System Security
Return-Oriented Programming

Graded Assignment

Distribution: 08.11.2018
Hand in: 22.11.2018 (13:15)

Introduction

This exercise introduces you to *chained return-to-libc* attacks. It builds on your knowledge from the previous exercise. Here, you will build exploits for the binary program `rop`. The goal of this attack is to be able to execute a shell script called `somefile.sh`. Please use the `rop` folder. To setup the `rop` folder, run `setup.sh` (enter the syssec password when prompted).

This is a long exercise. Please read each part carefully and answer all questions as they are all given points.

1 Goal

In this exercise, you will have to chain several libc functions to execute `somefile.sh` that you find in your `rop` folder. When you check the permissions of `somefile.sh`, you will see that it can only be read/written by its owner (root in this case)—so the normal user (syssec) cannot execute it.

However, the user (syssec), has access to a vulnerable setuid program, which is `rop` that he can use to execute `somefile.sh`. His final goal is to execute the equivalent of the following commands:

- chmod 700 ./somefile.sh
- ./somefile.sh
- chmod 600 ./somefile.sh

Note that the **user cannot simply try to get a root shell (as in the previous exercise) and execute somefile.sh** because the creation of all shells is being monitored/logged[1]. So he has to resort to executing `somefile.sh` without explicitly spawning a shell. Specifically, his goal is to chain libc-functions that will help him achieve his goal. You are not allowed to use `system` (or similar functions) to execute anything other than `somefile.sh`[2].

[1]Technically, `system` also spawns a shell. We will ignore this for simplicity, because we want to allow you to use `system`.

[2]In other words, you are not allowed to use, e.g., a wrapper function for all three calls or put them into one call to `system` (as `system("chmod 700 somefile.sh; ./somefile.sh; chmod 600 somefile.sh")`), because the purpose of this exercise is to chain multiple calls.

Structure of the Exercise and Advice

The rest of this exercise is broken down into small steps that will allow you to achieve the above goals. Here is some additional advice for successfully completing this exercise:

- Please do not run your exploits in the folder that is shared between your VM and host.
- Note that this program is slightly different from the older exercise. You are allowed only one command-line input and one runtime input. You have to redo your analysis of stack frames before you exploit the new `rop` executable.
- Only use the input taken at run-time for storing strings not for exploitation.
- Please run your final exploit outside of gdb. Intermediate steps can be run inside gdb.
- As mentioned earlier, you cannot simply spawn a root shell and complete the exercise; you have to chain libc functions.
- You are allowed **at most 2** environment variables for the final exploit (in addition to the commandline and runtime inputs)
- Your exploit must end without a segmentation fault, but does not have to give a specific exit code.

Unix File Permissions (3 points)

In Unix-based file systems, every file has a 9-bit permission. The first three bits are for read, write and execute permissions for the owner. The middle three and last three represent similar permissions for the group and others respectively. Furthermore, there is a "setuid" permission bit that if set allows any user to execute the file with the permissions of its owner. Please answer the following questions:

- Who is the owner of `somefile.sh`?
- Who is allowed to read `somefile.sh`?
- Who is allowed to write `somefile.sh`?
- Who is allowed to execute `somefile.sh`?
- What is the 32-bit hexadecimal representation of the current permissions of `somefile.sh`?
- What is the 32-bit hexadecimal representation for the mode 0700?

Format String Vulnerabilities (1 points)

C library functions like `printf` and `scanf` accept format strings as a first argument and then a set of variable parameters. If the user can supply the first argument, the execution can have undesired consequences. For instance, some format strings are especially dangerous because they can be used to overwrite arbitrary memory locations. The "%n" format string is one such example. Please answer the following questions regarding its use:

