Format String Attacks

Sources:

- *Hacking: the Art of Exploitation*, 2nd edition, by Jon Erickson
- *Format String Attack*, Open Web Application Security Project (OWASP), www.owasp.org/index.php/Format_string_attack
- *FormatGuard: Automatic Protection from printf Format String Vulnerabilities* by C. Cowan et al, 2001 USENIX Security Symposium
- *Analysis of Format String Bugs* by A. Thuemmel, http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.187.9134
- *Buffer overflow and format string overflow vulnerabilities* by K.S. Lhee and S.J. Chapin, Software Practice & Experience 33, 2003

Format string vulnerabilities (FSVs) permit an attacker to execute code, read the call stack, or cause a segmentation fault in the running application.

FSVs typically involve the **`printf`** family of functions.

Their root cause is that the C-language "varargs" mechanism for "variadic" functions is type-unsafe.

Varargs permits a function to accept an indefinite (variable) number of arguments.

One fixed argument must indicate the total number of arguments.

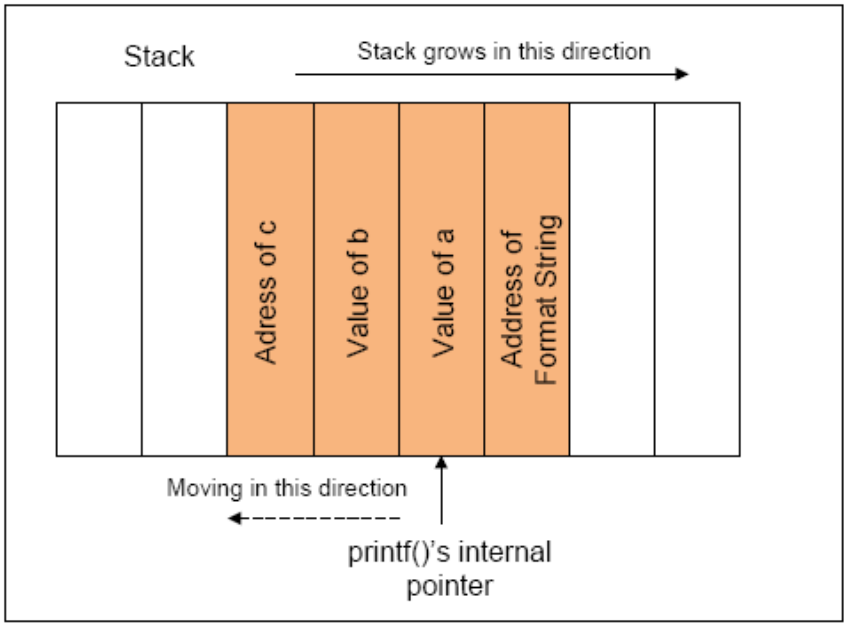
Varargs provides primitives for "popping" arguments off the call stack.

The function receiving the arguments is wholly responsible for popping the correct number, type, and sequence of arguments.

The **`printf`** family of functions uses varargs.

Parameter	Meaning	Passed as
%d	decimal (int)	value
%u	unsigned decimal (unsigned int)	value
%x	hexadecimal (unsigned int)	value
%s	string ((const) (unsigned) char *)	reference
%n	number of bytes written so far, (* int)	reference

```
printf ("a has value %d, b has value %d, c is at address: %08x\n",
                            a, b, &c);
```



[www.cis.syr.edu/~wedu/Teaching/cis643/LectureNotes_New/Format_String.pdf]

The format string tells the function the type and sequence of arguments to pop and how to format the output.

The FSV can be exploited if an attacker supplies a *bogus format string*.

If the number of format directives is *less* than the number of parameters, the string format function will *underflow* the stack.

If the number of directives *exceeds* the number of parameters, it will *overflow* the stack.

If the wrong directive is given for a parameter, the string format function will *misinterpret* the parameter.

If users can control the format string, they can *alter the memory space* of the process.

The **`%n`** directive assumes the corresponding argument to **`printf`** is of type **`int*`**.

It writes back the number of bytes formatted so far.

%n can be used to write an *arbitrary value* to an *arbitrary address*.

An attacker can send a single packet of data to a vulnerable program and obtain a remote shell.

In 2000, format bugs eclipsed buffer overflow vulnerabilities for the most common type of remote penetration vulnerability.

