COMP2700 Lab 6 – Software Security

In this lab, we will look at several patterns of vulnerability in software and how they can be exploited. To be more concrete, we will examine some toy C programs that contain these patterns of vulnerability, but the patterns of vulnerabilities themselves do occur in real (more complicated) software.

Exercises marked with (*) are extension exercises and are not covered in the lab.

Lab setup

For this lab, we will again use the lab VM from Lab 1. For some of the exercises that require probing contents at certain memory addresses, the default settings in the lab VM may produce different memory locations of variables each time a program is run. This is because of a defense mechanism in Linux called **Address Space Layout Randomisation (ASLR)**, that randomises the memory location of executables. You would need to disable ASLR first as follows:

- Login as user admin2700 and run

  sudo sysctl -w kernel.randomize_va_space=0

Note that this disables ASLR temporarily until the next reboot. So if you restart your system, you will need to repeat the above steps again to disable ASLR.

The C programs for this lab is available in the file lab6.tar.gz, which you can download and extract using the following commands:

```
wget http://users.cecs.anu.edu.au/~tiu/comp2700/lab6.tar.gz
extract-lab lab6.tar.gz
```

If the above link does not work, use the following alternative link:

```
wget https://cloudstor.aarnet.edu.au/plus/s/IgSC2mVQOS5BWr0/download -O lab6.tar.gz
extract-lab lab6.tar.gz
```

This will create a directory called lab6 in admin2700's home directory.

C code compilation

Each exercise below comes with C code that you'd need to compile first before testing it. We will use the gcc compiler already installed in the lab VM. For this lab, we will use the following command to compile the C programs: assuming the input C program is called 'myprogram.c'

```
gcc myprogram.c -o myprogram -m32 -fno-stack-protector
```

This command compiles myprogram.c and produces an output file myprogram, which is the executable binary for myprogram.c. The option -m32 tells the compiler to produce a binary in the 32-bit architecture (the current lab VM assumes a 64-bit architecture). This is so that it will be easier to reproduce some attacks, especially the ones related to format string vulnerability. The option -fno-stack-protector disables the stack guard, so we can reliably launch a buffer overflow attack.

Buffer overruns

Exercise 1. Consider the following program, which is a variation of a program discussed in a lecture on stack-based buffer overflow.

File: ex1.c

```c
void Win()
{
    printf("Well done!\n");
}
int main(int argc, char *argv[])
{
    int a = 0;
    char buffer[16];
    if(argc < 2){
        printf("Usage: %s <string>\n", argv[0]);
        exit(1);
    }
    printf("Address of a: %p\n", &a);
    printf("Address of buffer: %p\n", buffer);
    strcpy(buffer, argv[1]);
    printf("Value of a: %08x\n", a);
    if (a == 0x41424344) Win();
    else printf("Try again\n");
    return 0;
}
```

As discussed in the lecture, an attacker can overflow the value of ‘buffer’ variable to overwrite the value of the variable ‘a’. Try to overwrite the value of a in such a way that the Win() function will be called. *Hint: pay attention to the endianness of how the integer a is represented in the stack and make sure you overwrite its underlying bytes in the correct order.*

Exercise 2. Consider the following code:

File: ex2.c

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    char a = 'a';
    char *buffer;

    if(argc < 2){
        printf("Usage: %s <string>\n", argv[0]);
        exit(1);
    }

    buffer=(char *) malloc(16);

    if(buffer == NULL) {
        printf("Memory allocation failed\n");
        exit(1);
    }

    printf("Address of a: %p\n", &a);
    printf("Address of buffer: %p\n", buffer);

    strcpy(buffer, argv[1]);
    printf("Value of a: %c\n", a);

    free(buffer);

    return 0;
}
```

This is very similar to the C code in Exercise 1. However, you will find that the buffer overrun attack that works for Exercise 1 does not work for this exercise. Why is that?

Exercise 3. Consider the following code:

File: ex3.c

