

CHAPTER 7

Advanced Code Evolution Techniques and Computer Virus Generator Kits

"In mathematics you don't understand things. You just get used to them."

—John von Neumann

Chapter 7—Advanced Code Evolution Techniques and Computer Virus Generator Kits

In this chapter you will learn about the advanced self-protection techniques computer virus writers have developed over the years to fight back against scanners. In particular, you will learn about encrypted, oligomorphic, polymorphic¹, and advanced metamorphic computer viruses². Finally, we will look at computer virus generator kits³ that use similar techniques to create different-looking virus variants.

7.1 Introduction

We will examine the various ways in which virus writers have challenged our scanning products over the last decade. Although most of these techniques are used to obfuscate file-infector viruses, we can surely expect similar techniques to appear in future computer worms.

Over the years, code evolution has come a long way in binary viruses. If someone were to trace the development result of viruses, it might appear that almost everything possible has already been done, and problems are not escalating. However, there are still computing distribution models that have not yet been seen in viruses.

7.2 Evolution of Code

Virus writers continually challenge antivirus products. Their biggest enemies are the virus scanner products that are the most popular of current antivirus software. Generic AV solutions, such as integrity checking and behavior blocking, never managed to approach the popularity of the antivirus scanner.

In fact, such generic virus detection models need a lot more thought and technology in place under Windows platforms. These technologies were beaten by some of the old DOS viruses in the DOS days. As a result, some people draw the incorrect conclusion that these techniques are not useful.

Scanning is the market's accepted solution, regardless of its drawbacks. Thus it must be able to deal with the escalating complexity and emerging number of distributed and self-distributing malware.

Although modern computing developed extremely quickly, for a long time binary virus code could not catch up with the technological challenges. In fact, the DOS viruses evolved to a very complex level until 1996. At that point, however, 32-bit Windows started to dominate the market. As a result, virus writers had to go back years in binary virus development. The complexity of DOS polymorphism

7.3 Encrypted Viruses

peaked when Ply was introduced in 1996 with a new permutation engine (although the metamorphic virus, ACG, was introduced in 1998). These developments could not continue. New 32-bit infection techniques had to be discovered by the pioneer virus writers and later on Win32 platforms.

Some virus writers still find the Windows platforms far too challenging, especially when it comes to Windows NT/2000/XP/2003. The basic infection techniques, however, have already been introduced, and standalone virus assembly sources are distributed widely on the Internet. These sources provide the basis of new mass-mailing worms that do not require major skills—just cut and paste abilities.

In the following sections, we will examine the basic virus code obfuscation techniques, from encrypted viruses to modern metamorphic techniques.

7.3 Encrypted Viruses

From the very early days, virus writers tried to implement virus code evolution. One of the easiest ways to hide the functionality of the virus code was *encryption*. The first known virus that implemented encryption was Cascade on DOS⁴. The virus starts with a constant decryptor, which is followed by the encrypted virus body. Consider the example extracted from Cascade.1701 shown in Listing 7.1.

Listing 7.1

The Decryptor of the Cascade Virus

```
lea    si, Start ; position to decrypt (dynamically set)
mov    sp, 0682 ; length of encrypted body (1666 bytes)
```

Decrypt:

```
xor    [si],si ; decryption key/counter 1
xor    [si],sp ; decryption key/counter 2
inc    si ; increment one counter
dec    sp ; decrement the other
jnz    Decrypt ; loop until all bytes are decrypted
```

```
Start: ; Encrypted/Decrypted Virus Body
```

Note that this decryptor has antidebug features because the SP (stack pointer) register is used as one of the decryption keys. The direction of the decryption loop is always forward; the SI register is incremented by one.

Because the SI register initially points to the start of the encrypted virus body, its initial value depends on the relative position of the virus body in the file.

Chapter 7—Advanced Code Evolution Techniques and Computer Virus Generator Kits

Cascade appends itself to the files, so SI will result in the same value if two host programs have equivalent sizes. However, the SI (decryption key 1) is changed if the host programs have different sizes. The SP register is a simple counter for the number of bytes to decrypt. Note that the decryption is going forward with word (double-byte) key length. The decryption position, however, is moved forward by one byte each time. This complicates the decryption loop, but it does not change its reversibility. Note that simple XOR is very practical for viruses because XORing with the same value twice results in the initial value.

Consider encrypting letter *P* (0x50) with the key 0x99. You see, 0x50 XOR 0x99 is 0xC9, and 0xC9 XOR 0x99 will return to 0x50. This is why virus writers like simple encryption so much—they are lazy! They can avoid implementing two different algorithms, one for the encryption and one for the decryption.

Cryptographically speaking, such encryption is weak, though early antivirus programs had little choice but to pick a detection string from the decryptor itself. This led to a number of problems, however. Several different viruses might have the same decryptor, but they might have completely different functionalities. By detecting the virus based on its decryptor, the product is unable to identify the variant or the virus itself. More importantly, nonviruses, such as antidebug wrappers, might have a similar decryptor in front of their code. As a result, the virus that uses the same code to decrypt itself will confuse them.

Such a simple code evolution method also appeared in 32-bit Windows viruses very early. W95/Mad and W95/Zombie use the same technique as Cascade. The only difference is the 32-bit implementation. Consider the decryptor from the top of W95/Mad.2736, shown in Listing 7.2.

Listing 7.2

The Decryptor of the W95/Mad.2736 Virus

```

mov     edi,00403045h ; Set EDI to Start
add     edi,ebp      ; Adjust according to base
mov     ecx,0A6Bh   ; length of encrypted virus body
mov     al,[key]    ; pick the key

Decrypt:
xor     [edi],al    ; decrypt body
inc     edi         ; increment counter position
loop   Decrypt     ; until all bytes are decrypted
jmp    Start       ; Jump to Start (jump over some data)

DB     key         86          ; variable one byte key
Start:                               ; encrypted/decrypted virus body

```

7.3 Encrypted Viruses

In fact, this is an even simpler implementation of the simple XOR method. Detection of such viruses is still possible without trying to decrypt the actual virus body. In most cases, the code pattern of the decryptor of these viruses is unique enough for detection. Obviously, such detection is not exact, but the repair code can decrypt the encrypted virus body and easily deal with minor variants.

The attacker can implement some interesting strategies to make encryption and decryption more complicated, further confusing the antivirus program's detection and repair routines:

- The direction of the loop can change: forward and backward loops are supported (see all cases in Figure 7.1).
- Multiple layers of encryption are used. The first decryptor decrypts the second one, the second decrypts the third, and so on (see Figure 7.1c.). Hare⁵ by Demon Emperor, W32/Harrier⁶ by TechnoRat, {W32, W97M}/Coke by Vecna, and W32/Zelly by ValleZ are examples of viruses that use this method.
- Several encryption loops take place one after another, with randomly selected directions—forward and backward loops. This technique scrambles the code the most (see Figure 7.1c.).
- There is only one decryption loop, but it uses more than two keys to decrypt each encrypted piece of information on the top of the others. Depending on the implementation of the decryptor, such viruses can be much more difficult to detect. The size of the key especially matters—the bigger the key size (8, 16, 32 -bit, or more), the longer the brute-force decryption might take if the keys cannot be extracted easily.

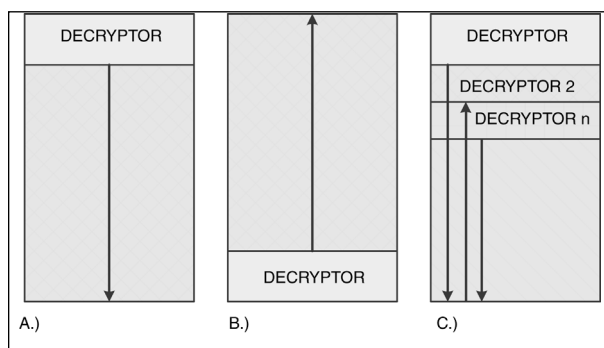


Figure 7.1 Decryption loop examples.

Chapter 7—Advanced Code Evolution Techniques and Computer Virus Generator Kits

- The start of decryptor is obfuscated. Some random bytes are padded between the decryptor and the encrypted body and/or the encrypted body and the end of the file.
- Nonlinear decryption is used. Some viruses, such as W95/Fono, use a simple nonlinear algorithm with a key table. The virus encryption is based on a substitution table. For instance, the virus might decide to swap the letters *A* and *Z*, the letters *P* and *L*, and so on. Thus the word *APPLE* would look like *ZLLPE* after such encryption.

Because the virus decryption is not linear, the virus body is not decrypted one byte after another. This easily might confuse a junior virus analyst because in some cases, the virus body might not look encrypted at all. Consequently, if a detection string is picked from such a sample, the virus detection will be partial. This technique easily can confuse even advanced detection techniques that use an emulator. Although in normal cases the emulation can continue until linear detection is detected, such as consecutive byte changes in the memory of a virtual machine used by the scanner, a nonlinear algorithm will force the emulation to continue until a hard-to-guess minimum limit.

