

Format Strings

Exploiting for Fun and Profit*



"I didn't even give you my coat!"

Control and exploitation through format strings.

By Steve Hanna
SIGMil Research Labs 2006

<http://www.sigmil.org/>

*it is probably not fun nor profitable

What is a format string?

- Common in C/C++ programs
- Used by these functions:
- `fprintf`, `printf`, `sprintf`, `snprintf`, `vfprintf`, `vprintf`, `vsprintf`, `vsnprintf`
- These functions accept a string and an arbitrarily number of arguments.
- Often the form is as follows:
- `printf(“%s\n”,someString);`
- The “format” string specifies how you want the resulting data to be “format”ed. Do you see what I did there?

Format Strings Cont.

- Typically the format string consists of one or more of the following:
 - %d decimal (int)
 - %u unsigned decimal (unsigned int)
 - %x hexadecimal (unsigned int)
 - %s string ((const) (unsigned) char *)
 - %n number of bytes written so far, (* int)
- %n is the most interesting and we will see why later. **REMEMBER KEEP THINKING ABOUT %n.**

Format String Errors

```
//assume hai2u is user input  
printf(hai2u);
```

What happens when hai2u is “hai”?

Okay, what happens when hai2u is
“%s%s%s%s%s”?

Any idea what will most likely happen?

What will happen?

- Well obviously the first one will just print the message.
- The second one is a bit more interesting, let's examine exactly what goes on under the hood of a format string function...

Demystifying

- Typically the function would be declared:
 - `int printf(const char* fmt, ...)`
- The `...` tells C that we can add an arbitrary number of arguments to the end of this function.
- But why does that work?
 - Recall that the stack grows from high memory addresses towards low memory addresses and in `__cdecl` (the C calling convention) arguments are pushed into the stack from right to left.

...Okay?

- So I need to elaborate a bit more... so how about a really excellent ASCII drawing?

```
char ex[] = "51gm1ll 15 ";
int num = 31337;
short iLikeHex[] = {0x6B,0x00};
printf("%s %i %s?\n",ex,num,iLikeHex);
```

So what does the stack look like after calling this function?

```
STACK!
0xFFFFFFFF
RandomGarbage
iLikeHex pointer
num
ex pointer
FormatString
return addr
Saved Stack Frame Pointer
PrintFunction Code
More Garbage
0x00000000
```

So what does the printf function code do?

Conceptually it:

Loops over the Format String and determines what kind of data it is going to deal with.

When it finds a %, it examines the previous item on the stack and formats it for output.

This process will continue until the entire format string has been iterated over.

So, this is dangerous because...?

What if the code had been:

```
printf("%i %i %i %i %i %i %i %i\n");
```

Uh oh! Where are the arguments?

Well, C has no way of knowing that you didn't supply arguments. So recall this stack...

0xFFFFFFFF
RandomGarbage
FormatString
return addr
Saved Stack Frame Pointer
PrintFunction Code
More Garbage
0x00000000

What will happen?

Now it will still try to pull arguments from the stack, but we haven't supplied any!

The result: we will output arbitrary data from the stack.

Woah, what?

- Why would ANYONE make their format string “%i %i %i %i %i %i %i %i”?
- Well, they wouldn't, but do you recall this old friend:
 - `printf(some_string);`
- So, this shows that if we can supply user input to the `printf` function, we can do all kinds of crazy things!
- Woah woah woah Steve, you're talking crazy now. Just how crazy is crazy? Well...

How crazy is crazy?

- Crash the program
 - With a very high probability we can force a crash if we supply “%s%s%s%s%s%s%s%s”. This can be used as a denial of service attack, as well as forcing a core dump. The core dump could have useful information.
- Examine arbitrary stack data.
 - By supplying the proper format string (ie: “%08X...”, we can force the program to output stack contents. This leads to an information leak and can provide information to construct other attacks.
- This is not NEARLY as crazy as advertised
 - Okay, okay, we are almost done with the boring part. We all know why you came to this talk. You want to know how to remotely execute code, you want root and moon money, or at least the same privilege as the vulnerable process.

Welcome to crazy town, population you.

- We all know the basic stack overflow exploit. Overwrite a return address on the stack, when ret is called, we jump into our malicious code. Wee yay, wohoo, okay it's been done about a billion times.
- What happens when there isn't a buffer to overflow, yet a foolish programmer has left behind a `printf(really_cool_secure_string);`
- Well, we can probably do all sorts of crazy things. Do me a favor, when you're at the moon, can you pick up any of that green moon money for me?
- Let's examine, how we can use this to our advantage.

Finally, profit...almost

- Remember %n?

- This format string write the number of characters written thus far by the printf family function, to an address specified by an argument passed to printf.

- Example:

- `printf("how many?%n",&some_int);`
- `some_int` should contain 9.

- How can this help us?

- First a little pop quiz. We need some basic knowledge...
 - x86 Big-Endian or Little-Endian? What's that mean?
 - What's the difference between a number and an address?
 - What happens if we try to shove an integer into a short integer?
 - Which way does the stack grow?

Answers

- x86 Big-Endian or Little-Endian? What's that mean?
 - Little-Endian. It means the most significant byte (MSB) is stored at the memory location with the lowest address.
Ex: 0xAABBCCDD
Stored in memory in this order...
0xFFFFFFFF
DD
CC
BB
AA
0x00000000
 - What's the difference between a number and an address? Nothing! representation!
 - What happens if we try to shove an integer into a short integer? It gets truncated!
 - Which way does the stack grow? Towards lower memory addresses!

Profit

- We will only cover writing an arbitrary address to the stack. From there, we can use the other techniques (shellcode, etc) to do whatever we want.
- So let's use repeated calls to %n to write the address we want (0xAABBCCDD)
- The format string would look something like this
- xDCx96x04x08 JUNK xDDx96x04x08 %x %x%146x%n%017x%n
- The initial %x's are to eat up miscellaneous bytes on the stack.
- The JUNK is is to supply an argument to the %NUMBERx (for incrementing the variable).

```
AA 00 00 00 | 0x080496dc
BB 00 00 00 | 0x080496dd
CC 00 00 00 | 0x080496de
DD 00 00 00 | 0x080496df
-----|
AA BB CC DD |
Result starting at 0x080496dc
```

After re-writing this in the correct byte order:
0xDDCCBBAA

Moon Money

- Well, this isn't all that useful because how often will we write an address that increments?
- We can use the same trick to write any arbitrary address we want. The trick is to increment the %n using %x, and increment until we get the number we want. This introduces an additional caveat that a byte will be written. However, because we will overwrite that location with the next byte in our address it is a moot point.

Practicality?

- In general, these exploits are very rare, as static code analyzers will catch this programming blunder.
- The first in the wild case of a format string vulnerability was not discovered until 1999.
- I think this is a pretty cool technique, but in general it is not practical.