CSE 543 - Fall 2017 - Assignment 2: Return-oriented Programming

1 Dates

- **Out:** *October 12, 2017*
- **Due:** *November 6, 2017*

2 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 [4]) to extract gadgets from the executable.

3 Background

Return-oriented Programming The paper by Roemer *et al.* describes the principles and capabilities of *return-oriented programming* [3]. 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 [4] 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.

For the Linux Lab machines (for which you do not have root access), capstone has already been installed. To run ROPgadget, you must make sure you have a `/.software`, which you can do by running the command `/usr/local/bin/new_soft` and then uncomment the lower line of the following.

```
# Anaconda2.4-Python2.7: Anaconda 2.4 Python 2.7
#Anaconda2.4-Python2.7
```

so it looks like

```
# Anaconda2.4-Python2.7: Anaconda 2.4 Python 2.7
Anaconda2.4-Python2.7
```

You also must add the following to your .cshrc

```
alias python /home/software/python/Anaconda2.4-Python2.7/bin/python
```

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.py --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 `input_passwords` function given the available buffer overflows in that function.

You are given the source code to the program (`input_passwords` is in cse543-pwdmgr.c), which is similar to the Project 1 code in some ways regarding the processing of input, but contains at least one buffer overflow. Hopefully, this is not the way you gathered input. A safe method for processing input is in `main` for processing Lookup requests.

Use the variable `input_domain` 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 `input_passwords`. 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.

```
cse-p204inst01.cse.psu.edu 98\%   gdb cse543-p2
GNU gdb (GDB) Red Hat Enterprise Linux (7.2-92.el6)
Copyright (C) 2010 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 ``x86\_64-redhat-linux-gnu''.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from ***...done.
(gdb)   run test abcd rockyou.txt.6.4.a.mc1 payload l
```

We run the same API as from project 1, although the only input that matters is the "payload" file which contains domain-password pairs (only one is necessary). The other arguments may be any string (i.e., may not correspond to any real file name or password value). You will write programs to produce payloads to exploit the buffer overflow.

Learning about ROP Attacks One video [1] 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.

Another (short) video [2] 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).

Note that program input functions are senstive 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 PLPT 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 [1] has several examples of this to help you.

To choose the PLT functions to use, use `objdump -dl cse543-p2` 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 [1]. 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.

4 Project Platforms

For this project, we will use the Linux Lab machines in Westgate W204 once again. I have tested the exploit is consist across machines cse-p204inst(01-05 and 10 and 20).cse.psu.edu. You must use these machines for developing your exploits.

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

5 Exercise Tasks

The initial code for the project is available at `http://www.cse.psu.edu/~trj1/cse543-f17/p2-rop.tgz`. This code consists of two groups of files: (1) the victim program `cse543-p2` (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.

I have also provided each of you with a personalized binary. You should learn how to accomplish each of the tasks with the common binary (i.e., from the tar file), and then apply the same steps to configure your personal binary. Keep close track of the steps you take on the common binary, so you can repeat and identify steps that do not work. Note that Tasks #1 and #2 only need to be performed on the common binary, whereas Tasks #3 and #4 must use your personal binary. 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 "Hello,World!" 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 "Hello,World" on the stack. Fourth, you need to add the string value "Hello,World" 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 "Hello,World" 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 only print "Hello,World!" from gdb. The output under gdb should look like below (may not be exactly the same, but important that "Hello,World!" appears). The program will segmentation fault from the command line.

   ```
   Input Domain:
   ```

```
Password:
Input Domain: Hello,World!
Program exited with code 0110.
(gdb)
```

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 ZZYZX ``Hello, WORLD\!''
   ```

   Then run the victim under gdb to find the location of this environment variable in the program's memory. The video [1] shows how this is done. You can execute `x/500s $esp` to print the next 500 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 "Hello, WORLD!" from gdb. The output under gdb should look like below (may not be exactly same). The program will segmentation fault from the command line.

   ```
   Input Domain:
   Password:
   Input Domain:
   Password: Hello, WORLD!
   Program exited normally.
   (gdb)
   ```

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" as the command. The string "/bin/ls" 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-p2
   ```

   You will have to compute the actual address of the "/bin/ls" 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. Answer the question below about the results.

4. In this task, you write a program in the file `cse543-string-attack.c` to build a payload to launch a new shell (`/bin/sh`) using the function `system`.

   The main challenge is that the program does not contain the string "/bin/sh", so you must use the payload to construct the string "/bin/sh" first, and then invoke system to run that command.

   The idea of this attack is to find pieces of strings from the victim executable that enable you to construct "/bin/sh". "/bin/ls" is an obvious starting point, but you will need to replace "ls" with "sh". Other strings in the executable have the necessary components.

   An issue is that the strings are in code memory (i.e., read-only memory), so you will not be able to modify them in place. The video [1] recommends writing the string to the writable `.dynamic` section of the executable memory and demonstrates how to do this. You can find information about sections of the executable with objdump.

   ```
   objdump -x cse543-p2 | less
   ```

   To copy string values from the code to writable memory, I highly recommend using strcpy. For programming, strcpy is not safe, but you are producing an exploit, so this is not an issue. Since strcpy only requires the source and destination addresses, it is easier to use. There is also a PLT stub for strcpy.

   You will need to use gadgets in the task to clear (pop) the arguments from one strcpy for the next strcpy or system stub to set the stack pointer at the next stub. The video [1] demonstrates this as well.

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

6 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 4. Precisely describe how that defense would prevent the attack.
5. Specify the gadgets used and their purpose for Task 4.

7 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

8 Grading

The assignment is worth 100 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 and Task #2 on the common binary included in the project tarfile (exploit programs and trace of output). (20 pts)
4. Task #3 and Task #4 on the personal binary provided (exploit programs and trace of output). (45 pts)

References

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

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

[3] 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.

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