A variant of the W32/Chiton (“Efish”) virus uses a similar approach to Fono’s, but Chiton makes sure it always replaces each byte of the virus body with another value using a complete substitution table. In addition, Chiton uses multiple values to correspond to each byte in the code, significantly complicating the decryption.

Viruses such as W95/Drill and {W32, Linux}/Simile.D represent the state of the art in nonlinear encryption, decrypting each piece of the encrypted virus body in a semi-random order, hitting each position in the virus only once.⁷

- The attacker can decide not to store the key for encryption anywhere in the virus. Instead, the virus uses brute force to decrypt itself, attempting to recover the encryption keys on its own. Viruses like this are much harder to detect and said to use the RDA (random decryption algorithm) technique. The RDA.Fighter virus is an example that uses this method.
- The attacker can use a strong encryption algorithm to encrypt the virus. The IDEA family of viruses, written by Spanska, utilizes this method. One of several decryptors uses the IDEA cipher.⁸ Because the virus carries the key for the decryption, the encryption cannot be considered strong, but the repair of such viruses is painful because the antivirus needs to reimplement the

7.3 Encrypted Viruses

encryption algorithm to deal with it. In addition, the second decryption layer of IDEA virus⁹ uses RDA.

- The Czech virus W32/Crypto by Prizzy demonstrated the use of Microsoft crypto API in computer viruses. Crypto encrypts DLLs on the system using a secret/public key pair generated on the fly. Other computer worms and back-door programs also use the Crypto API to decrypt encrypted content. This makes the job of antivirus scanners more difficult. An example of a computer worm using the Crypto API is W32/Qint@mm, which encrypts EXE files.
- Sometimes the decryptor itself is not part of the virus. Viruses such as W95/Resur¹⁰ and W95/Silcer are examples of this method. These viruses force the Windows Loader to relocate the infected program images when they are loaded to memory. The act of relocating the image is responsible for decrypting the virus body because the virus injects special relocations for the purpose of decryption. The image base of the executable functions as the encryption key.
- The Cheeba virus demonstrated that the encryption key can be external to the virus body. Cheeba was released in 1991. Its payload is encrypted using a filename. Only when the virus accesses the file name will it correctly decrypt its payload¹¹. Virus researchers cannot easily describe the payload of such virus unless the cipher in the virus is weak. Dmitry Gryaznov managed to reduce the key size necessary to attack the cipher in Cheeba to only 2,150,400 possible keys by using frequency cryptanalysis of the encrypted virus body, assuming that the code under the encryption was written in a similar style as the rest of the virus code¹². This yielded the result, and the magic filename, “users.bbs” was found. This filename belonged to a popular bulletin board software. It is expected that more, so-called “clueless agents”¹³ will appear as computer viruses to disallow the defender to gain knowledge about the intentions of the attacker.
- Encryption keys can be generated in different ways, such as constant, random but fixed, sliding, and shifting.
- The key itself can be stored in the decryptor, in the host, or nowhere at all. In some cases, the decryptor’s code functions as a decryption key, which can cause problems if the code of the decryptor is modified with a debugger. Furthermore, this technique can attack emulators that use code optimization techniques to run decryptors more efficiently. (An example of such as virus is Tequila.)

Chapter 7—Advanced Code Evolution Techniques and Computer Virus Generator Kits

- The randomness of the key is also an important factor. Some viruses only generate new keys once per day and are said to use a slow generator. Others prefer to generate keys every single time they infect an object; these are known as fast generators. The attacker can use many different methods to select the seed of randomness. Simple examples include timer ticks, CMOS time and date, and CRC32. A complicated example is the Mersenne Twister¹⁴ pseudo-number generator used by W32/Chiton and W32/Beagle.
- The attacker can select several locations to decrypt the encrypted content. The most common methods are shown in Figure 7.2.

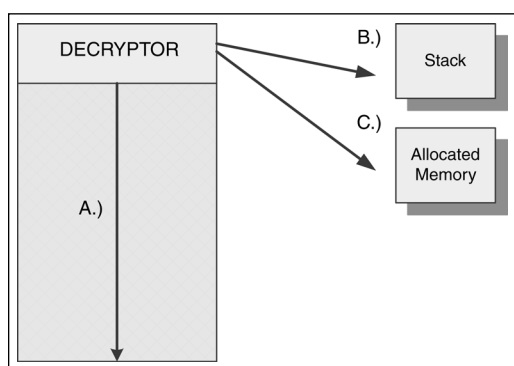


Figure 7.2 Possible places of decryption. A) The decryptor decrypts the data at the location of the encrypted virus body. This method is the most common; however, the encrypted data must be writeable in memory, which depends on the actual operating system. B) The decryptor reads the encrypted content and builds the decrypted virus body on the stack. This is very practical for the attacker. The encrypted data does not need to be writeable. C) The virus allocates memory for the decrypted code and data. This can be a serious disadvantage for the attacker because nonencrypted code needs to allocate memory first—before the decryptor.

Note

Metamorphic viruses such as Simile circumvent this disadvantage because the code that allocates memory is made variable without providing the ability to pick a search string.

The preceding techniques work very effectively when combined with variable decryptors that keep changing in new generations of the virus. Oligomorphic and polymorphic decryption are discussed in the following sections.

7.4 Oligomorphic Viruses

7.4 Oligomorphic Viruses

Virus writers quickly realized that detection of an encrypted virus remains simple for the antivirus software as long as the code of the decryptor itself is long enough and unique enough. To challenge the antivirus products further, they decided to implement techniques to create mutated decryptors.

Unlike encrypted viruses, *oligomorphic viruses* do change their decryptors in new generations. The simplest technique to change the decryptors is to use a set of decryptors instead of a single one. The first known virus to use this technique was Whale. Whale carried a few dozen different decryptors, and the virus picked one randomly.

W95/Memorial had the ability to build 96 different decryptor patterns. Thus the detection of the virus based on the decryptor's code was an impractical solution, though a possible one. Most products tried to deal with the virus by dynamic decryption of the encrypted code. The detection is still based on the constant code of the decrypted virus body.

Consider the example of Memorial shown in Listing 7.3, a particular instance of 96 different cases.

Listing 7.3

An Example Decryptor of the W95/Memorial Virus

```

mov     ebp,00405000h      ; select base
mov     ecx,0550h         ; this many bytes
lea     esi,[ebp+0000002E] ; offset of "Start"
add     ecx,[ebp+00000029] ; plus this many bytes
mov     al,[ebp+0000002D] ; pick the first key

Decrypt:
nop                    ; junk
nop                    ; junk
xor     [esi],al        ; decrypt a byte
inc     esi             ; next byte
nop                    ; junk
inc     al              ; slide the key
dec     ecx             ; are there any more bytes to decrypt?
jnz    Decrypt         ; until all bytes are decrypted
jmp    Start           ; decryption done, execute body

; Data area

Start:
;     encrypted/decrypted virus body

```

Chapter 7—Advanced Code Evolution Techniques and Computer Virus Generator Kits

Notice the sliding-key feature. The order of the instructions can be slightly changed, and the decryptor can use different instructions for looping.

Compare this example with another instance, shown in Listing 7.4.

Listing 7.4

A Slightly Different Decryptor of W95/Memorial

```

mov     ecx,0550h           ; this many bytes
mov     ebp,013BC000h      ; select base
lea     esi,[ebp+0000002E] ; offset of "Start"
add     ecx,[ebp+00000029] ; plus this many bytes
mov     al,[ebp+0000002D] ; pick the first key

Decrypt:
nop                ; junk
nop                ; junk
xor     [esi],al          ; decrypt a byte
inc     esi               ; next byte
nop                ; junk
inc     al                ; slide the key
loop   Decrypt           ; until all bytes are decrypted
jmp    Start             ; Decryption done, execute body

; Data area

Start:
;     Encrypted/decrypted virus body

```

Notice the appearance of a “loop” instruction in this instance, as well as the swapped instructions in the front of the decryptor. A virus is said to be oligomorphic if it is capable of mutating its decryptor only slightly.

Interestingly, some products that we tested could not detect all instances of Memorial. This is because such viruses must be examined to their smallest details to find and understand the oligomorphic decryptor generator. Without such careful manual analysis, the slow oligomorphic virus techniques are impossible to detect with any reliability. For example, the decryptor of the Badboy virus¹⁵ changes in one instruction—and only very rarely. Obviously, they are a great challenge for automated virus analysis centers.

Another early example of an oligomorphic virus is the Russian virus family called WordSwap.

7.5 Polymorphic Viruses

7.5 Polymorphic Viruses

The next step in the level of difficulty is a *polymorphic* attack. Polymorphic viruses can mutate their decryptors to a high number of different instances that can take millions of different forms.

