CSE 543: Computer Security

Fall 2020
Project 1: Return Oriented Programming
Due: 11:59 pm (eastern time), September 21, 2020

November 27, 2020

1 Introduction

In this assignment, you will produce a few return-oriented programming (ROP) attacks. First, you learn some attacks that invoke shared library functions with arguments obtained from different places in memory (injected by you, from environment variables, and from the hard coded strings in the code). Then, you will write a ROP attack that combines shared library functions and ROP gadgets from the executable code to invoke more powerful and robust attacks. We will use available tools (ROPgadget [1]) to extract gadgets from the executable.

2 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 3 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-p2 > 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 cse543-p2 | 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.

Launching Buffer Overflows. The idea in all the return-oriented attacks in this project is to overwrite the return address of the `generate_lottery_ticket` function given the available buffer overflows in that function. You are given the source code to the program (`generate_lottery_ticket`

is in `cse543-victim.c`), which contains at least one buffer-overflow vulnerability. Hopefully, this is not the way you gathered input. Use the variable `name` to overflow the return address. It is closest to the return address. Fortunately, you can use the same flaw for each task in the project. To design a buffer overflow attack, you must determine where your buffer is in memory when the program is running and where the return address is in memory. To do this, you have to find the return address. Using objdump the return address is the address of the instruction after the call to `generate_lottery_ticket`. Then, you have to run the program under gdb to determine location of the buffer you want to overflow relative to the return address of the function on the stack. Your input to run the victim program follows "(gdb)" below.

```
cse543-VirtualBox:~/cse543-f20-p1> gdb cse543-p1
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from cse543-p1...done.
gdb-peda$ run test-payload
```

We run the `cse543-p1` which takes an `<input-file>` as input. The `<input-file>` contains name(s) of the people for whom lottery tickets (check `main` and `generate_lottery_ticket` functions) are to be generated. You will write programs to produce payloads to exploit the buffer overflow.

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. There are two types of payloads we will 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 -dl cse543-p1` to find the PLT stub addresses as described above and place those addresses on the stack.

Choosing and Running Gadgets. In this project, you will mainly 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 use of 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.

3 Project Platforms

For this project, we will use the Linux virtual machine (VM), which is available at `https://drive.google.com/file/d/1_3JO_d3I1hAPVi7T71NmYmaSP159Ze0L/view?usp=sharing`. 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 download and install VirtualBox, and then run cse543.vbox.

Note that the binary is compiled for this particular VM, so whether the binary even runs on another machine is questionable. The exploits will not correlate across different platforms with a very high likelihood.

4 Exercise Tasks

The initial code for the project is available at `https://syed-rafiul-hussain.github.io/index.php/teaching/cse543-f20/projects/cse543-f20-p1.tar.gz`. This code consists of two groups of files: (1) the victim program cse543-p1 (binary and source code files) and (2) the attack programs that produces buffers to attack return addresses cse543-*-attack.c and supporting code. `‘‘make all’’` 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.

The project consists of the following tasks.

1. Write the program cse543-buf-attack.c to build a payload to print the string "`You_are_awesome!`" that will be included in the 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 "`You_are_awesome!`" on the **stack**. Fourth, you need to add the string value "`You_are_awesome!`" to the stack.

   You will need to determine how far the return address is from the beginning of the input domain buffer you want to overflow. This will be the same for all exploits. You are requested to fill the space up to the return address with 'x's.

   Then 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 "`You_are_awesome!`" 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 "`You_are_awesome!`" from only gdb. The output under gdb should look like below (may not be exactly the same, but important that "`You_are_awesome!`" appears). The program will segmentation fault from the command line.

```
cse543-VirtualBox:~/cse543-f20-p1> gdb -q cse543-p1
Reading symbols from cse543-p1...done.
gdb-peda$ r buf-payload
Starting program: /home/cse543/cse543-f20-p1/cse543-p1 buf-payload
Hello, CSE543


 ==== Welcome to get the lottery ticket. Wish you the best of luck! ====

Name:
Lottery ticket ID = 5326
You_are_awesome![Inferior 1 (process 5976) exited with code 0131]
Warning: not running
```

2. Next, you will write a program in cse543-env-attack.c to build a payload to print a string from an environment variable. I will test under tcsh, so please verify that you are running the right shell.

   `printenv | grep SHELL`

   should be `/bin/tcsh`.

   Then please assign the following environment variable.

   `setenv CSE543 ‘‘You_are_awesome!’’`

   Then run the victim under gdb to find the location of this environment variable in the program's memory. The video [3] shows how this is done. You can execute `x/300s $esp` (or `x/400s $esp`) to print the next 300 (or 400) strings from the stack pointer in gdb. You should be able to find the string in there.

Use the memory address of the string from the environment variable as the argument to printf in a second payload construction in cse543-env-attack.c.

Run the victim program using the generated payload under gdb and from the command line. It will only print "`You_are_awesome!`" from gdb. The output under gdb should look like below (may not be exactly same). The program will segmentation fault from the command line.

```
Hello, CSE543

==== Welcome to get the lottery ticket. Wish you the best of luck! ====

Name:
Lottery ticket ID = 5406
You_are_awesome![Inferior 1 (process 6099) exited normally]
```

3. In this task, you write a program in the file cse543-system-attack.c to build a payload to execute a shell command using the function system.

   Instead of printf, you will find the PLT stub for the system function and invoke that to call "`/bin/ls -la`" as the command. The string "`/bin/ls -la`" happens to already be in the victim binary (i.e., hardcoded in the code). You can find strings and their offsets in a binary using the strings command.

