CSE 543: Computer Security

Fall 2021
Project 1: Return Oriented Programming and Heap Overflow
Due: 11:59 pm (eastern time), September 20, 2021

August 31, 2021

1 Introduction

In this assignment, you will produce overflow and code-reuse attacks. First, you learn some attacks that invoke shared library 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). You will also write a ROP attack that combines shared library functions and ROP gadgets from the executable code to invoke more powerful and robust attacks. The overflow attacks will include stack overflow and heap overflow. We will use available tool (ROPgadget [1]) to extract gadgets from the executable. 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, heap allocation, memory layout of program, use of GDB, how to read assembly code, and big-endian vs little-endian

3 Background

Return-oriented Programming. The paper by Roemer et al. describes the principles and capabilities of return-oriented programming [2]. 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 [1] is an open-source tool for extracting gadgets from binaries. For your convenience, ROPgadget has already been installed in the given VM (see Section 4 later).

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 cse543-p1 > gadgets
```

The result is a collection of the possible gadgets available in the executable's code.

Procedure Linkage Table (PLT). The PLT also provides some useful options for launching ROP attacks. The PLT provides stub code for invoking the library calls used by the executable. Since library code does useful things, such as invoking system calls, invoking this code via the PLT is often desirable. You can view the PLT stub code by disassembling the executable.

