CSE 543 - Spring 2015 - Assignment 3: Return-oriented Programming

1 Dates

- **Out:** *March 26, 2015*
- **Due:** *April 22, 2015*

2 Introduction

In this assignment, you will produce 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 ROP for different types of vulnerabilities and to find gadgets capable of 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. Once you install the ROPgadget tool (and supporting software Capstone), you can easily compute gadgets. To compute gadgets from the 32-bit libc library:

```
./ROPgadget.py /lib32/libc-2.13.so
```

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

Choosing Gadgets There are many, many gadgets in libc, so a question is which gadgets should you be looking for. As a start, you will need gadgets to reset the stack pointer (for triggering ROP attacks from function pointers and for creating loops and direct jumps). 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"). See `http://www.chemie.fu-berlin.de/chemnet/use/info/gdb/gdb_7.html` 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  /home/jaeger/psu/cse543-s15/p3/p2-bu
08049000-0804a000 rw-p 00001000 08:01 3547850  /home/jaeger/psu/cse543-s15/p3/p2-bu
f763f000-f7640000 rw-p 00000000 00:00 0
f7640000-f779e000 r-xp 00000000 08:01 1707696  /lib32/libc-2.13.so
...
```

The first two lines refer to the executable's (`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, you should disable the ASLR on your system (temporarily). 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 will need sudo access on your evaluation machines to perform the attacks. Please let me know if this is a problem for your team.

4 Problem Statement

Your task is to produce ROP attacks for against the given program (file-victim) for buffer overflows that overwrite the return address and overwrite a function pointer. You will produce exploits from a library's code (e.g., I tested with libc) to perform ROP tasks described below. The vulnerable code has a function called `shell` that you can invoke to complete the exploits (demonstrate that your code works visibly).

5 Exercise Steps

The initial code for the project is available at `http://www.cse.psu.edu/~tjaeger/cse543-s15/p3-rop.tgz`. This code consists of three main files: (1) the victim program `cse543-file-victim.c`; (2) the program that produces buffers to attack return addresses `cse543-buf-attack.c`; and (3) the program that produces buffers to attack function pointers `cse543-fn-attack.c`.

The current code in `cse543-buf-attack.c` produces a buffer that performs what is known as a return-to-libc attack using the *shell* function in the victim program, `cse543-file-victim.c`. These attacks are a type of ROP attack where the gadget stack only consists of the libc function.

In this project, we will explore choosing of different types of gadgets to launch the *shell* function and use the overflow of a function pointer in `cse543-fn-attack.c` to launch such attacks - using the function pointer requires an extra step of pointing the exploit to the target stack.

The project consists of the following steps.

1. Produce exploits for two buffer overflow vulnerabilities in the program `cse543-file-victim.c`: (1) overflow the return address using the buffer overflow vulnerability in the function `buffer_victim` and (2) overflow the function pointer using the buffer overflow in the function `function_victim`.

   The program `cse543-buf-attack.c` demonstrates the buffer overflow of the return address to launch shell directly - a return-to-libc attack, a type of ROP attack. Code in `cse543-fn-attack.c` can be modified similarly to attack the function pointer (or you may write your own code).

2. Produce ROP attacks with the following behaviors. Note that each of these attacks must end in creating a shell (using the function `shell` in cse543-file-victim.c). These attacks are described in the ROP paper [2] in Section 5.3.

   - An unconditional jump. You must create one stack that jumps to a second stack and invokes shell.
   - Gadget that ends in "call" instruction. At least one gadget should end in a call instruction (e.g., that calls the shell program).
   - A loop. This is another unconditional jump, but you must choose gadgets that: (a) establish an ROP stack; (b) execute off of that stack; (c) modify a value at a stack location that has already been used by a gadget (e.g., replace with a call to `shell`); and (d) reset the stack pointer back to a stack location that has already been used by a gadget (e.g., back to the modified location).
   - A conditional jump. This is the most difficult one, so may require some research and discussion although the ROP paper provides some guidance. (This one is optional)

So, there will be 6 attack programs (2 optional). Two for each attack type above, one covering attacks on the return addresses and one covering attacks on function pointers.

5.1 Teams

1. Minkin Ilia, Upreti Nitish, Xu Dongpeng (VM 130.203.47.51 - username: london)
2. Jadidi Amin, Rengasamy Prasanna Venkatesh, Mukhopadhyay Manjari (VM 130.203.47.52 - username: beijing)
3. Wang Shuai, Zientara Peter, Lv Weining (VM 130.203.47.53 - username: moscow)
4. Sharma Aakash, Narayanan Iyswarya, Wang Kaiyu (VM 130.203.47.54 - username: washington)

5. Nima Elyasi, Saghaian Nejad Esfahani Sayed, Qiu Li, Cao Wenqi (VM 130.203.47.55 - username: paris)

6 Deliverables

Please submit the following:

1. Your exploit programs and the necessary makefiles for them.
2. Gadgets used in each exploit (base address and instructions in gadget)
3. Victim process (file-victim) address space map from /proc
4. Trace of print messages (and shell invocation) from your execution of each case
5. Brief writeup on how to execute your programs (on ladon.cse.psu.edu - please make sure your exploit code runs there)
6. Answer to the question: How could we launch an ROP attack if the stack canary and ASLR defenses are enabled? Be as precise as possible.

7 Grading

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

1. Writeup for how to execute your program works (10 pts)
2. Each attack (80 pts)
3. Answer to the question above (10 pts)

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`.