7.5.1 The 1260 Virus

The first known polymorphic virus, 1260, was written in the U.S. by Mark Washburn in 1990¹⁶. This virus has many interesting techniques that were previously predicted by Fred Cohen. The virus uses two sliding keys to decrypt its body, but more importantly, it inserts junk instructions into its decryptor. These instructions are garbage in the code. They have no function other than altering the appearance of the decryptor.

Virus scanners were challenged by 1260 because simple search strings could no longer be extracted from the code. Although 1260's decryptor is very simple, it can become shorter or longer according to the number of inserted junk instructions and random padding after the decryptor for up to 39 bytes of junk instructions. In addition, each group of instructions (prolog, decryption, and increments) within the decryptor can be permuted in any order. Thus the "skeleton" of the decryptor can change as well. Consider the example of an instance of a decryptor extracted from 1260 (see Listing 7.5).

Listing 7.5

An Example Decryptor of 1260

```
; Group 1 - Prolog Instructions
inc     si           ; optional, variable junk
mov     ax,0E9B     ; set key 1
clc     ; optional, variable junk
mov     di,012A     ; offset of Start
nop     ; optional, variable junk
mov     cx,0571     ; this many bytes - key 2

; Group 2 - Decryption Instructions
Decrypt:
xor     [di],cx     ; decrypt first word with key 2
sub     bx,dx       ; optional, variable junk
xor     bx,cx       ; optional, variable junk
sub     bx,ax       ; optional, variable junk
sub     bx,cx       ; optional, variable junk
nop     ; non-optional junk
xor     dx,cx       ; optional, variable junk
xor     [di],ax     ; decrypt first word with key 1
```

continues

Chapter 7—Advanced Code Evolution Techniques and Computer Virus Generator Kits

Listing 7.5 continued

An Example Decryptor of 1260

```
; Group 3 - Decryption Instructions
inc    di          ; next byte
nop                ; non-optional junk
clc                ; optional, variable junk
inc    ax          ; slide key 1
; loop
loop   Decrypt    ; until all bytes are decrypted - slide key 2
; random padding up to 39 bytes
```

Start:

```
; Encrypted/decrypted virus body
```

In each group of instructions, up to five junk instructions are inserted (INC SI, CLC, NOP, and other do-nothing instructions) with no repetitions allowed in the junk. There are two NOP junk instructions that always appear.

1260 does not have register replacement, but more complicated polymorphic attacks use that trick. Nonetheless, 1260 is an effective polymorphic engine that generates a high variety of decryptors.

7.5.2 The Dark Avenger Mutation Engine (MtE)

The next important development in the history of polymorphic viruses was *MtE*¹⁷, a mutation engine written by the Bulgarian Dark Avenger. The first version MtE was released during the summer of 1991, later followed by another version in early 1992. The idea of the mutation engine is based on modular development. For novice virus writers, it was difficult to write a polymorphic virus. However, more advanced virus writers came to their rescue. The MtE engine was released as an object that could be linked to any simple virus.

The concept of MtE is to make a function call to the mutation engine function and pass control parameters in predefined registers. The engine takes care of building a polymorphic shell around the simple virus inside it.

The parameters to the engine include the following:

- A work segment
- A pointer to the code to encrypt
- Length of the virus body
- Base of the decryptor
- Entry-point address of the host
- Target location of encrypted code

7.5 Polymorphic Viruses

- Size of decryptor (tiny, small, medium, or large)
- Bit field of registers not to use

In response, the MtE engine returns a polymorphic decryption routine with an encrypted virus body in the supplied buffer. (See Listing 7.6.)

Listing 7.6

An Example Decryptor Generated by MtE

```

mov     bp,A16C      ; This Block initializes BP
                    ; to "Start"-delta
mov     cl,03        ; (delta is 0x0D2B in this example)
ror     bp,cl
mov     cx,bp
mov     bp,856E
or      bp,740F
mov     si,bp
mov     bp,3B92
add     bp,si
xor     bp,cx
sub     bp,B10C      ; Huh ... finally BP is set, but remains an
                    ; obfuscated pointer to encrypted body

Decrypt:
mov     bx,[bp+0D2B] ; pick next word
                    ; (first time at "Start")
add     bx,9D64      ; decrypt it
xchg    [bp+0D2B],bx ; put decrypted value to place

mov     bx,8F31      ; this block increments BP by 2
sub     bx,bp
mov     bp,8F33
sub     bp,bx        ; and controls the length of decryption

jnz     Decrypt     ; are all bytes decrypted?

Start:
                    ; encrypted/decrypted virus body

```

This example of MtE illustrates how remarkably complicated this particular engine is. Can you guess how to detect it?

It makes sense to look at a large set of samples first. The first time, it took me five days before I managed to write a reliable detector for it. MtE could produce some decryptor cases that appeared only in about 5% or less of all cases. However, the engine had a couple of minor limitations that were enough to detect the virus reliably using an instruction size disassembler and a state machine. In fact, there

Chapter 7—Advanced Code Evolution Techniques and Computer Virus Generator Kits

is only one constant byte in an MtE decryptor, the 0x75 (JNZ), which is followed by a negative offset—and even that is placed at a variable location (at the end of the decryptor, whose length is not constant).

Note

MtE does not have garbage instructions in the decryptor, as 1260 does. Therefore MtE attacks techniques that attempt to optimize decryptors to defeat polymorphism.

MtE's impact on antivirus software was clear. Most AV engines had to go through a painful rearchitecting to introduce a virtual machine for the use of the scanning engine. As Frans Veldman used to say, "We simply let the virus do the dirty work."

MtE was quickly followed by many similar engines, such as TPE (Trident Polymorphic Engine), written by Masouf Khafir in Holland in 1993.

Today, hundreds of polymorphic engines are known. Most of these engines were only used to create a couple of viruses. After a polymorphic decryptor can be detected, using it becomes a disadvantage to virus writers because any new viruses are covered by the same detection. Such detections, however, usually come with the price of several false positives and false negatives. More reliable techniques detect and recognize the virus body itself.

This opens up the possibility for virus writers to use the same polymorphic engine in many different viruses successfully unless such viruses are handled with heuristic or generic detection methods.

7.5.3 32-Bit Polymorphic Viruses

W95/HPS and W95/Marburg¹⁸ were the first viruses to use real 32-bit polymorphic engines. These two viruses were authored by the infamous Spanish virus writer, GriYo, in 1998. He also created several highly polymorphic viruses earlier on DOS, such as the virus Implant¹⁹.

Just like Implant's polymorphic engine, HPS's polymorphic engine is powerful and advanced. It supports subroutines using CALL/RET instructions and conditional jumps with nonzero displacement. The code of the polymorphic engine takes about half of the actual virus code, and there are random byte-based blocks inserted between the generated code chains of the decryptor. The full decryptor is built only during the first initialization phase, which makes the virus a slow polymorphic. This means that antivirus vendors cannot test their scanner's detection

7.5 Polymorphic Viruses

rate efficiently because the infected PC must be rebooted to force the virus to create a new decryptor.

The decryptor consists of Intel 386-based instructions. The virus body is encrypted and decrypted by different methods, including XOR/NOT and INC/DEC/SUB/ADD instructions with 8, 16, or 32-bit keys, respectively. From a detection point of view, this drastically reduces the range of ideas. I am sad to say that the polymorphic engine was very well written, just like the rest of the virus. It was certainly not created by a beginner.

Consider the following example of a decryptor, simplified for illustration. The polymorphic decryptor of the virus is placed after the variably encrypted virus body. The decryptor is split between small islands of code routines, which can appear in mixed order. In the example shown in Listing 7.7, the decryptor starts at the `Decryptor_Start` label, and the decryption continues until the code finally jumps to the decrypted virus body.

Listing 7.7

An Illustration of a W95/Marburg Decryptor Instance

```

Start:
                                ; Encrypted/Decrypted Virus body is placed here

Routine-6:
dec     esi                    ; decrement loop counter
ret

Routine-3:
mov     esi,439FE661h         ; set loop counter in ESI
ret

Routine-4:
xor     byte ptr [edi],6F     ; decrypt with a constant byte
ret

Routine-5:
add     edi,0001h            ; point to next byte to decrypt
ret

Decryptor_Start:
call    Routine-1            ; set EDI to "Start"
call    Routine-3            ; set loop counter

Decrypt:
call    Routine-4            ; decrypt
call    Routine-5            ; get next
call    Routine-6            ; decrement loop register

```

continues

Chapter 7—Advanced Code Evolution Techniques and Computer Virus Generator Kits

Listing 7.7 continued

An Illustration of a W95/Marburg Decryptor Instance

```

cmp     esi,439FD271h    ; is everything decrypted?
jnz     Decrypt         ; not yet, continue to decrypt
jmp     Start           ; jump to decrypted start

```

```

Routine-1:
call    Routine-2       ; Call to POP trick!

