

# Format String Vulnerability Lab

Copyright © 2006 - 2009 Wenliang Du, Syracuse University.

The development of this document is funded by the National Science Foundation's Course, Curriculum, and Laboratory Improvement (CCLI) program under Award No. 0618680 and 0231122. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation. A copy of the license can be found at <http://www.gnu.org/licenses/fdl.html>.

## 1 Lab Overview

The learning objective of this lab is for students to gain the first-hand experience on format-string vulnerability by putting what they have learned about the vulnerability from class into actions. The format-string vulnerability is caused by code like `printf(user_input)`, where the contents of variable of `user_input` is provided by users. When this program is running with privileges (e.g., `Set-UID` program), this `printf` statement becomes dangerous, because it can lead to one of the following consequences: (1) crash the program, (2) read from an arbitrary memory place, and (3) modify the values of in an arbitrary memory place. The last consequence is very dangerous because it can allow users to modify internal variables of a privileged program, and thus change the behavior of the program.

In this lab, students will be given a program with a format-string vulnerability; their task is to develop a scheme to exploit the vulnerability. In addition to the attacks, students will be guided to walk through a protection scheme that can be used to defeat this type of attacks. Students need to evaluate whether the scheme work or not and explain why.

It should be noted that the outcome of this lab is operating system dependent. Our description and discussion are based on Ubuntu Linux. It should also work in the most recent version of Ubuntu. However, if you use different operating systems, different problems and issues might come up.

## 2 Lab Tasks

### 2.1 Task 1: Exploit the vulnerability

In the following program, you will be asked to provide an input, which will be saved in a buffer called `user_input`. The program then prints out the buffer using `printf`. The program is a `Set-UID` program (the owner is `root`), i.e., it runs with the root privilege. Unfortunately, there is a format-string vulnerability in the way how the `printf` is called on the user inputs. We want to exploit this vulnerability and see how much damage we can achieve.

The program has two secret values stored in its memory, and you are interested in these secret values. However, the secret values are unknown to you, nor can you find them from reading the binary code (for the sake of simplicity, we hardcode the secrets using constants `0x44` and `0x55`). Although you do not know the secret values, in practice, it is not so difficult to find out the memory address (the range or the exact value) of them (they are in consecutive addresses), because for many operating systems, the addresses are exactly the same anytime you run the program. In this lab, we just assume that you have already known the exact addresses. To achieve this, the program “intentionally” prints out the addresses for you. With such knowledge, your goal is to achieve the followings (not necessarily at the same time):

- Crash the program .

- Print out the secret[1] value.
- Modify the secret[1] value.
- Modify the secret[1] value to a pre-determined value.

Note that the binary code of the program (Set-UID) is only readable/executable by you, and there is no way you can modify the code. Namely, you need to achieve the above objectives without modifying the vulnerable code. However, you do have a copy of the source code, which can help you design your attacks.

```
/* vul_prog.c */

#define SECRET1 0x44
#define SECRET2 0x55

int main(int argc, char *argv[])
{
    char user_input[100];
    int *secret;
    int int_input;
    int a, b, c, d; /* other variables, not used here.*/

    /* The secret value is stored on the heap */
    secret = (int *) malloc(2*sizeof(int));

    /* getting the secret */
    secret[0] = SECRET1; secret[1] = SECRET2;

    printf("The variable secret's address is 0x%8x (on stack)\n", &secret);
    printf("The variable secret's value is 0x%8x (on heap)\n", secret);
    printf("secret[0]'s address is 0x%8x (on heap)\n", &secret[0]);
    printf("secret[1]'s address is 0x%8x (on heap)\n", &secret[1]);

    printf("Please enter a decimal integer\n");
    scanf("%d", &int_input); /* getting an input from user */
    printf("Please enter a string\n");
    scanf("%s", user_input); /* getting a string from user */

    /* Vulnerable place */
    printf(user_input);
    printf("\n");

    /* Verify whether your attack is successful */
    printf("The original secrets: 0x%x -- 0x%x\n", SECRET1, SECRET2);
    printf("The new secrets:      0x%x -- 0x%x\n", secret[0], secret[1]);
    return 0;
}
```