```
#include <stdio.h>
#include <string.h>

void print_message(char * msg)
{
  char secret[8] = "SECRET";
  char buffer[15];
  strncpy(buffer, msg, 15);
  printf("Message: %s\n", buffer);
}

int main(int argc, char* argv[])
{
  if(argc < 2) {
     printf("Usage: %s <message>\n", argv[0]);
     return 0;
  }
  print_message(argv[1]);
}
```

The function print_message first copies the parameter msg to a local buffer, and then prints the content of the buffer. Notice that the function uses a safer function (strncpy) to copy a string to the buffer. This will prevent a buffer overflow from overwriting the return address stored in the stack. However, it is still possible to exploit this function to display the value of the variable 'secret'. Show how this attack works, and explain why it works.

Exercise 4. Consider the following program:

ex4.c

```
#include <stdio.h>

int main(int argc, char * argv[])
{
    int  secret_int = 0x41424344;

    if(argc < 2) {
        printf("Usage: %s <message>\n", argv[0]);
        return 0;
    }

    printf(argv[1]);
    return 0;
}
```

The printf command in this program prints a format string controlled by the user (via argv[1]). What input do you provide to the program so that it prints the value of secret_int? How many format specifiers you need to print the variable secret_int? You can use the format specifier %x to print an integer in hexadecimal to help you identifying the correct value on the stack of the function main that corresponds to the variable of secret_int.

Exercise 5. Consider the following program:

File: ex5.c

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>

int main()
{
    int n = 0;
    int guess;
    int helper=0x12345678; // use this value to find the location of guess
    char str[64]="What is your guess? ";
    char answer[16];

    printf("Address of n: %p\n", &n);
    printf("Address of guess: %p\n", &guess);
    printf("Address of helper: %p\n", &helper);
    srand(time(0));
    guess = rand();
    do {
        printf(str);
        fgets(answer,16,stdin);
        printf("Your answer: ");
        printf(answer);
        n = atoi(answer);
        if(n == guess) {
            printf(" is correct!\n");
            break;
        }
        printf(" is wrong!\n");
        printf("Guess again (y/n)? ");
        fgets(answer,3,stdin);
    }
    while(answer[0] == 'y');
    return 0;
}
```

This program contains a format string vulnerability, in the sense that the attacker can influence a format string used in a `printf' command. Identify that vulnerable printf, and show how the attacker can exploit this vulnerability to print the content of the stack frame to obtain the randomly generated value in the variable 'guess'.

Hint: Since the buffer ('answer') you can exploit in this case has only 16 bytes, you may not be able to insert enough format specifiers, e.g., %x, to print the entire stack, so you may miss the target variable. Fortunately, printf allows one to address an argument directly by specifying its relative position with respect to the format string, by inserting a 'n$' (where n is the relative position) in format specifier. So "%n$x" means 'print the n-th argument of printf as an integer in hex format'. For example:

```
printf("%3$x", 1, 2, 3)
```

will print '3' (the third argument after the format string). Use this format specifier to print the stack address one by one, and when you find a value that matches the value of the 'helper' variable (0x12345678), you know that the variable 'guess' must be the next 'argument' after 'helper'.   You would need to write a shell script to automate this process. See the Appendix below for some hints related to shellscripting that you may find useful.

*Note that this trick works with other types of format specifiers (not just integers), so "%10s", for example, will treat the 10-th argument of printf (relative to the format string) as a pointer to string (char *) and print the content pointed to by that argument.*

Extension Exercises

The following exercises are extension exercises and will not be covered in the lab.

Exercise 6 (*). In the previous exercise, we have seen how to use a printf format string vulnerability to probe elements of the stack frame, and use an 'anchor (the value 0x12345678 in the helper variable) to determine the location of a target variable in the stack. In this exercise, we will look how we can actually use the format string itself as an anchor, and if the memory address of the target is known, we can further exploit it to dereference that address (using the format specifier %s -- i.e., to treat that address as a pointer to char *). Let us look at how this two-stage exploit can be used to print the secret string in the following code:

ex6.c

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main()
{
    char fmtstr[64];
    char secret[8] = "findme";
    char junk[100] = "some random stuff to fill the stack";