```

```

Routine-2:
pop     edi
sub     edi,143Ah       ; EDI points to "Start"
ret

```

The preceding decryptor is highly structural, with each differently functioning piece placed in its own routine. The result is millions of possible code patterns filled with random garbage instructions between the islands.

Polymorphic viruses can create an endless number of new decryptors that use different encryption methods to encrypt the constant part (except their data areas) of the virus body.

Some polymorphic viruses, such as W32/Coke, use multiple layers of encryption. Furthermore, variants of Coke also can infect Microsoft Word documents in a polymorphic way. Coke mutates its macro directly with the binary code of the virus instead of using macro code directly. Normally, polymorphic macro viruses are very slow because they need a lot of interpretation. Because Coke generates mutated macros with binary code, it does not suffer from slow-down issues and, as a result, is harder to notice. Consider the following example of Coke taken from two instances of mutated AutoClose() macros shown in Listing 7.8.

Listing 7.8

An Example Snippet of Coke's Polymorphic Macro

```

'BsbK
Sub AuT0c10SE()
oN ERR0r RESuMe NeXT
SHOWviSuALBASiCEdit0r = faLsE
If nmnGG > WYff Then
For XgfqLwDTT = 70 To 5
JhGPTT = 64
KjfLL = 34
If qqSsKWW < vMmm Then
For QpMM = 56 To 7
If qtWQHU = PCYKWvQQ Then
If lXynNrr > mxTwjWW Then
End If

```


7.5 Polymorphic Viruses

```
If FFnfrjj > GHgpE Then
End If
```

The second example is a little longer because of the junk. I have highlighted some of the essential instructions in these examples. Notice that even these are not presented in the same order whenever possible. For example, the preceding instance turns off the Visual Basic Editor, so you will no longer see it in Word's menus. However, in Listing 7.9, this happens later, after lots of inserted junk and other essential instructions.

Listing 7.9

Another Snippet of Coke's Polymorphic Macro

```
'fYJm
Sub AUt0cL0se()
oN ERRor REsUME NexT
optiOns.saVenorMALPrOmpT = fAlse
DdXLwjvVlQxU$ = "TmDKK"
NrCyxbahfPtt$ = "fnMM"
If MKbyqtt > mHba Then
If JluVV > mkpSS Then
jMJFFXkTfgMM$ = "DmJcc"
For VPQjTT = 42 To 4
If PGNwygui = bMVrr Then
dJTKQi = 07
'wcHpsxllwuCC
End If
Next VPQjTT
quYY = 83
End If
DsSS = 82
bFVpp = 60
End If
tCQFv=1
Rem kJPpjNNGQCVpjj
LyBDXXXGnWW$ = "wPyTd1e"
If cnkCvCww > FupJLQSS Then
VbBCCcxKWxww$ = "Ybrr"
End If
optiOnS.COnfIrmCOnvErsiOnS = faLse
Svye = 55
PgHKfiVXuFF$ = "rHKVMdd"
ShOwVisUALbaSiCEdITOR = fALse
```

Newer polymorphic engines use an RDA-based decryptor that implements a brute-force attack against its constant but variably encrypted virus body in a multi-encrypted manner. Manual analysis of such viruses might lead to great surprises.

Chapter 7—Advanced Code Evolution Techniques and Computer Virus Generator Kits

Often there are inefficiencies of randomness in such polymorphic engines, which can be used as a counterattack. Sometimes even a single wildcard string can do the magic of perfect detection.

Most virus scanners, years ago, already had a code emulator capable of emulating 32-bit PE (portable executable) files. Other virus researchers only implemented dynamic decryption to deal with such viruses. That worked just as in the previous cases because the virus body was still constant under encryption. According to the various AV tests, some vendors were still sorry not to have support for difficult virus infection techniques.

Virus writers used the combination of entry-point obscuring techniques with 32-bit polymorphism to make the scanner's job even more difficult. In addition, they tried to implement antiemulation techniques to challenge code emulators.

Nevertheless, all polymorphic viruses carry a constant code virus body. With advanced methods, the body can be decrypted and identified. Consider Figure 7.3 as an illustration.

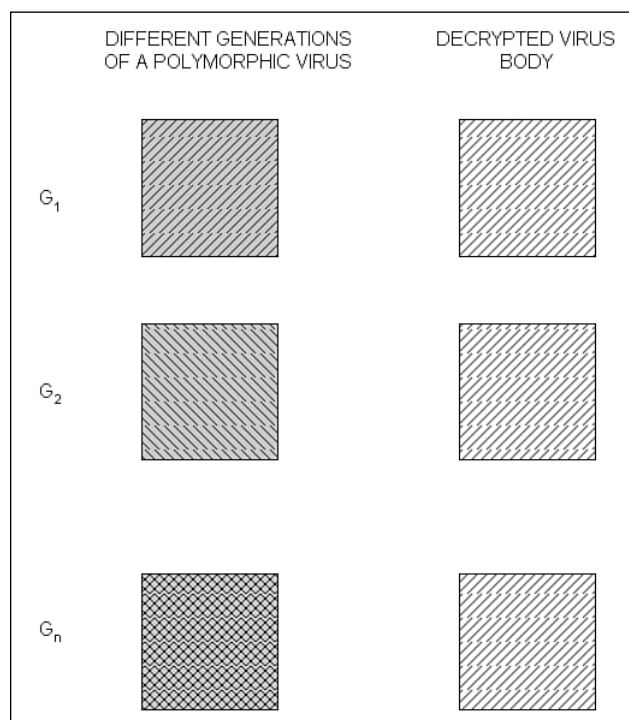


Figure 7.3 Instances of encrypted and decrypted polymorphic virus bodies.

7.6 Metamorphic Viruses

7.6 Metamorphic Viruses

Virus writers still must often waste weeks or months to create a new polymorphic virus that does not have chance to appear in the wild because of its bugs. On the other hand, a researcher might be able to deal with the detection of such a virus in a few minutes or few days. One of the reasons for this is that there are a surprisingly low number of efficient external polymorphic engines.

Virus writers try, of course, to implement various new code evolution techniques to make the researcher's job more difficult. The W32/Apparition virus was the first-known 32-bit virus that did not use polymorphic decryptors to evolve itself in new generations. Rather, the virus carries its source and drops it whenever it can find a compiler installed on the machine. The virus inserts and removes junk code to its source and recompiles itself. In this way, a new generation of the virus will look completely different from previous ones. It is fortunate that W32/Apparition did not become a major problem. However, such a method would be more dangerous if implemented in a Win32 worm. Furthermore, these techniques are even more dangerous on platforms such as Linux, where C compilers are commonly installed with the standard system, even if the system is not used for development. In addition, MSIL (Microsoft Intermediate Language) viruses already appeared to rebuild themselves using the System.Reflection.Emit namespace and implement a permutation engine. An example of this kind of metamorphic engine is the MSIL/Gastropod virus, authored by the virus writer, Whale.

The technique of W32/Apparition is not surprising. It is much simpler to evolve the code in source format rather than in binary. Not surprisingly, many macro and script viruses use junk insertion and removal techniques to evolve in new generations²⁰.

7.6.1 What Is a Metamorphic Virus?

Igor Muttik explained metamorphic viruses in the shortest possible way: "Metamorphics are body-polymorphics." Metamorphic viruses do not have a decryptor or a constant virus body but are able to create new generations that look different. They do not use a data area filled with string constants but have one single-code body that carries data as code.

Material metamorphosis does exist in real life. For instance, shape memory polymers have the ability to transform back to their parent shape when heated²¹. Metamorphic computer viruses have the ability to change their shape by themselves from one form to another, but they usually avoid generating instances that are very close to their parent shape.

Chapter 7—Advanced Code Evolution Techniques and Computer Virus Generator Kits

Figure 7.4 illustrates the problem of metamorphic virus bodies as multiple shapes.

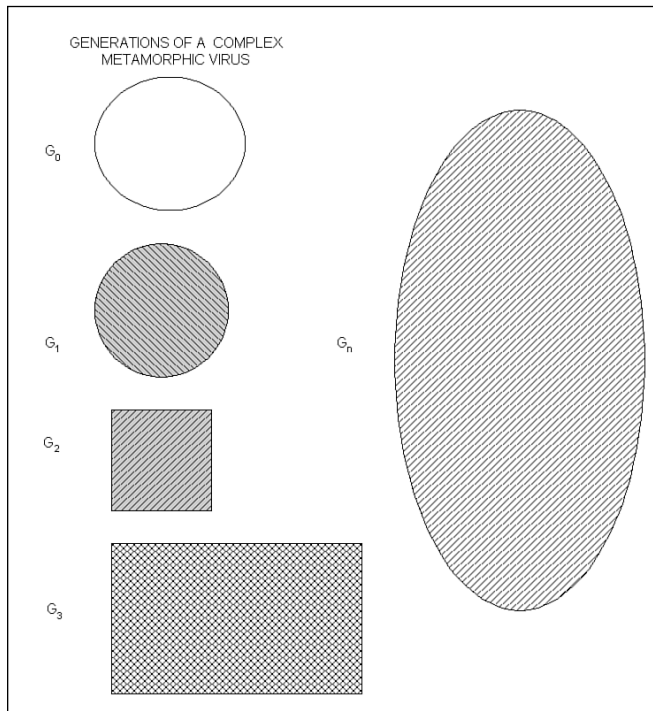


Figure 7.4 The virus body keeps changing in different generations of a metamorphic virus.

