Operating Systems Security – Assignment 2

2017/2018
Due Date: 30 Nov 2017 (23:59 CET)

1 Compile and load your own Linux kernel module

Login to your (Kali) Linux system as a **root** user and compile the program `cr4.c`:

```
#include <stdio.h>

void main() {
   unsigned long long result;
   /*unsigned long result; (for 32-bit OS)*/
   __asm__("movq %%cr4, %%rax\n" : "=a"(result));
   /*__asm__("mov %%cr4, %%eax\n" : "=a"(result)); (for 32-bit OS)*/
   printf("Value of CR4 = %llx\n", result);
}
```

with the command line:
`# gcc -o cr4 cr4.c`

Notice that executing will result in an exception:
`# ./cr4`
`Segmentation fault`

Using a debugger, we can quickly pinpoint what the problem is. Start debugger in assembly mode
`# gdb -ex "layout asm" ./cr4`
and execute it using the following `GDB` instruction
`# run`

Objectives

a) Figure out where the register `CR4` is used for and report back why you think it should not be accessible in user mode[1].

b) Figure out which exact assembly instruction of `cr4.c` triggers the segmentation fault and briefly write down what it tries to do.

c) Follow the *"How to Write Your Own Linux Kernel Module with a Simple Example"* guide hosted at this website[2] and try to reproduce their results. You should be able to see your kernel module output with the following command:
$ **`dmesg | tail -10`**

d) If your kernel module is working correctly, try to adjust the kernel module to read out the exact same `CR4` register. Hand in the source-code of your kernel module together with a Makefile to build it and report back which value the `CR4` in your (Kali) Linux system has.

[1] http://en.wikipedia.org/wiki/Control_register
[2] http://www.thegeekstuff.com/2013/07/write-linux-kernel-module/

2 Return to libc

The standard mechanism to circumvent a non-executable stack in a buffer-overflow attack is to use return-oriented programming. This exercise is a classical return-to-libc attack on the AMD64 architecture.

Prerequisites

We will (at first) make our life easy and attack a textbook vulnerable program, which additionally prints the address of a buffer:

```
#include <stdio.h>

int main(void)
{
  char name[256];
  printf("%p\n",&name);
  puts("What's your name?");
  gets(name);
  printf("Hello, %s!\n", name);
  return 0;
}
```

Let us assume that this program is running with suid-root; the target of the attacker is to use a buffer overflow of the `name` buffer to obtain a root shell.
The idea of the attack is the following: make sure that the code eventually returns into the `system` function of libc with a pointer to the string "`/bin/sh`" in register `rdi`. This assumes that the attack is running on a 64-bit Linux system (AMD64 architecture); on this architecture, the first argument of a function is passed through register `rdi`. The attack needs the following building blocks:

– Put the string "`/bin/sh`" somewhere into the address space of the program, e.g., into the buffer `name`;
– find a gadget that consists of the instruction `pop %rdi` followed by `retq`;
– overwrite the return address of the function with the address of this gadget;
– write behind this gadget the address of the string "`/bin/sh`" (this is what is going to be popped into `rdi`);
– write behind the address of the string the address of `system` in libc. This is what is finally going to be called, giving you the root shell.
– (optionally) write behind this address the address of `exit` in libc. This will avoid the segmentation fault.

An excellent walkthrough of this attack is given by Ben Lynn on http://crypto.stanford.edu/~blynn/rop/.
Remark 1: Some parts of the assignment may depend on various aspects of your Linux system (in particular, the version of libc). If you have trouble with some parts, then try on `lilo.science.ru.nl` (where it has been tested). Obviously on this machine you cannot run the program suid-root, but you can still get a (non-root) shell and confirm that the attack works.
Remark 2: Note that in our exercise the size of the buffer changed; take this into account when mounting the attack.
Remark 3: It is important to compile the program with gcc flag `-fno-stack-protector`.
Remark 4: It is important to disable ASLR (either by using `setarch `arch` -R` as in Lynn's tutorial for each call or by running `echo 0 > /proc/sys/kernel/randomize_va_space` once as root).
Remark 5: Tip: try to make your attack execute `/bin/ls` first. If that works, change it to `/bin/sh`.

2.1 Objectives

a) Run the attack and obtain a root shell (you might want to try first without suid-root to not allow too much disaster if things go wrong). Now automate this attack in a bash script. The bash script should be robust, i.e., it should handle the case that offsets in libc are different. You can test this by running the script on a different machine, e.g., on `lilo.science.ru.nl`. Submit this script.

b) The attack does not work against the "original" version of the program in Lynn's tutorial with a buffer size of 64. Use gdb to find out why not In particular, answer the following questions:
   - Does the attacked program jump (return) to the `pop %rdi, retq` gadget? If not, why not?
   - Does the attacked program put the right address into `rdi`? If not, why not?
   - Does the attacked program jump (return) to `system`? If not, why not?
   - Does the attacked program issue the correct syscall? If so, which one? If not, why not?
   - Summarize and explain why the attack does not work.

   Note: It is of course perfectly fine to compile the program with the `-g` flag).

   Note: Addresses (for example, of `name`) are probably slightly different when running the program in gdb.

c) Can you think about a way to make the attack work with a buffer of size 64?

   Hint: Where else can you find the string `/bin/sh` or similar?

d) **Bonus task**: Make the attack work against a buffer of size 64 and against a buffer of size 4.