```
objdump -dl task12 | less
```

You can then search for "plt" to locate the stub code for a variety of library calls from the executable code. We will use a variety of PLT functions. You should also look at the man page of `objdump`. Certain options like R might make it easier to locate appropriate address.

Learning about ROP Attacks. One video [3] provides background and demonstrates examples for building ROP exploits. The video is one hour long, but the first 30 minutes or so is ROP background. The second 30 minutes is useful hands-on information for launching ROP attacks. This video demonstrates how to find and invoke library functions via the PLT, which is fundamental to our approach.

Note that program input functions are sensitive to some byte values. A zero-byte will terminate the read (fscanf). However, other byte values such as 8 and 15 may cause fscanf to terminate. Fortunately, you should not need these in the way the project is formulated.

Running PLT Payloads. You can use two types of payloads use in this project: PLT calls and ROP gadgets. Invoking each is slightly different. Launching a PLT stub essentially calls the library function. Therefore, to invoke any PLT stub you will specify the PLT stub address on the stack at the return address. You will also have to build the rest of the stack as the compiler work for a function call. Above the PLT stub address will be the address of the instruction to run when this function returns (i.e., the return address for the PLT stub) and then the arguments to the targeted function (in order from first to last). Arguments may be values or pointers to data values. Our attacks will mainly need pointers placed on the stack.

This is the format of the stack for any function call. When the stack pointer points to the PLT stub address on a return instruction, the PLT stub function will be invoked and run to completion using the arguments above the return address of the PLT stub, then the code referenced by the return address will be run. The video [3] has several examples of this to help you.

To choose the PLT functions to use, use `objdump` to find the PLT stub addresses as described above and place those addresses on the stack.

Choosing and Running Gadgets. In this project, you can use gadgets to remove arguments to PLT stubs from the stack to call the next PLT stub. The pop-ret gadgets are also discussed in the video [3]. See `http://x86.renejeschke.de/html/file_module_x86_id_248.html` for information on pop. This website has specs for most x86 instructions, but we will only depend on a few.

To launch a gadget, the return address on the stack should be assigned the address of the gadget (first instruction of the gadget). Gadgets may use values on the stack as well in order. When a gadget returns, whatever is present at the current stack pointer will be executed next.

Once you have determined which gadgets you want to use, a challenge is to invoke them. While ROPgadget provides the address of the gadgets in the victim executable. Since the victim is loaded at the expected address, these gadget addresses may be used directly.

Crafting Exploits. Crafting ROP exploits is a non-trivial exercise, requiring an understanding of the memory layout of the program, particularly the stack. Key to understanding memory layout will be the debugger.

Please pay close attention to the commands used in the debugger, as you will want to utilize the same commands to create a split layout showing the program and the assembly view (layout split), step one instruction at a time (si), and print the stack. Other useful gdb commands are "print" for displaying the values in memory (`''p var''` to print the value of variable "var") and `''info register''` to print the values of registers, such as the stack pointer in esp (`''i r esp''`), and `''x/16wx $esp''` to print 16 bytes from the esp address. See `http://users.ece.utexas.edu/~adnan/gdb-refcard.pdf` for more information. The idea is that you want to overwrite the return address on the stack with the address of the first instruction (gadget or PLT stub) that you want to run. Then, you prepare the rest of the stack with the arguments and instructions to run when your gadgets complete.

4 Project Platforms

For this project, we will use the Linux virtual machine (VM), which is available at `https://pennstateoffice365-my.sharepoint.com/:u:/g/personal/njk5270_psu_edu/Ef6wA1E67f1DhKgatoUoVL8B9tIen4Q0X1PrXRGgmkkMaA?e=zZSf6D`. I have tested the exploit on the same VM. You must use the same machine for developing your exploits. To get the VM running on your host machine, you will have to download and install VirtualBox. The necessary tar files containing all the .c files and binary should be present on the Desktop of the VM.

Note that the binary is compiled for this particular VM, so whether the binary even runs on another machine is questionable.

5 Exercise Tasks

The initial code for the project is available at `https://syed-rafiul-hussain.github.io/index.php/teaching/cse543-f21/projects/cse543-f21-p1.tar.gz`. This code consists of two groups of files: (1) the victim programs cse543-victim-task12, cse543-victim-task3, cse543-victim-task4,cse543-victim-task5 (source code files) along with their respective binaries task12. task3. task4, and task 5 and (2) the attack programs that produces buffers to attack return addresses cse543-task1.c, cse543-task2.c, cse543-task3.c, cse543-task4.c and cse543-task5.c and supporting code. `‘‘make ’’` should build attack programs corresponding to each attack goal. You do not need to build the binary.

You should learn how to accomplish each of the tasks with the binary (i.e., from the tar file). See grading below.

For task 1 and task 2 you have access to a *not* Jedi school training program. Your first task is to print your name below a lightsaber id (which determines the color of the lightsaber). In second task you have two goals, to get four lightsabers under one name and use the *force* to get access to a new shell. For task 3 and task 4 you have access to a program that can open the chamber of secrets. You use overflow techniques to invoke the right function that will open the door for you.

Taks 5 is optional, all you have to do is get logged in and tell us how and why your were able to log in without using any password.

The project consists of the following tasks.

1. Write the program cse543-task1.c to build a payload `task1-payload` so that it prints the string of your name like "`Name: <YOUR NAME>!`". This stromg will be part of your payload.
   Your payload should consist of four parts.
   - First, you should encode the address of the `printf` stub from the PLT at the return address.
   - Second, when `printf` returns, the program should `exit`. There is a PLT stub for this too.
   - Third, `printf` should be given an argument that is the address of the hardcoded string for "`Name: <YOUR NAME>!`". You have to figure out where the string can be stored.
   - Fourth, you need to add the string value "`Name: <YOUR NAME>!`"to wherever you decided to store the string.

You will need to determine how far the return address is from the beginning of the input domain buffer you want to overflow. You are requested to fill the space up to the return address with 'A's (It will be easier for you to debug if you know the hex value of 'A').

Once you have all the four parts ready and you know the buffer overflow value, You can start to construct the payload. The program includes a C macro `pack` that you can use to add 4-byte values (addresses of instructions and arguments) into the payload.

Since "`Name: <YOUR NAME>!`" is not an address you can use memcpy to add this to the payload. The function write to file is available to write the payload to a file. It takes 4 arguments: (1) name of the file; (2) buffer to write to file; (3) size of the buffer in bytes; and (4) whether to clear the file before writing or whether to append the buffer to the existing file.

Run the victim program using the generated payload under `gdb` and from the command line. It will print "`Name: <YOUR NAME>!`" from only gdb (Think Why?). The output under gdb should look like below (may not be exactly the same, but important that "`Name: <YOUR NAME>!`" appears). The program will segmentation fault from the command line (Why?).