Although there are some DOS metamorphic viruses, such as ACG (Amazing Code Generator), these did not become a significant problem for end users. There are already more metamorphic Windows viruses than metamorphic DOS viruses. The only difference between the two is in their respective potentials. The networked enterprise gives metamorphic binary worms the ability to cause major problems. As a result, we will not be able to turn a blind eye to them, assuming that we do not need to deal with them because they are not causing problems. They will.

7.6.2 Simple Metamorphic Viruses

In December of 1998, Vecna (a notorious virus writer) created the W95/Regswap virus. Regswap implements metamorphosis via register usage exchange. Any part of the virus body will use different registers but the same code. The complexity of this, clearly, is not very high. Listing 7.10 shows some sample code fragments

7.6 Metamorphic Viruses

selected from two different generations of W95/Regswap that use different registers.

Listing 7.10

Two Different Generations of W95/Regswap

a.)

```

5A                pop     edx
BF04000000       mov     edi,0004h
8BF5             mov     esi,ebp
B80C000000       mov     eax,000Ch
81C288000000     add     edx,0088h
8B1A             mov     ebx,[edx]
899C8618110000   mov     [esi+eax*4+00001118],ebx

```

b.)

```

58                pop     eax
BB04000000       mov     ebx,0004h
8BD5             mov     edx,ebp
BF0C000000       mov     edi,000Ch
81C088000000     add     eax,0088h
8B30             mov     esi,[eax]
89B4BA18110000   mov     [edx+edi*4+00001118],esi

```

The bold areas show the common areas of the two code generations. Thus a wildcard string could be useful in detecting the virus. Moreover, support for half-byte wildcard (indicated with the ? mark) bytes such as 5? B? (as described by Frans Veldman) could lead to even more accurate detection. Using the 5?B? wildcard pattern we can detect snippets such as 5ABF, 58BB, and so on.

Depending on the actual ability of the scanning engine, however, such a virus might need an algorithmic detection because of the missing support of wildcard search strings. If algorithmic detection is not supported as a single database update, the product update might not come out for several weeks—or months—for all platforms!

Other virus writers tried to re-create older permutation techniques. For instance, the W32/Ghost virus has the capability to reorder its subroutines similarly to the BadBoy DOS virus family (see Figure 7.5). Badboy always starts in the entry point (EP) of the virus.

Chapter 7—Advanced Code Evolution Techniques and Computer Virus Generator Kits

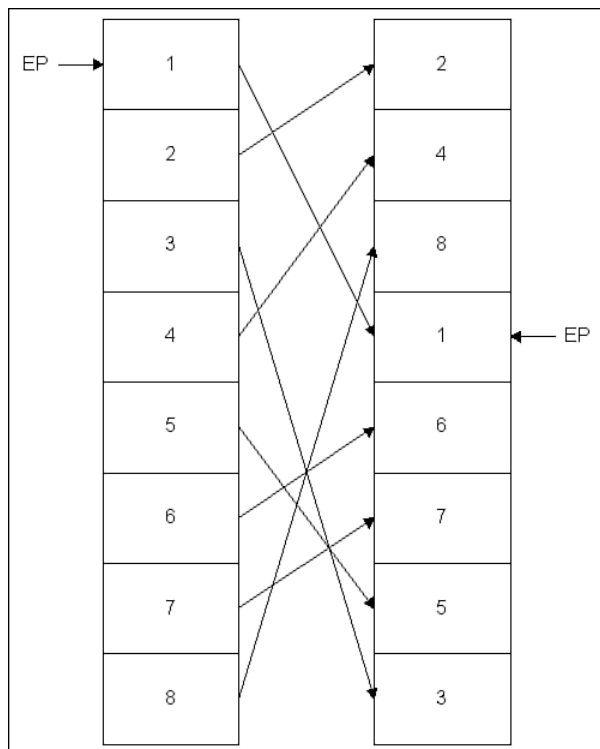


Figure 7.5 The Badboy virus uses eight modules.

The order of the subroutines will be different from generation to generation, which leads to $n!$ different virus generations, where n is the number of subroutines. BadBoy had eight subroutines, and $8!=40,320$ different generations. W32/Ghost (discovered in May 2000) has 10 functions, so $10!=3,628,800$ combinations. Both of them can be detected with search strings, but some scanners need to deal with such a virus algorithmically.

Two different variants of the W95/Zmorph virus appeared in January of 2000. The polymorphic engine of the virus implements a build-and-execute code evolution. The virus rebuilds itself on the stack with push instructions. Blocks of code decrypt the virus instruction-by-instruction and push the decrypted instructions to the stack. The build routine of the virus is already metamorphic. The engine supports jump insertion and removal between any instructions of the build code. Regardless, code emulators can be used to deal easily with such viruses. A constant code area of the virus is useful for identification because the virus body is decrypted on the stack.

7.6 Metamorphic Viruses

7.6.3 More Complex Metamorphic Viruses and Permutation Techniques

The W32/Evol virus appeared in July of 2000. The virus implements a metamorphic engine and can run on any major Win32 platform. In Listing 7.11, section *a.* shows a sample code fragment, mutated in *b.* to a new form in a new generation of the same virus. Even the "magic" DWORD values (5500000Fh, 5151EC8Bh) are changed in subsequent generations of the virus, as shown in *c.* Therefore any wildcard strings based on these values will not detect anything above the third generation of the virus. W32/Evol's engine is capable of inserting garbage between core instructions.

Listing 7.11

Different Generations of the W32/Evol Virus

a. An early generation:

```
C7060F000055      mov     dword ptr [esi],5500000Fh
C746048BEC5151    mov     dword ptr [esi+0004],5151EC8Bh
```

b. And one of its later generations:

```
BF0F000055      mov     edi,5500000Fh
893E             mov     [esi],edi
5F              pop     edi
52              push   edx
B640            mov     dh,40
BA8BEC5151     mov     edx,5151EC8Bh
53              push   ebx
8BDA            mov     ebx,edx
895E04          mov     [esi+0004],ebx
```

c. And yet another generation with recalculated ("encrypted") "constant" data:

```
BB0F000055      mov     ebx,5500000Fh
891E             mov     [esi],ebx
5B              pop     ebx
51              push   ecx
B9CB00C05F     mov     ecx,5FC000CBh
81C1C0EB91F1   add     ecx,F191EBC0h ; ecx=5151EC8Bh
894E04          mov     [esi+0004],ecx
```

Chapter 7—Advanced Code Evolution Techniques and Computer Virus Generator Kits

Variants of the W95/Zperm family appeared in June and September of 2000. The method used is known from the Ply DOS virus. The virus inserts jump instructions into its code. The jumps will be inserted to point to a new instruction of the virus. The virus body is built in a 64K buffer that is originally filled with zeros. The virus does not use any decryption. In fact, it will not regenerate a constant virus body anywhere. Instead, it creates new mutations by the removal and addition of jump instructions and garbage instructions. Thus there is no way to detect the virus with search strings in the files or in the memory.

Most polymorphic viruses decrypt themselves to a single constant virus body in memory. Metamorphic viruses, however, do not. Therefore the detection of the virus code in memory needs to be algorithmic because the virus body does not become constant even there. Figure 7.6 explains the code structure changes of Zperm-like viruses.

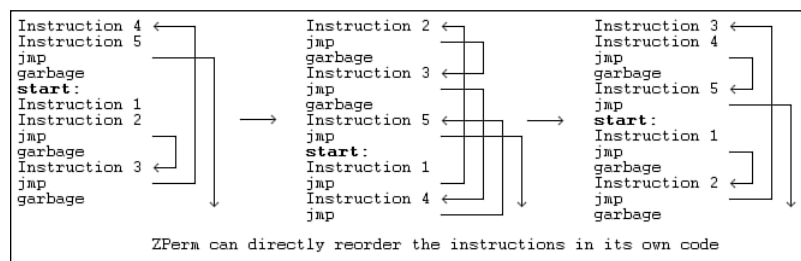


Figure 7.6 The Zperm virus.

Sometimes the virus replaces instructions with other, equivalent instructions. For example, the instruction `xor eax, eax` (which sets the `eax` register to zero) will be replaced by `sub eax, eax` (which also zeroes the contents of the `eax` register). The opcode of these two instructions will be different.