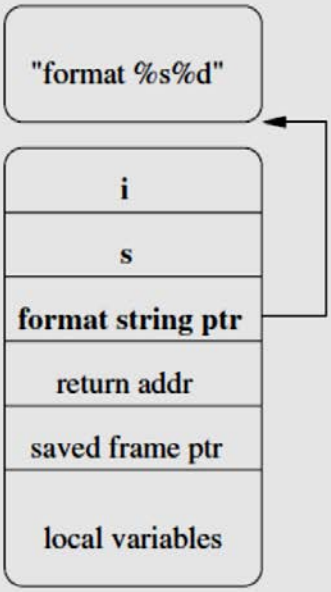


Figure 28. Activation record of *printf("format %s%d", s, i)* that has three parameters: the format string pointer, a string pointer and an integer. A format function uses an internal stack pointer to access the parameters in the stack as it encounters the directives in the format string.

```
void vulnfunc (char *user)
{
   ...
   printf(user);
}
```

Figure 29. A vulnerable function that has a user-supplied (or user-alterable) format string. The *printf* is most likely expecting printable characters only. It would be safe if it is called as *printf("%s", user)* instead.

[Lhee & Chapin]

We next discuss four exploitation techniques.

Viewing Memory

The **`printf`** below prints the values of five consecutive words on the stack, as 8-digit padded hex numbers:

```
printf("%08x %08x %08x %08x %08x");
```

In this way, we can *walk the stack* and *view its contents*.

Overwriting a Word in Memory

The **%n** directive writes the number of characters written by the format function so far, at the location pointed to be the corresponding parameter.

The following **`printf`** writes a small integer at address 0x08480110:

```
printf('\x10\x01\x48\x08%08x%08x%08x%08x%08x%n');
```

The five **%08x**'s pop the internal stack pointer so it points to the format string itself when **%n** is processed.

See Figure 30 below.

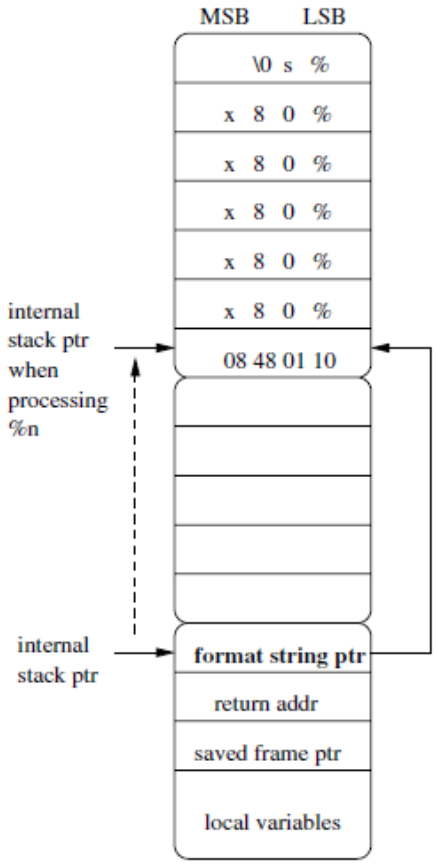


Figure 30. Writing an integer value at an arbitrary memory location.

[Lhee & Chapin]

The **`printf`** writes 44 at 0x08480110 (four characters for **`'\x10\x01\x48\x08'`** and eight characters for each **`'%08x'`**).

We can control the number of characters to be written by using the minimum field-width specifier if the specifier value is not too large.

Overflowing the Buffer Using the Minimum Field-Width Specifier

This overflows a buffer and *overwrites the return address* with the address of a code pointer.

In the function of Figure 31, **`safebuf`** can't be overflowed through the parameter **`user`** due to the use of **`snprintf`** (safe **`sprintf`**).

Hence, a conventional buffer-overflow (stack smashing) attack can't exploit the program.

The **`safebuf`** is the format string of the next **`sprintf`**, so it will be interpreted before being copied to **`vulnbuf`**.

For example, a format string **`'%513d'`** (in **`safebuf`**) will result in printing 513 characters to **`vulnbuf`**.

Figure 31 shows the exploit that overflows **`vulnbuf`** and overwrites the return address with the address of shellcode.

