University of Potsdam

Institute of Computational Science

Homework 4

Software Security

Martin Stähr
M.Sc. Computational Science
754245

Max Schrötter
M.Sc. Computational Science
761220

June 14, 2018

1 Exercise 1

1.1 (a)

A static canary is not save, because you can override the return address of the function without notice by the canary, because you override the canary also with initial value while overriding the return address.

1.2 (b)

If executable of the original source code is executed via the following command. The execution path will be diverted to the Hack function. This will only work with executables made by our provided Makefile. It is although import that ASLR is disabled.

```
./hw.out 11111111111111111111111111111111~~~~~~~~@GUUUU
```

The argument is constructed as the following:

- There are 32 `1` because this fills everything from the dest array until the saved rbp address with 1s including the canary. The saved rbp is set to some address writeable (8). The return Address is overwritten with `@GUUUU` which is the first instruction of the hacked method in our text alignment!

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  #include<string.h>
4  void vuln_copy (char *attack_source)
5  {
6      char dest[20];
7      char canary='1';
8  //insecure copy
9      strcpy(dest, attack_source);
10     if(canary!='1')
11         exit(0);
12 }
13 int main (int argc, char **argv)
14 {
15 // call the vulnerable function
16     vuln_copy(argv[1]);
17 }
18 void hack_me ()
19 {
20     printf("Hacked!\n");
21 }
```

2 Exercise 2

The Control flow integrity approach uses static program analysis to create an control flow graph and add protection the function can only return to addresses shown in the CFG. This can be done by labels or shadow stacks. In our case we used labels to ensure correct control flow.

2.1 (a)

The check should be done before the assembly `ret` and `leave` are executed inside the vulnerable function. This check will test if there is a label at the position of the return address.

2.2 (b)

We have modified the original source code with the following inline assembly.

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  #include<string.h>
4  void vuln_copy (char *attack_source)
5  {
6      unsigned long long flag;
7      char dest[20];
8      char canary='1';
9  //insecure copy
10     strcpy(dest, attack_source);
11     if(canary!='1')
12         exit(0);
13     asm volatile ( "movq %%rbp, %%rax \n" //Get the stack base pointer
           into general purpose register
14             "addq $8, %%rax \n"  //increase registe to return address
15             "movq (%%rax), %%rax \n" // dereference return address
16             "cmpq $0x12345678, (%%rax) \n" //compare label at the return
                   address with static saved label to ensure Programm flow
17             "je .LMAX\n " // Jump in success to Label MAX
18             "movl $1, %%edi \n" // Set exit value to one in case of
                   manipulation
19             "movq $60, %%rax \n" //Set the syscall number for exit
20             "syscall\n" // invoce the kernel trap
21             ".LMAX: \n" // Label MAX
22             "addq $8, 8(%%rbp)" //Add 8 to the return address to bypass
                   label and execute next instruction in main.
23             :
24             :
25             :"rax"); //cobbler register (Register used in the assembly
                   code)
26 }
27 int main (int argc, char **argv)
28 {
29 // call the vulnerable function
30     vuln_copy(argv[1]);
31     asm volatile ( "; .quad 0x12345678 \n"); // The label used for the
           check
32
33 }
34 void hack_me ()
35 {
36     printf("Hacked!\n");
37 }
```

Please use the Makefile provided in the zip file to compile the code and run it.