The core instruction set of the virus has the very same execution order; however, the jumps are inserted in random places. The B variant of the virus also uses garbage instruction insertion and removal such as `nop` (a do-nothing instruction). It is easy to see that the number of generations can be at least $n!$, where n is the number of core set instructions in the virus body.

Zperm introduced the real permutating engine (RPME). RPME is available for other virus writers to create new metamorphic viruses. We should note here that permutation is only a single item on the list of metamorphic techniques. To make

7.6 Metamorphic Viruses

the virus truly metamorphic, instruction opcode changes are introduced. Encryption can be used in combination with antiemulation and polymorphic techniques.

In October 2000, two virus writers created a new permutation virus, W95/Bistro, based on the sources of the Zperm virus and the RPME. To further complicate the matter, the virus uses a random code block insertion engine. A randomly activated routine builds a do-nothing code block at the entry point of the virus body before any active virus instructions. When executed, the code block can generate millions of iterations to challenge a code emulator's speed.

Simple permutating viruses and complex metamorphic viruses can be very different in their complexity of implementation. In any case, both permutating viruses and metamorphic viruses are different from traditional polymorphic techniques.

In the case of polymorphic viruses, there is a particular moment when we can take a snapshot of the completely decrypted virus body, as illustrated Figure 7.7. Typically, antivirus software uses a generic decryption engine (based on code emulation) to abstract this process. It is not a requirement to have a complete snapshot to provide identification in a virus scanner, but it is essential to find a particular moment during the execution of virus code when a complete snapshot can be made—to classify a virus as a traditional polymorphic virus. It is efficient to have a partial result, as long as there is a long-enough decrypted area of each possible generation of the virus.

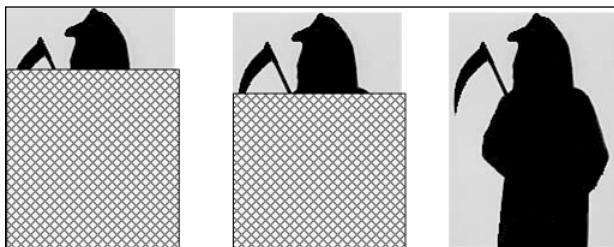


Figure 7.7 Snapshot of a decrypted "polymorphic virus."

On the contrary, a complex metamorphic virus does not provide this particular moment during its execution cycle. This is true even if the virus uses metamorphic techniques combined with traditional polymorphic techniques.

Chapter 7—Advanced Code Evolution Techniques and Computer Virus Generator Kits

7.6.4 Mutating Other Applications: The Ultimate Virus Generator?

Not only does W95/Bistro itself mutate in new generations, it also mutates the code of its host by a randomly executed code-morphing routine. In this way, the virus might generate new worms and viruses. Moreover, the virus cannot be perfectly repaired because the entry-point code area of the application could be different. The code sequence at the entry point of the host application is mutated for a range 480 bytes long. The next listing shows an original and a permuted code sequence of a typical entry point code:

Original entry-point code:

```

55          push   ebp
8BEC         mov    ebp, esp
8B7608      mov    esi, dword ptr [ebp + 08]
85F6        test   esi, esi
743B        je     401045
8B7E0C      mov    edi, dword ptr [ebp + 0c]
09FF        or     edi, edi
7434        je     401045
31D2        xor    edx, edx

```

Permuted entry-point code:

```

55          push   ebp
54          push   esp
5D          pop    ebp
8B7608      mov    esi, dword ptr [ebp + 08]
09F6        or     esi, esi
743B        je     401045
8B7E0C      mov    edi, dword ptr [ebp + 0c]
85FF        test   edi, edi
7434        je     401045
28D2        sub    edx, edx

```

Thus an instruction such as `test esi, esi` can be replaced by `or esi, esi`, its equivalent format. A `push ebp; mov ebp, esp` sequence (very common in high-level language applications) can be permuted to `push ebp; push esp, pop ebp`. It would certainly be more complicated to replace the code with different opcode

7.6 Metamorphic Viruses

sizes, but it would be possible to shorten longer forms of some of the complex instructions and include do-nothing code as a filler. This is a problem for all scanners.

If a virus or a 32-bit worm were to implement a similar morphing technique, the problem could be major. New mutations of old viruses and worms would be morphed endlessly! Thus, a virtually endless number of not-yet-detectable viruses and worms would appear without any human intervention, leading to the ultimate virus generator.

Note

An even more advanced technique was developed in the W95/Zmist virus²², which is described in the following section.

At the end of 1999, the W32/Smorph Trojan was developed. It implements a semimetamorphic technique to install a backdoor in the system. The standalone executable is completely regenerated during the installation of the Trojan. The PE header is re-created and will include new section names and section sizes. The actual code at the entry point is metamorphically generated. The code allocates memory and then decrypts its own resources, which contain a set of other executables. The Trojan uses API calls to its own import address table, which is filled with many nonessential API imports, as well as some essential ones. Thus everything in the standalone Trojan code will be different in new generations.

7.6.5 Advanced Metamorphic Viruses: Zmist

During *Virus Bulletin 2000*, Dave Chess and Steve White of IBM demonstrated their research results on undetectable viruses. Shortly after, the Russian virus writer, Zombie, released his *Total Zombification* magazine, with a set of articles and viruses of his own. One of the articles in the magazine was titled “Undetectable Virus Technology.”

Zombie has already demonstrated his set of polymorphic and metamorphic virus-writing skills. His viruses have been distributed for years in source format, and other virus writers have modified them to create new variants. Certainly this is the case with Zombie’s “masterpiece” creation, W95/Zmist.

Many of us have not seen a virus approach this complexity for some time. We could easily call Zmist one of the most complex binary viruses ever written. W95/SK, One_Half, ACG, and a few other virus names popped into our minds for

Chapter 7—Advanced Code Evolution Techniques and Computer Virus Generator Kits

comparison. Zmist is a little bit of everything: It is an entry-point obscuring (EPO) virus that is metamorphic. Moreover, the virus randomly uses an additional polymorphic decryptor.

The virus supports a unique new technique: code integration. The Mistfall engine contained in the virus is capable of decompiling PE files to their smallest elements, requiring 32MB of memory. Zmist will insert itself into the code; it moves code blocks out of the way, inserts itself, regenerates code and data references (including relocation information), and rebuilds the executable. This is something that has never been seen in any previous virus.

Zmist occasionally inserts jump instructions after every single instruction of the code section, each of which points to the next instruction. Amazingly, these horribly modified applications will still run as before, just as the infected executables do, from generation to generation. In fact, we have not seen a single crash during test replications. Nobody expected this to work—not even the virus's author Zombie. Although it is not foolproof, it seems to be good enough for a virus. It takes some time for a human to find the virus in infected files. Because of this extreme camouflage, Zmist is easily the perfect antiheuristic virus.

They say a good picture is worth a thousand words. The T-1000 model from the film *Terminator 2* is the easiest analogy to use. Zmist integrates itself into the code section of the infected application as the T-1000 model could hide itself on the floor.

7.6.5.1 Initialization

Zmist does not alter the entry point of the host. Instead, it merges with the existing code, becoming part of the instruction flow. However, the code's random location means that sometimes the virus will never receive control. If the virus does run, it will immediately launch the host as a separate process and hide the original process (if the `RegisterServiceProcess()` function is supported on the current platform) until the infection routine completes. Meanwhile, the virus will begin searching for files to infect.

7.6.5.2 Direct Action Infection

After launching the host process, the virus will check whether there are at least 16MB of physical memory installed. The virus also checks that it is not running in console mode. If these checks pass, it will allocate several memory blocks (including a 32MB area for the Mistfall workspace), permute the virus body, and begin a recursive search for PE files. This search will take place in the Windows directory and all subdirectories, the directories referred to by the PATH environment

7.6 Metamorphic Viruses

variable, and then all fixed or remote drives from A: to Z:. This is a rather brute-force approach to spreading.

7.6.5.3 Permutation

The permutation is fairly slow because it is done only once per infection of a machine. It consists of instruction replacement, such as the reversing of branch conditions, register moves replaced by push/pop sequences, alternative opcode encoding, XOR/SUB and OR/TEST interchanging, and garbage instruction generation. It is the same engine, RPME, that is used in several viruses, including W95/Zperm, which also was written by Zombie.

7.6.5.4 Infection of Portable Executable Files

A file is considered infectable if it meets the following conditions:

- It is smaller than 448KB.
- It begins with *MZ*. (Windows does not support ZM-format Windows applications.)
- It is not infected already. (The infection marker is Z at offset 0x1C in the MZ header, a field generally not used by Windows applications.)
- It is a PE file.

The virus will read the entire file into memory and then choose one of three possible infection types. With a one-in-ten chance, only jump instructions will be inserted between every existing instruction (if the instruction was not a jump already), and the file will not be infected. There is also a one in ten chance that the file will be infected by an unencrypted copy of the virus. Otherwise, the file will be infected by a polymorphically encrypted copy of the virus. The infection process is protected by structured exception handling, which prevents crashes if errors occur. After the rebuilding of the executable, the original file is deleted, and the infected file is created in its place. However, if an error occurs during the file creation, the original file is lost, and nothing will replace it.