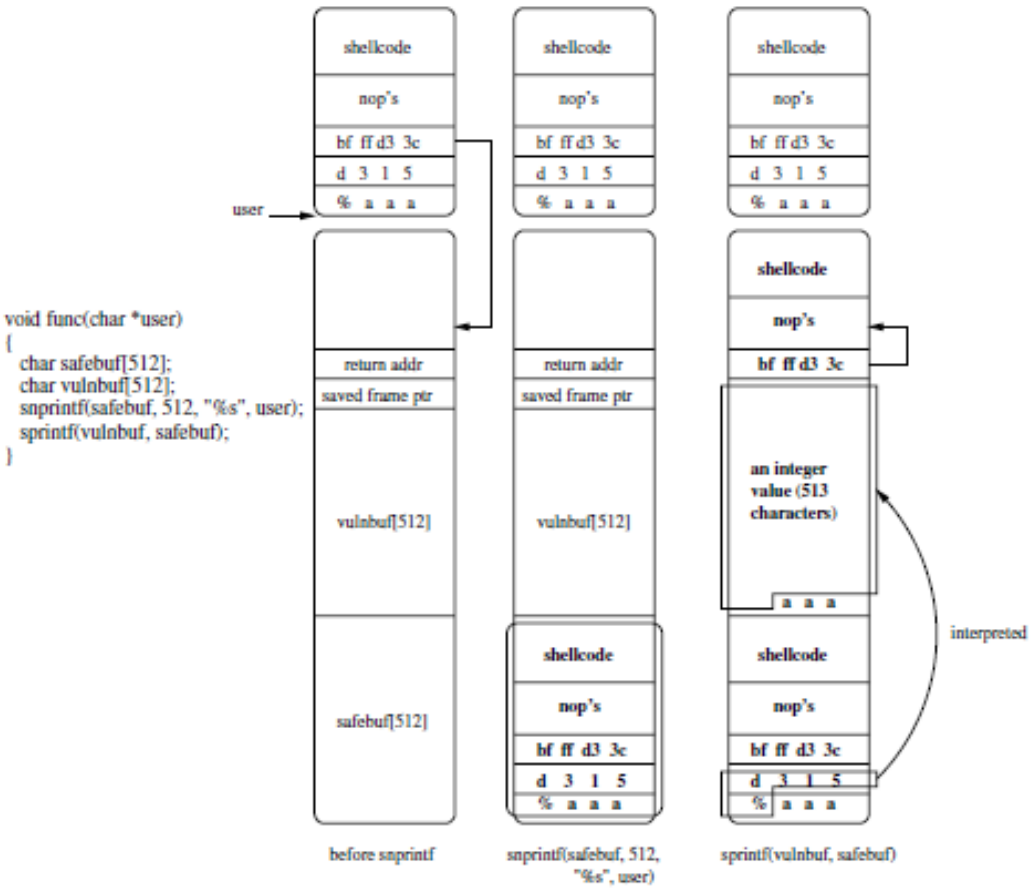


Figure 31. A program vulnerable to the minimum field width specifier and the exploit.

[Lhee & Chapin]

Overwriting a Code Pointer Using a %n Directive

We can't write an arbitrarily large integer value using the minimum field width specifier while exploiting the **%n** directive.

To write an arbitrary integer, we write one byte at a time, from least to most significant.

For example, we can write 0x80402010 by writing 0x10, 0x20, 0x40, and 0x80 (see Figure 32).

```
printf("%16u%n%16u%n%32u%n%64u%n",
       1, (int *)&foo[0], 1, (int *)&foo[1],
       1, (int *)&foo[2], 1, (int *)&foo[3]);
```

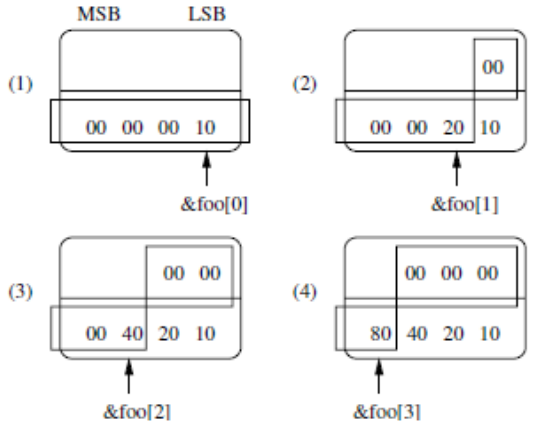


Figure 32. The four-stage word overwrite. Four bytes are written for each write (by a %n directive, inner square box). The first %16u writes 16 characters to *stdout*, so the following %n writes 16 (hexadecimal 0x10) at &foo[0]. The first parameter (integer 1) is a dummy parameter for %16u, so its value is unimportant. The next parameter &foo[0] is the address at which we want to overwrite. For each write we advance this address by one byte. The second %16u writes another 16 characters to *stdout*, a total of 32 characters so far. The following %n thus writes 32 (hexadecimal 0x20) at &foo[1], and so on.

