shell by identifying the right buffer overflow vulnerability in the `victim9-binary` and supplying the correct payload as a string. You can use Return Oriented Programming concepts and take help of `ROPgadget` tool to achieve this attack successfully.

   HINT: The victim binary is compiled with the options `noexecstack` and `-fno-stack-protector`. Therefore, any data that resides in the program stack frame will be considered Non-executable. The binary can be called as "**DEP Enabled**" where DEP stands for Data Execution Prevention.

   NOTE: The `victim9-binary`, unlike the previous task, requires a payload file. The payload should be generated by the `attack9.py` python script appropriately.

   A successful attack will look like the following -

```
$ ./victim9-binary attack9-payload
Get Shell from here!
$
$ whoami
cse543-fa24
$
```

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. Draw the heap diagram in **Task 6** before and after the attack. Use tools like Paint, Excel or any other online tool to show the stack frame. Refrain from providing diagrams drawn using hand.
3. Were you able to exploit the vulnerability successfully and complete the attack in **Task 9** ? Did you use ROP Gadgets to carry out the attack ? If yes, specify the gadgets that you used. Otherwise, explain how did you complete the attack.

8 Deliverables

Phase 1: Please attend one of office hours on Sept. 10 or Sept. 12 to show the following:

1. Working VM set-up in your local machine should be shown to the TA.

Phase 2: Please submit a .zip file containing the following (file name should be your student ID, e.g. `919191919.zip`):

1. `attack{1..9}.py` files (9 files).
2. A report in PDF containing: (1) Screenshots of output printed (e.g., shell invocation) from your execution of each case and (2) Answers to project questions.

9 Grading

The assignment is worth 100 points total broken down as follows.

1. Successful installation of VM in your machine (5 points). [**You should complete this task by Sept 12th, 2024**]
2. Answers to three questions (15 pts, 5 points each).
3. Packaging of your attack programs, payloads and the report in the "zip" file you submit. Your attack programs build without any errors. (5 pts).
4. Completeness of report (9 pts).
5. Task1-3 (5 pts each), Task4-6 (7 pts each), Task7-9 (10 pts each).

Final Note: There will be a demo scheduled for each student individually with the TA wherein you would be asked to explain 1-2 tasks from the project in random. Keep your VM and attacks saved even after the project submission. If you are unable to explain the tasks in question, it can likely affect your points in this project. Each student should work on the project individually.