```
Starting program: /home/cse543/Desktop/cse543-f21-p1/task12 task1-payload
Welcome to not a Jedi Academy for CSE543

<<<<<<<<<CSE 543 Project 1>>>>>>>>>

Your Jedi Name:
This is your lightsaber ID := 1511
Neeraj[Inferior 1 (process 10258) exited with code 0116]
Warning: not running
```

2. In this task, you have two goals (1) Collect more lightsabers by printing not just one, but four lightsaber ID under one Jedi name and (2) Get access to launch new shell (/bin/sh) using the functions `get_this` and `and_get_that`.
   You have to have to achieve (1) by writing a program in the file cse543-task2.c to build a payload to print `This is your lightsaber ID = xxx` four times on your terminal with different lightsaber IDs. After packing your payload by enough 'A's to start overflow you can give the address of the function that generates lightsaber IDS as return address. Do it 3 more times.
   For (2) You need to overwrite the return address and redirect the execution to `get_this` and `and_get_that` functions, respectively. The main challenge is that the`get_this` and `and_get_that` functions have arguments. You will, therefore, need to use gadgets in the task to clear (pop) the arguments from `get_this` to `and_get_that` or the exit stub to set the stack pointer correctly for the next function/stub. The video [3] demonstrates this as well.

   The output with command line should look like below (may not be exactly the same).

```
cse543-VirtualBox:~/Desktop/cse543-f21-p1> ./task12 task2-payload
Welcome to not a Jedi Academy for CSE543

<<<<<<<<<CSE 543 Project 1>>>>>>>>>

Your Jedi Name:
This is your lightsaber ID := 12721

This is your lightsaber ID := 2851

This is your lightsaber ID := 4768
This is your lightsaber ID := 18611
Hello There! General Kenobi. You got the shell.
$
```

3. In this task you will create a payload to open the chambers of secret via function `parseltongue`. To achieve this you will have to overflow heap instead of stack. All you need to do is first identify how many 'A's you have to use to overflow the buffer so that `f->fp` points to `parseltongue`.

   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. Commands like `info proc mappings, x/<integer>wx <address>` can be very handy. Use man pages to know more about these kind of commands.

   The output with command line should look like below (may not be exactly the same).

   ```
   cse543-VirtualBox:~/Desktop/cse543-f21-p1> ./task3 ‘cat task3-payload‘
   Enter
   sss Open! sss
   The chamber is now open
   The end
   ```

4. The goal of this task is similar to the previous task i.e. to open the chambers of secret via function `parseltongue`. Difference being you will have to decide what to overflow and point to what address. This task tests the understanding on how `malloc()` allocates memory in heap. For this task you do not have to build a payload. You have to craft a payload on the command terminal itself. You use `$echo with option -ne` and ` (back tick) to achieve that. Example: ./binary "‘/bin/echo -ne 'AABBCCDDEEFF⟨*address_in_right_format*⟩'‘"

   The output with command line should look like below (may not be exactly the same).

   ```
   cse543-VirtualBox:~/Desktop/cse543-f21-p1> ./task4 <payloads using /bin/echo>
   sss Open! sss
   The chamber is now open
   The End!
   ```

5. **Optional Bonus** The goal of this task is to successfully sign in the program ./task5. In order to do that you will have to change the value of `admin->userID`. You must understand what happens in the program memory when `malloc` and `free` are called. Looking at *man* pages of certain functions used in task5 might be useful. The last three lines of output in command line should look like below (may not be exactly the same).

   ```
   SignIn
   Welcome! you have signed in.
   <<< Task 5 is now complete >>>
   ```

6 Questions

1. Why do Tasks 1 fail to run from the command line, but succeed when run in gdb?
2. Draw the function's stack frame from task1 and task 2 to demonstrate the overflow
3. Identify a defense that would prevent the attack in Task 1-4. Precisely describe how that defense would prevent the attack.

4. Specify the gadgets used and their purpose for Task 2.
5. Draw diagram for heap for task 3 as well as task 4 before and after the attack What is the best way to ensure safe string processing in the C language?
6. (For bonus task 5) What vulnerability were you able to exploit and how? What are any constraints on the string used (if any)

7 Deliverables

Please submit a tar ball containing the following:

1. Your exploit programs (view readMe.txt for more info)
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

8 Grading

The assignment is worth 100 points (+15 bonus) total broken down as follows.

1. Answers to five questions (20 pts)
2. Packaging of your attack programs using "make tar" and inclusion of that tar file and the report in the tar file you submit. Your attack programs build without incident. (10 pts)
3. Completeness of report (10 pts)
4. Task 1 (15 pts), Task 2 (15 pts), task 3 (10 pts), task 4 (20 pts) and task 5 (15 bonus pts).

References

[1] J. Salwan, *ROPgadget*.

[2] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 15, no. 1, Mar. 2012. [Online]. Available: https://doi.org/10.1145/2133375.2133377

[3] *Return oriented exploitation (rop)*, https://www.youtube.com/watch?v=5FJxC59hMRY.