CSE 543 - Fall 2016 - Assignment 3: Return-oriented Programming

1 Dates

- **Out:** *November 2, 2016*
- **Due:** *November 17, 2016*

2 Introduction

In this assignment, you will produce a few return-oriented programming (ROP) attacks. First, you will use available tools (ROPgadget [3]) to extract gadgets from programs. Using these gadgets, you will build exploits for provided programs that will enable you to launch return-oriented attacks. The idea is to build different types of ROP attacks to see how to use gadgets to enable different types of exploit behaviors.

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 [3] is an open-source tool for extracting gadgets from binaries. Obtain the ROPgadget codebase from `https://github.com/JonathanSalwan/ROPgadget`. See the `README.md` file for directions on loading and executing. The basic use is below, but there are several options. We can generate gadgets from any binary, but libc provides far more gadgets than the victim program, so we will explore using those gadgets.

./ROPgadget.py /lib32/libc-2.23.so ¿ gadgets

The result is a collection of the possible gadgets available in the libc code.

Choosing Gadgets There are many, many gadgets in libc, so a question is which gadgets should you be looking for. For example, you may need gadgets to reset the stack pointer. Thus, any gadget that loads a value in the stack pointer register (`%esp`) would be useful for that. Use the ROP paper for guidance and we will discuss in class.

Crafting Exploits from Gadgets Crafting ROP exploits from a set of gadgets is a non-trivial exercise, requiring an understanding of the memory layout of the program and the Intel x86 instruction set (at least to some extent).

A video [1] demonstrates how one may use ROPgadget and other tools such as *gdbtui* to generate exploits and view them in action (and debug mistakes at that level).

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.

Finding Gadgets in the Binary 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 library, the library is dynamically linked into the process's address space at an address chosen by the system. The question is how do you find where the library is loaded. The `/proc` filesystem helps here. Type `cat /proc/$PID/maps` where `$PID` is the process ID of the `file-victim` process. You will see something like:

```
08048000-08049000 r-xp 00000000 08:01 3547850  .../cse543-file-victim
08049000-0804a000 rw-p 00001000 08:01 3547850  .../cse543-file-victim
f763f000-f7640000 rw-p 00000000 00:00 0
f7640000-f779e000 r-xp 00000000 08:01 1707696  /lib32/libc-2.23.so
...
```

The first two lines refer to the executable's (`cse543-file-victim`) code (see the execute permission is set) and data. The address range identifies the beginning and end of these memory segments. The fourth line (in this case) refers to libc's code segment. Use the base address plus the gadget address to compute the location of the gadgets.

Defenses (Turning Them Off) Systems already deploy a variety of defenses that restrict attacks, including ROP attacks. For example, StackGuard protects the return address from being overwritten without detection and ASLR moves the location of the stack and libraries to prevent useful code injection or knowledge of gadget target locations.

In this project, we will turn off these defenses to make our task a bit easier. For example, you will notice in the Makefile there is the flag `-fno-stack-protector`, which removes the stack canaries that are now set by default.

Also, for some project tasks, you should disable the ASLR on your system. First, record the value of the file `/proc/sys/kernel/randomize_va_space` (mine was "2") using `cat`. Then invoke the following command to reset the value to 0.

```
echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

You should have sudo access on your evaluation machines to perform the attacks. Please let me know if this is a problem for your team.

4 Research Platforms

For this project, we have created a set of VMs, one for each team. I will provide the user names, initial passwords, and VM IP addresses for each VM separately. Please change your passwords by class time on Th, as they are weak.

5 Problem Statement

Your task is to produce ROP attacks against the given program (cse543-file-victim) for buffer overflows that overwrite the return address. You will produce exploits from libc code to perform ROP tasks described below. The vulnerable code has an obvious buffer overflow and a function called `shell` that launches a shell that we will use in some of the exploits.

6 Exercise Steps

The initial code for the project is available at `http://www.cse.psu.edu/~tjaeger/cse543-f16/p3-rop.tgz`. This code consists of two main files: (1) the victim program `cse543-file-victim.c` and (2) the program that produces buffers to attack return addresses `cse543-buf-attack.c`. "make all" should build executables corresponding to each file.

The current code in `cse543-buf-attack.c` produces a buffer that performs a return-oriented attack using the *shell* function in the victim program, `cse543-file-victim.c`. In this case, the attack stack consists only of the address of the shell function in the code segment - we jump directly there to launch the attack. In this case, the following value is non-zero (for argument i) already, so we need not do anything else to test that `cse543-buf-attack` produces a buffer will open a new shell when run on your virtual machine.

In this project, you will design multiple return-oriented attacks using gadgets *from libc* to explore the design of return-oriented attacks.

The project consists of the following steps.

1. With ASLR off, launch the function `shell()` using gadgets in libc in the following manner. First, using the gadgets in libc, change the stack pointer to a new stack containing the address of `shell()` to launch a new shell. You must create the new stack as part of the attack also. You may create the new stack in any writeable data area. Ensure that the gadget exits cleanly from the exploit. Call this program *cse543-buf-attack.shell.c*.

2. With ASLR off, Use an indirect jump gadget to launch `shell()`. Call this program *cse543-buf-attack.jump.c*.

3. With ASLR off, perform a return-to-libc attack to use `system()` to run `/bin/bash`. Use the existing string `/bin/bash` assigned to the environment variable SHELL. Call this program *cse543-buf-attack.system.c*.

4. Launch at least one of the attacks above with ASLR enabled. This will require some research of how attacks are performed given that ASLR is enabled. In this case, you may leverage other gadgets than the ones in libc. Call this program *cse543-buf-attack.aslr.c*.

The metric for this project is the *length of the gadget chains* used to achieve all 4 attacks.

6.1 Teams

Below are the teams. I will provide information on your VMs and logins later.

1. Sheatsley, Frank Liu, Latkar

2. Veerabhadraiah, Sharma, Brotzman-Smith

3. Papernot, Jonas Wang, Kaul
4. Sridhar, Sharp, Huhman
5. Sun, Riegel, Luo
6. Pahadia, Wheatman, Kang-Lin Wang
7. Adavi, Zhao, Sha Liu
8. Bandyopadhyay, Chun-Yi Liu, Karamchandani
9. Acquaviva, Basu, Heidorn
10. Shan, Krych, De, Yunqi Zhang
11. Hanling Zhang, Lee, Remazani

7 Deliverables

Please submit the following:

1. Your exploit programs.
2. Gadgets used in each exploit (base address and instructions in gadget)
3. Victim process (cse543-file-victim) address space map from /proc (ASLR off)
4. Trace of print messages (and shell invocation) from your execution of each case
5. Brief writeup on how to execute your programs (in your VM).

8 Grading

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

1. Writeup for how to execute your program and gadget information (20 pts)
2. Attacks (80 pts - 20 pts each)

References

[1] Return-oriented programming attack demo. `https://www.youtube.com/watch?v=a8_fDdWB2-M`.

[2] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur.*, 15(1):2:1–2:34, Mar. 2012.

[3] J. Salwan. Ropgadget. `https://github.com/JonathanSalwan/ROPgadget`.