Shell Code For Beginners

Beenu Arora
Site: www.BeenuArora.com
Email: beenudel1986@gmail.com

```
#################################################################
#        .___              __          _______       .___       #
#      __| _/____ _______|  | __ ____  \   _  \    __| _/____   #
#     / __ |\__  \\_  __ \  |/ // ___\ /  /_\  \  / __ |/ __ \  #
#    / /_/ | / __ \|  | \/    <\  \___ \  \_/   \/ /_/ \  ___/  #
#    \____ |(______/__|  |__|_ \\_____>\_____  /\_____|\____\   #
#         \/                  \/             \/                 #
#                                                               #
#                 ___________   ______ _  __                    #
#               _/ ___\_  __ \_/ __ \ \/ \/ /                   #
#               \  \___|  | \/\  ___/\     /                    #
#                \___  >__|    \___  >\/\_/                     #
#      est.2007      \/            \/   forum.darkc0de.com      #
#################################################################
```

What is a shell Code?

Shellcode is defined as a set of instructions injected and then executed by an exploited program. Shellcode is used to directly manipulate registers and the functionality of a exploited program. We can of course write shell codes in the high level language but would let you know later why they might not work for some cases, so assembly language is preferred for this. I would take an clean example of the exit() syscall used for exiting from a program. Many of you might be wondered to see why this being used is, the reason is the newer kernel don't allow anymore the code execution from the stack so we have to use some C library wrapper or libc (responsible for providing us the malloc function).

Usage at darker site:

We write shellcode because we want the target program to function in a manner other than what was intended by the designer. One way to manipulate the program is to force it to make a system call or syscall. System calls in Linux are accomplished via software interrupts and are called with the int 0x80 instruction. When int 0x80 is executed by a user mode program, the CPU switches into kernel mode and executes the syscall function.

Then process followed as:

1. The specific syscall number is loaded into EAX.   This is the linux kernel command number which we given inside the EAX register (system call).

Ex: movl $1, %eax (used for the exit system call)

2. Arguments to the syscall function are placed in other registers. For information status number inside the ebx register.

Ex: movl $1, %ebx

3. The instruction int 0x80 is executed. To wake up the kernel to run the command.
4. The CPU switches to kernel mode.
5. The syscall function is executed.

The most basic syscall is exit ()

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

In assembly tt can be written as

```
Section .text

global _start

        _start:
        mov $0x0,%ebx
        mov $0x1,%eax
        int 0x80
```

Now disassemble the object file created from this using objdump to see the opcode.

Opcode	Instruction
bb 00 00 00 00	mov $0x0,%ebx
b8 01 00 00 00	mov $0x1,%eax
cd 80	int $0x80

To the left is our opcode. We need to do is place the opcode into a character array.

It would look like:

```
Char exitshell[] = “\xbb\x00\x00\x00\x00\xb8\x01\x00\x00\x00\xcd\x80”;
int main()
{
		int *ret;
		ret = (int *)&ret + 2;
		(*ret) = (int)exitshell;
}
```

The most likely place you will be placing shellcode is into a buffer allocated for user input. Even more likely, this buffer will be a character array. If you see the exit shell above :
\xbb\x00\x00\x00\x00\xb8\x01\x00\x00\x00\xcd\x80.we will notice that there are some nulls (\x00) present. These nulls will cause shellcode to fail when injected into a character array because the null character is used to terminate strings. To resolve this again there are many ways but again I would suggest most simple is to avoid the opcode which have the such null strings.

Now if we again look at the asm codes for the exit ()

The first statement is:

mov $0x0,%ebx : Corresponding debugged opcode : bb 00 00 00 00
The same thing can be implemented using the XOR since if the two registers have same value ten corresponding result is 0.

Newer one would be : xor %ebx,%ebx (This would store same 0 in ebx register)

The second one was
Mov $0x1,%eax : Corresponding debugged opcode : b8 01 00 00 00

Newer one would be: mov al, 1 (since the 1 is just 8 byte and we use the lower 8 bits of the 16 bit AX register (AL register))

The newer asm code for the exit() would be :

```
Section .text
global _start
        _start:
        xor ebx,ebx
        mov al,1
        int 0x80
```

The corresponding debugged opcode would be

```
Opcode                          Instruction

31 db                            xor %ebx,%ebx
b0 01                           mov $0x1,%al
cd 80                           int $0x80
```

So the first compact shell is now ready to be executed:

```
Char exitshell[] = "\x31\xdb\xb0\x01\xcd\x80";
int main()
{
        int *ret;
        ret = (int *)&ret + 2;
        (*ret) = (int)exitshell;
}
```

One thing we must keep in mind that shell codes has to be simple and compact since in real time condition where we have limited space in the buffer where we have to insert our shell as we as the return address to it .