The polymorphic decryptor consists of “islands” of code that are integrated into random locations throughout the host code section and linked by jumps. The decryptor integration is performed in the same way as for the virus body integration—existing instructions are moved to either side, and a block of code is placed between them. The polymorphic decryptor uses absolute references to the data section, but the Mistfall engine will update the relocation information for these references, too. An antiheuristic trick is used for decrypting the virus code:

Chapter 7—Advanced Code Evolution Techniques and Computer Virus Generator Kits

Instead of making the section writeable to alter its code directly, the host is required to have, as one of the first three sections, a section containing writeable, initialized data. The virtual size of this section is increased by 32KB, large enough for the decrypted body and all variables used during decryption. This allows the virus to decrypt code directly into the data section and transfer control there.

If such a section cannot be found, the virus will infect the file without using encryption. The decryptor receives control in one of four ways:

- Via an absolute indirect call (0xFF 0x15)
- Via a relative call (0xE8)
- Via a relative jump (0xE9)
- As part of the instruction flow itself

If one of the first three methods is used, the transfer of control will appear soon after the entry point. In the case of the last method, though, an island of the decryptor is simply inserted into the middle of a subroutine somewhere in the code (including before the entry point). All used registers are preserved before decryption and restored afterward, so the original code will behave as before. Zombie calls this last method *UEP*, perhaps an acronym for “unknown entry point” because there is no direct pointer anywhere in the file to the decryptor.

When encryption is used, the code is encrypted with ADD, SUB, or XOR with a random key, which is altered on each iteration by ADD/SUB/XOR with a second random key. Between the decryption instructions are various garbage instructions. They use a random number of registers and a random choice of loop instruction, all produced by the executable trash generator (ETG) engine, which was also written by Zombie. Randomness features heavily in this virus.

7.6.5.5 Code Integration

The integration algorithm requires that the host have fixups to distinguish between offsets and constants. After infection, however, the fixup data is not required by the virus. Therefore, though it is tempting to look for a gap of about 20KB in the fixup area (which would suggest that the virus body is located there), it would be dangerous to rely on this during scanning.

If another application (such as one of an increasing number of viruses) were to remove the fixup data, the infection would be hidden. The algorithm also requires that the name of each section in the host be one of the following: CODE, DATA, AUTO, BSS, TLS, .bss, .tls, .CRT, .INIT, .text, .data, .src, .reloc, .idata, .rdata, .edata, .debug, or DGROUP. These section names are produced by the most com-

7.6 Metamorphic Viruses

mon compilers and assemblers in use: those of Microsoft, Borland, and Watcom. The names are not visible in the virus code because the strings are encrypted.

A block of memory is allocated that is equivalent to the size of the host memory image, and each section is loaded into this array at the section's relative virtual address. The location of every interesting virtual address is noted (import and export functions, resources, fixup destinations, and the entry point), and then the instruction parsing begins.

This is used to rebuild the executable. When an instruction is inserted into the code, all following code and data references must be updated. Some of these references might be branch destinations, and in some cases the size of these branches will increase as a result of the modification. When this occurs, more code and data references must be updated, some of which might be branch destinations, and the cycle repeats. Fortunately—at least from Zombie's point of view—this regression is not infinite; although a significant number of changes might be required, the number is limited. The instruction parsing consists of identifying the type and length of each instruction. Flags are used to describe the types, such as instruction is an absolute offset requiring a fixup entry, instruction is a code reference, and so on.

In some cases, an instruction cannot be resolved in an unambiguous manner to either code or data. In that case, Zmist will not infect the file. After the parsing stage is completed, the mutation engine is called, which inserts the jump instructions after every instruction or generates a decryptor and inserts the islands into the file. Then the file is rebuilt, the relocation information is updated, the offsets are recalculated, and the file checksum is restored. If overlay data is appended to the original file, then it is copied to the new file too.

7.6.6 (W32, Linux)/Simile: A Metamorphic Engine Across Systems

W32/Simile is the latest "product" of the developments in metamorphic virus code. The virus was released in the 29A #6 issue in early March 2002. It was written by the virus writer who calls himself The Mental Driller. Some of his previous viruses, such as W95/Drill (which used the Tuareg polymorphic engine), were already very challenging to detect.

W32/Simile moves yet another step in complexity. The source code of the virus is approximately 14,000 lines of Assembly code. About 90% of the virus code is spent on the metamorphic engine itself, which is extremely powerful. The virus's author has called it *MetaPHOR*, which stands for "metamorphic permutating high-obfuscating reassembler."

The first-generation virus code is about 32KB, and there are three known variants of the virus in circulation. Samples of the variant initially released in the 29A

Chapter 7—Advanced Code Evolution Techniques and Computer Virus Generator Kits

issue have been received by certain AV companies from major corporations in Spain, suggesting a minor outbreak.

W32/Simile is very obfuscated and challenging to understand. The virus attacks disassembling, debugging, and emulation techniques, as well as standard evaluating-based virus-analysis techniques. As with many other complex viruses, Simile uses EPO techniques.

7.6.6.1 Replication Routine

Simile contains a fairly basic direct-action replication mechanism that attacks PE files on the local machine and the network. The emphasis is clearly on the metamorphic engine, which is unusually complex.

7.6.6.2 EPO Mechanism

The virus searches for and replaces all possible patterns of certain call instructions (those that reference `ExitProcess()` API calls) to point to the beginning of the virus code. Thus the main entry point is not altered. The metamorphic virus body is sometimes placed, together with a polymorphic decryptor, into the same location of the file. In other cases, the polymorphic decryptor is placed at the end of the code section, and the virus body is not placed there but in another section. This is to further obfuscate the location of the virus body.

7.6.6.3 Polymorphic Decryptor

During the execution of an infected program, when the instruction flow reaches one of the hooks that the virus places in the code section, control is transferred to a polymorphic decryptor responsible for decoding the virus body (or simply copying it directly, given that the virus body is intentionally not always encrypted).

This decryptor, whose location in the file is variable, first allocates a large chunk of memory (about 3.5MB) and then proceeds to decipher the encrypted body into it. It does this in a most unusual manner: Rather than going linearly through the encrypted data, it processes it in a seemingly random order, thus avoiding the triggering of some decryption-loop recognition heuristics. This “pseudo-random index decryption” (as the virus writer calls it) relies on the use of a family of functions that have interesting arithmetic properties, modulo 2^n . Although the virus writer discovered this by trial and error, it is in fact possible to give a mathematical proof that his algorithm will work in all cases (provided the implementation is correct, of course). This proof was made by Frederic Perriot at Symantec.

The size and appearance of the decryptor varies greatly from one virus sample to the next. To achieve this high variability, the virus writer simply generates a

7.6 Metamorphic Viruses

code template and then puts his metamorphic engine to work to transform the template into a working decryptor.

Additionally, in some cases the decryptor might start with a header whose intent is not immediately obvious upon reading it. Further study reveals that its purpose is to generate antiemulation code on the fly. The virus constructs a small oligomorphic code snippet containing the instruction RDTSC (ReaD Time Stamp Counter), which retrieves the current value of an internal processor ticks counter. Then, based on one random bit of this value, the decryptor either decodes and executes the virus body or bypasses the decryption logic altogether and simply exits.

Besides confusing emulators that would not support the somewhat peculiar RDTSC instruction (one of Mental Driller's favorites, already present in W95/Drill), this is also a very strong attack against all algorithms relying on emulation either to decrypt the virus body or to determine viral behavior heuristically—because it effectively causes some virus samples to cease infecting completely upon a random time condition.

When the body of the virus is first executed, it retrieves the addresses of 20 APIs that it requires for replication and for displaying the payload. Then it will check the system date to see whether either of its payloads should activate. Both payloads require that the host import functions from USER32.DLL. If this is the case, the virus checks whether or not it should call the payload routine (which is explained further on).

7.6.6.4 Metamorphism

After the payload check has been completed, a new virus body is generated. This code generation is done in several steps:

1. Disassemble the viral code into an intermediate form, which is independent of the CPU on which the native code executes. This allows for future extensions, such as producing code for different operating systems or even different CPUs.
2. Shrink the intermediate form by removing redundant and unused instructions. These instructions were added by earlier replications to interfere with disassembly by virus researchers.
3. Permutate the intermediate form, such as reordering subroutines or separating blocks of code and linking them with jump instructions.
4. Expand the code by adding redundant and unused instructions.
5. Reassemble the intermediate form into a final, native form that will be added to infected files.