**Hints:** From the printout, you will find out that `secret[0]` and `secret[1]` are located on the heap, i.e., the actual secrets are stored on the heap. We also know that the address of the first secret (i.e., the value of the variable `secret`) can be found on the stack, because the variable `secret` is allocated on the stack. In other words, if you want to overwrite `secret[0]`, its address is already on the stack; your format string can take advantage of this information. However, although `secret[1]` is just right after `secret[0]`, its address is not available on the stack. This poses a major challenge for your format-string exploit, which needs to have the exact address right on the stack in order to read or write to that address.

## 2.2 Task 2: Memory randomization

If the first `scanf` statement (`scanf('%d', int_input)`) does not exist, i.e., the program does not ask you to enter an integer, the attack in Task 1 become more difficult for those operating systems that have implemented address randomization. Pay attention to the address of `secret[0]` (or `secret[1]`). When you run the program once again, will you get the same address?

Address randomization is introduced to make a number of attacks difficult, such as buffer overflow, format string, etc. To appreciate the idea of address randomization, we will turn off the address randomization in this task, and see whether the format string attack on the previous vulnerable program (without the first `scanf` statement) is still difficult. You can use the following command to turn off the address randomization (note that you need to run it as root):

```
sysctl -w kernel.randomize_va_space=0
```

After turning off the address randomization, your task is to repeat the same task described in Task 1, but you have to remove the first `scanf` statement (`scanf('%d', int_input)`) from the vulnerable program.

**How to let `scanf` accept an arbitrary number?** Usually, `scanf` is going to pause for you to type inputs. Sometimes, you want the program to take a number `0x05` (not the character '5'). Unfortunately, when you type '5' at the input, `scanf` actually takes in the ASCII value of '5', which is `0x35`, rather than `0x05`. The challenge is that in ASCII, `0x05` is not a typable character, so there is no way we can type in this value. One way to solve this problem is to use a file. We can easily write a C program that stores `0x05` (again, not '5') to a file (let us call it `mystring`), then we can run the vulnerable program (let us call it `a.out`) with its input being redirected to `mystring`; namely, we run `a.out < mystring`. This way, `scanf` will take its input from the file `mystring`, instead of from the keyboard.

You need to pay attention to some special numbers, such as `0x0A` (newline), `0x0C` (form feed), `0x0D` (return), and `0x20` (space). `scanf` considers them as separator, and will stop reading anything after these special characters if we have only one “%s” in `scanf`. If one of these special numbers are in the address, you have to find ways to get around this. To simplify your task, if you are unlucky and the secret's address happen to have those special numbers in it, we allow you to add another `malloc` statement before you allocate memory for `secret[2]`. This extra `malloc` can cause the address of secret values to change. If you give the `malloc` an appropriate value, you can create a “lucky” situation, where the addresses of secret do not contain those special numbers.

The following program writes a format string into a file called `mystring`. The first four bytes consist of an arbitrary number that you want to put in this format string, followed by the rest of format string that you typed in from your keyboard.

```
/* write_string.c */
```

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main()
{
    char buf[1000];
    int fp, size;
    unsigned int *address;

    /* Putting any number you like at the beginning of the format string */
    address = (unsigned int *) buf;
    *address = 0x22080;

    /* Getting the rest of the format string */
    scanf("%s", buf+4);
    size = strlen(buf+4) + 4;
    printf("The string length is %d\n", size);

    /* Writing buf to "mystring" */
    fp = open("mystring", O_RDWR | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
    if (fp != -1) {
        write(fp, buf, size);
        close(fp);
    } else {
        printf("Open failed!\n");
    }
}
```

### 3 Submission

You need to submit a detailed lab report to describe what you have done and what you have observed; you also need to provide explanation to the observations that are interesting or surprising.