    printf("Address of fmtstr: %p\n", fmtstr);
    printf("Address of secret: %p\n", secret);

    printf("Input a format string: ");
    fgets(fmtstr, 64, stdin);
    printf(fmtstr);

    return 0;
}
```

Note that here we hard code the secret string ("findme") so we can actually just go through the stack element one at a time until we find that pattern in the stack. However, if the value of the secret is unknown, we may not know when we have actually found the secret. Instead, we are going to attack this program in two stages:

Stage 1. Find out how far up in the stack is the format string itself from the stack frame of 'printf(fmtstr)'. That is, if we treat each memory address (in a 4-byte increment) as an integer argument for printf(fmstr), how many format specifiers we would have to put in fmtstr in order to print the address of the variable 'fmtstr' itself?

Hint: since we control the value of fmtstr, we could inject a distinguished pattern in fmtstr, and use that pattern as an 'anchor' to determine when you have reached the address of fmtstr. For example, if fmtstr starts with "AAAA" then in the stack this would be encoded as the hex number 0x41414141 (as 0x41 is the ASCII code of 'A'). So if we print the element of the stack one by one, and we eventually encounter that pattern, we know we have found the location of fmstr in the stack.

Stage 2. Once you have found out how many format specifiers needed to reach fmtstr, you can inject the target address into fmtstr (instead of the pattern "AAAA"), and dereference that address (using %s) to print the target string.

Hint: you will need to inject the target address in the first four bytes of fmstr. For example, if the target address is 0xffff9c1b, then the first 4 bytes of fmstr should contain these four bytes: 0x1b, 0x 9c, 0xff, 0xff (note the reverse order). You may want to look at how to generate this array of bytes using 'echo', to inject a target address into the format string fmtstr. See the Appendix for an example.

Exercise 7 (*). Consider the following program:

ex7.c

```
#include <stdio.h>
#include <string.h>

int n = 0;

void Win1()
{
    printf("Well done! Can you get to Win2()? \n");
}

void Win2()
{
    printf("You did it!\n");
}

int main(int argc, char * argv[])
{
    char buf[64];

    printf("Address of n = %p\n", &n);
    printf("Enter a string: ");
    fgets(buf, 64, stdin);
    printf(buf);
    printf("Value of n = %08x\n", n);
    if(n == 100) Win1();
    if(n == 3) Win2();
    return 0;
}
```

Use the format specifier %n to overwrite the memory location of variable n, to execute Win1() and Win2() (in two separate runs). The format specifier %n is used to write the number of characters printed so far to a variable. In a normal use, for example, the following code

Int x;
printf(“hello%n”, &x);

will cause the value 5 (the length of the string `hello`) to be stored in the variable x.

Appendix. Some shellscripting tips

The following are some useful shell scripting commands/features that will help you automate some tasks that require you to manipulate input to programs in the above exercises.

Storing the result of a command

One useful feature of shell script is the ability to store the output of a program (the one printed to the standard output/display) in a variable. This is done using the following syntax:

```
var=$(command)
```

where var is the name of the variable and command is the command to be executed. For example,

```
x=$(echo "hello")
```

stores the string "hello" to the variable x.

"Printing" bytes with echo

In shellscripting, sometimes you may find that you need to produce a stream of bytes as an input to a program. If a byte you want to input falls within the category of 'printable character' in ASCII encoding (e.g., upper/lower case letters, numbers or punctuation symbols), you can simply type the character directly. If however, the byte value does not correspond to a printable character, one way to input it is to generate that byte using the 'echo' command (with option -e). For example,

```
echo -e "\x1b"
```

outputs a byte 1b (in HEX), that corresponds to the ASCII code for the Esc (escape) key press.

This can be useful when you are testing a program that receives input from the standard input (keyboard), and you want to feed an input stream of bytes to the program programmatically.

Testing for substrings in a string

To test that a string is a substring of another string, we can use the 'wildcard' symbol '*' in the string comparison. For example:

```
if [[ $x == *"test"* ]]
then
    echo "test is a substring of $x"
fi
```

checks whether the variable x contains a substring test.