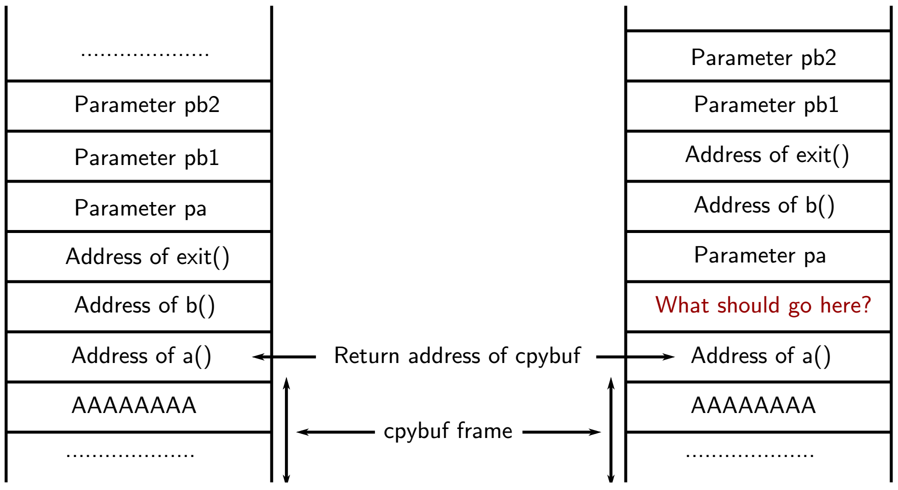

Figure 1: Potential stack frames for chaining functions `a` and `b`.

- What is the value of `i` after the following code executes?

  `int i; printf("%n",&i);`

- What is the value of `i` after the following code executes?

  `int i; printf("%16x%n",i,&i);`

Chaining Arbitrary Functions (4 points)

Assume that cpybuf is a vulnerable function whose buffer can be overflowed. Consider the following functions: `void a(`*pa*`);` and `void b(`*pb*1, *pb*2`);`

- Now, if on overflowing cpybuf, one would first like to execute function `a` with parameter `pa`, then function `b` with parameters `pb1` and `pb2` and finally `exit`, does the stack layout on the left in Figure 1 work? Justify your answer.

- Given the stack on the right in Figure 1, what instructions must the placeholder point to in order to make functions `a` and `b` execute correctly? **Hint: When function `a` returns, the stack pointer points to the placeholder. Now you have to remove the parameter `pa` and the jump to the next location pointed to by the $esp, which would be `address of b()`.**

- Could you find the instructions required in the placeholder anywhere in your program already?

Simple Libc Chaining (5 points)

This is your first task of chaining libc functions. On doing this successfully, you will know how to manipulate the stack to chain arbitrary functions. On examining the source code of `rop`, you will see that it has a global variable called `test`. Your task is to exploit `rop`, overwrite `test` to `0x100` using printf, and print its value using the `print_test` function, which is part of `rop`. In other words, please chain `printf`, `print_test` and `exit` to achieve this. Please answer the following questions regarding this task:

- What is the address of variable `test`?
- What `printf` command will let you overwrite variable `test` appropriately?
- What instructions do you need to "fix" the stack after calling `printf` and before calling `print_test`? **Hint: How many parameters of `printf` do you have to remove before jumping to `print_test`?**
- When you chain `printf`, `print_test` and `exit`, what does the stack layout look like after you overflow the vulnerable buffer in `cpybuf` but before you return from `cpybuf`?
- What is the final command that you used to successfully run this exploit?

Final Task: Creating Longer Libc Chains (10 points)

Finally, you will now design and run the original exploit to run `somefile.sh`. You are allowed to use **only two environment variables** for this task. **You have to accomplish this task both inside and outside of gdb.** When you specify the shell file to execute (either to the program or as an environment variable), please enter "./somefile.sh" (and not just "somefile.sh"). Please answer the following questions regarding this task:

- What libc functions would you chain to achieve the equivalent of the three commands listed under the goals of this exercise? Please provide your answer as a list of function calls with appropriate parameters.
- Do these calls work as an exploit? Justify your answer. **Hint: The `strcpy` function that is used to overflow the buffer stops on encountering a `NULL` byte.**
- What would you do to overcome it? Can you think of some functions to generate the required values? Please list the required function calls with appropriate parameters. **Hint: you have done this already in the exercise if you got this far.**
- Given that `rop` takes one command-line input and one runtime input, where could you put any additional inputs that you need? Please specify the exact unix commands that you used to do this.
- Please sketch the stack layout that you used with annotations if necessary.
- What is the final exploit string that you used to accomplish this task? (The final exploit should not use gdb.)