   ```
   strings -t x cse543-p1
   ```

   You will have to compute the actual address of the "`/bin/ls -la`" string in the binary by adding the offset from the base address of the executable. Run the victim under gdb and from the command line. The output with command line should look like below (may not be exactly the same).

```
cse543-VirtualBox:~/cse543-f20-p1> ./cse543-p1 sys-payload
Hello, CSE543


 ==== Welcome to get the lottery ticket. Wish you the best of luck! ====

Name:
Lottery ticket ID = 15329
total 108
drwxrwxr-x 2 cse543 cse543 4096 Aug 31 10:52 .
drwxr-xr-x 21 cse543 cse543 4096 Aug 31 00:10 ..
-rw------- 1 cse543 cse543 885 Aug 30 22:58 cse543-buf-attack.c
-rw------- 1 cse543 cse543 819 Aug 30 22:58 cse543-env-attack.c
-rwxrwxr-x 1 cse543 cse543 18284 Aug 30 23:40 cse543-p1
-rw-rw-r-- 1 cse543 cse543 809 Aug 30 22:58 cse543-repeat-attack.c
-rw-rw-r-- 1 cse543 cse543 764 Aug 30 22:58 cse543-rop-attack.c
-rw------- 1 cse543 cse543 762 Aug 30 22:54 cse543-string-attack.c
-rw------- 1 cse543 cse543 827 Aug 30 22:58 cse543-system-attack.c
-rw------- 1 cse543 cse543 1789 Oct 3 2017 cse543-util.c
-rw------- 1 cse543 cse543 241 Oct 3  2017 cse543-util.h
-rw-rw-r-- 1 cse543 cse543 1929 Aug 30 22:18 cse543-victim.c
-rw------- 1 cse543 cse543 2102 Aug 31 10:03 .gdb_history
-rw------- 1 cse543 cse543 1763 Aug 30 23:40 Makefile
-rw------- 1 cse543 cse543 12288 Aug 22 10:28 .Makefile.swo
```

```
-rw------- 1 cse543 cse543 12288 Aug 22 10:23 .Makefile.swp
-rw-rw-r-- 1 cse543 cse543  49 Aug 31 10:02 peda-session-cse543-p1.txt
-rw------- 1 cse543 cse543  84 Aug 31 10:58 sys-payload
-rw-rw-r-- 1 cse543 cse543  16 Aug 22 12:53 test-payload
```

4. In this task, you want to maximize your probability of winning a lottery and you want to print not just one, but five lottery ID under your name. But you have to achieve this with a single invocation of `generate_lottery_ticket` function in `cse543-p1` program. You write a program in the file cse543-repeat-attack.c to build a payload to print `Lottery ticket ID = xxx` five times on your terminal with different ticket IDs.

   You will essentially build a payload that will execute the `print_lottery_id` function five times. The output with command line should look like below (may not be exactly the same).

```
cse543-VirtualBox:~/cse543-f20-p1> ./cse543-p1 repeat-payload
Hello, CSE543

 ==== Welcome to get the lottery ticket. Wish you the best of luck! ====

Name:
Lottery ticket ID = 7477

Lottery ticket ID = 2054

Lottery ticket ID = 19431

Lottery ticket ID = 11846

Lottery ticket ID = 10007
```

5. In this task, you write a program in the file cse543-rop-attack.c to build a payload to launch a new shell (/bin/sh) using the functions `catch_me_if_you_can_1` and `catch_me_if_you_can_2`. The output with command line should look like below (may not be exactly the same).

```
cse543-VirtualBox:~/cse543-f20-p1-v3> ./cse543-p1 rop-payload
Hello, CSE543

 ==== Welcome to get the lottery ticket. Wish you the best of luck! ====

Name:
Lottery ticket ID = 2311
You are amazing and cool. You got the shell.
$
```

   You need to overwrite the return address and redirect the execution to `catch_me_if_you_can_1` and `catch_me_if_you_can_2` functions, respectively. The main challenge is that the `catch_me_if_you_can_1` and `catch_me_if_you_can_2` functions have arguments. You will, therefore, need to use gadgets in the task to clear (pop) the arguments from `catch_me_if_you_can_1` to `catch_me_if_you_can_2` or the exit stub to set the stack pointer correctly for the next function/stub. The video [3] demonstrates this as well.

   Run the victim under gdb and from the command line. Answer the question below about the results.

5 Questions

1. Why do Tasks 1 and 2 fail to run from the command line, but succeed when run in gdb?
2. Describe the results of Task 3 when run from the command line. Why did Task 3 succeed when Task 1 and 2 failed?
3. Describe an alternative way to perform Task 3 given the victim code.
4. Identify a defense that would prevent the attack in Task 5. Precisely describe how that defense would prevent the attack.
5. Specify the gadgets used and their purpose for Task 5.

6 Deliverables

Please submit a tar ball containing the following:

1. Your exploit programs (in our tar ball built with 'make tar')
2. Trace of output printed (e.g., shell invocation) from your execution of each case
3. Answers to project questions

7 Grading

The assignment is worth 125 points total broken down as follows.

1. Answers to five questions (25 pts)
2. Packaging of your attack programs using "make tar" and inclusion of that tar file and the questions in the tar file you submit. Your attack programs build without incident. (10 pts)
3. Task 1 (15 pts), Task 2 (10 pts), task 3 (20), task 4 (20) and task 5 (25 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.