Since we are writing four times within one string format function, the number of characters written so far is strictly increasing and so is the value of each write.

For each write, bytes other than the least significant one will be overwritten in the next write or not used.

Figures 33 shows an attack code that overwrites a code pointer using the four-stage overwrite.

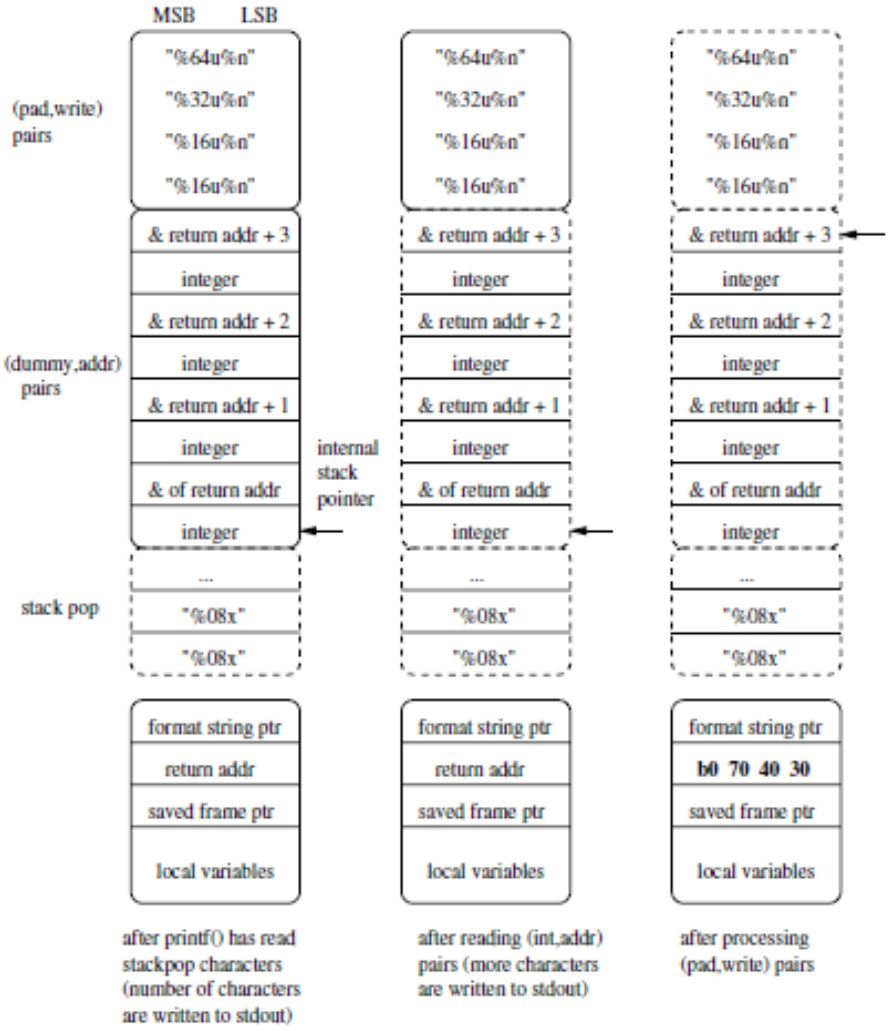


Figure 33. A pure format string overflow exploit. The stack popping directives move the internal stack pointer so that it points to the beginning of the dummy integer/address pairs. When *printf* reaches the pad/write-code pairs, it overwrites the address specified in the dummy/address pairs (since the internal stack pointer is pointing to the dummy/address pair). It overwrites the return address in this example.

Countermeasures

Remove the %n feature: breaks C programs

Permit only static format strings: breaks C programs

Count the arguments to `printf`: incompatible with existing libraries and programs

FormatGuard: uses properties of GNU CPP macro handling of variable arguments to extract the count of actual arguments.

This is passed to a safe **`printf`** wrapper.

The wrapper parses the format string to determine how many arguments to expect.

If the string calls for more than the actual number, it raises an intrusion alert and kills the process.

Libformat: is an implementation of a safe subset of C library format functions.

These functions check the format string.

If it is in a writable segment and contains “%n” directives, it is regarded as an attack.

Libformat is implemented as a shared library that is preloaded to intercept vulnerable format functions in the C library.

Its built-in heuristics can generate false alarms.

It is ineffective if programs are statically linked with the C library.