Chapter 7—Advanced Code Evolution Techniques and Computer Virus Generator Kits

Not only can Simile expand as most first-generation metamorphic viruses did, it can also shrink (and shrink to different forms). Simile.D is capable of translating itself to different metamorphic forms ($V_1, V_2 \dots V_n$) and does so on more than one operating system ($O_1, O_2 \dots O_n$). So far, the virus does not use multiple CPU forms, but it could also introduce code translation and metamorphism for different processors ($P_1 \dots P_n$) in the future, as shown in Figure 7.8.

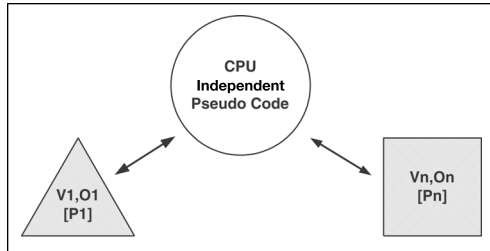


Figure 7.8 Stages of Simile from Linux to Windows and from one generation to another.

7.6.6.5 Replication

The replication phase begins next. It begins by searching for *.exe in the current directory and then on all fixed and mapped network drives. The infection will scan recursively into directories, but only to a depth of three subdirectories, avoiding completely any directory that begins with the letter *W*. For each file that is found, there is a 50% chance that it will be skipped. Additionally, files will be skipped if they begin with *F-*, *PA*, *SC*, *DR*, or *NO* or contain the letter *V* anywhere in the name. Due to the nature of the comparison, other character combinations are unintentionally skipped, such as any directory that begins with the number *7*, any file that begins with *FM*, or any file that contains the number *6* anywhere in the name.

The file-infection routine contains many checks to filter files that cannot be infected safely. These checks include such conditions as the file must contain a checksum, it must be an executable for the Intel 386+ platform, and there must exist sections whose names are .text or CODE and .data or DATA. The virus also checks that the host imports some kernel functions, such as ExitProcess.

For any file that is considered infectable, random factors and the file structure will determine where the virus places its decryptor and virus body. If the file contains no relocations the virus body will be appended to the last section in the file.

7.6 Metamorphic Viruses

(Apparently, this can happen randomly with a small chance as well, regardless if there are any relocations in the file.)

In this case, the decryptor will be placed either immediately before the virus body or at the end of the code section. Otherwise, if the name of the last section is `.reloc`, the virus will insert itself at the beginning of the data section, move all of the following data, and update all of the offsets in the file.

7.6.6.6 Payload

The first payload activates only during March, June, September, or December. Variants A and B of W32/Simile display their messages on the 17th of these months. Variant C will display its message on the 18th of these months. Variant A will display “Metaphor v1 by The Mental Driller/29A,” and variant B will display “Metaphor 1b by The Mental Driller/29A.” Variant C intends to display “Deutsche Telekom by Energy 2002 **g**”; however, the author of that variant had little understanding of the code, and the message rarely appears correctly. In all cases, the cases of the letters are mixed randomly, as shown in Figure 7.9.

The second payload activates on the 14th of May in variants A and B, and on the 14th of July in variant C. Variants A and B will display “Free Palestine!” on computers that use the Hebrew locale. Variant C contains the text “Heavy Good Code!” but due to a bug this is displayed only on systems on which the locale cannot be determined.

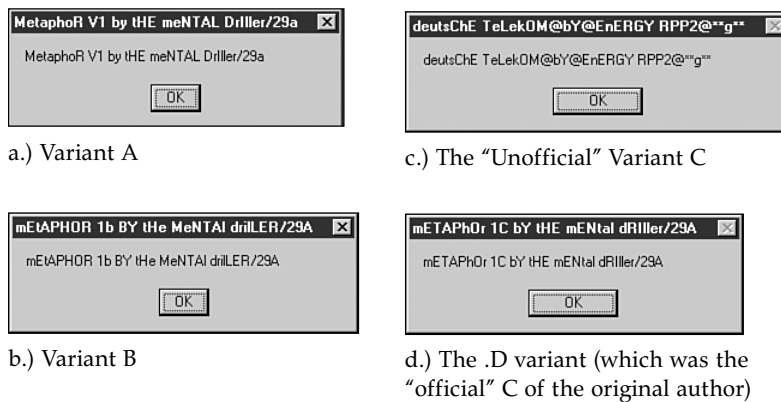


Figure 7.9 The “metamorphic” activation routines of the Simile virus.

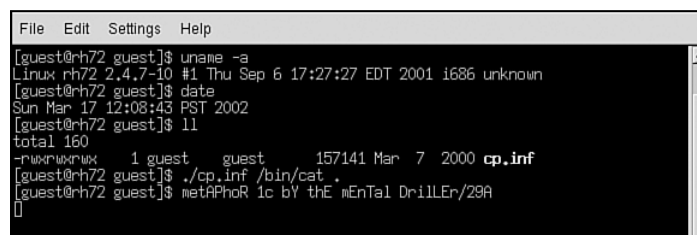
Chapter 7—Advanced Code Evolution Techniques and Computer Virus Generator Kits

The first W32/Linux cross-infector, {W32, Linux}/Peelf, uses two separate routines to carry out the infection on PE and ELF files. Simile.D shares a substantial amount of code between the two infection functions, such as the polymorphic and metamorphic engines. The only platform-specific part in the infection routine is the directory traversal code and the API usage.

The virus was confirmed to infect successfully under versions 6.2, 7.0, and 7.2 of Red Hat Linux, and it very likely works on most other common Linux distributions.

Infected files will grow by about 110KB on average, but the size increase is variable due to the shrinking and expansion capability of the metamorphic engine and to the insertion method.

When the .D variant activates on Linux, it simply prints a console message, as shown in Figure 7.10.



```

File Edit Settings Help
[guest@rh72 guest]$ uname -a
Linux rh72 2.4.7-10 #1 Thu Sep 6 17:27:27 EDT 2001 i686 unknown
[guest@rh72 guest]$ date
Sun Mar 17 12:08:43 PST 2002
[guest@rh72 guest]$ ll
total 160
-rwxrwxrwx 1 guest guest 157141 Mar 7 2000 cp.inf
[guest@rh72 guest]$ ./cp.inf /bin/cat .
[guest@rh72 guest]$ metAPhOr Ic bY thE mEnTAl DrILLer/29A

```

Figure 7.10 The payload of the Simile.D virus on Linux.

7.6.7 The Dark Future—MSIL Metamorphic Viruses

As discussed in this chapter, some new MSIL viruses, such as MSIL/Gastropod, already support semi-metamorphic (permutating) code generation under the .NET Framework. Such viruses have a great advantage because they do not need to carry their own source code. Instead, viruses can simply decompile themselves to generate new binaries, for example, by using the System.Reflection.Emit namespace. Listing 7.12 illustrates the MSIL metamorphic engine of the MSIL/Gastropod virus in two generations.

Listing 7.12

Different Generations of the MSIL/Gastropod Virus

a.)

```

.method private static hidebysig specialname void .cctor()
{

```

7.6 Metamorphic Viruses

```
ldstr "[.NET.Snail - sample CLR virus (c) whale 2004 ]"
stsfld class System.String Ylojnc.lgxmAxA::Wac1NvK
nop
ldc.i4.6
ldc.i4.s 0xF
call int32 [mscorlib]System.Environment::get_TickCount()
nop
newobj void nljvKpqb::.ctor(int32 len1, int32 len2, int32 seed)
stsfld class nljvKpqb Ylojnc.lgxmAxA::XxnArefPizsour
call int32 [mscorlib]System.Environment::get_TickCount()
nop
newobj void [mscorlib]System.Random::.ctor(int32)
stsfld class [mscorlib]System.Random Ylojnc.lgxmAxA::aajqebjtoBxjf
ret
}
```

b.)

```
.method private static hidebysig specialname void .ctor()
{
ldstr "[.NET.Snail - sample CLR virus (c) whale 2004 ]"
stsfld class System.String kivAklozuas.ghqrRr1v::ngnMTzqo
ldc.i4.6
ldc.i4.s 0xF
call int32 [mscorlib]System.Environment::get_TickCount()
newobj void xiWtNaocl::.ctor(int32 len1, int32 len2, int32 seed)
stsfld class xiWtNaocl kivAklozuas.ghqrRr1v::yXuzlmssjpp
call int32 [mscorlib]System.Environment::get_TickCount()
newobj void [mscorlib]System.Random::.ctor(int32)
stsfld // line continues on next line
class [mscorlib]System.Random kivAklozuas.ghqrRr1v::kaokaufdiehjs
nop
ret
}
```

The extracted snippets represent the class constructor (".ctor") function of MSIL/Gastropod in two different generations. The semi-metamorphic engine of the virus obfuscates class names and method names²³. In addition, the permutation engine also inserts and removes junk instructions (such as nop) into its body. Indeed, it is not well known to MSIL developers that they can invoke a compiler and generate code and assembly from a running application, but virus writers already use this feature.

MSIL/Gastropod is a code-builder virus: It reconstructs its own host program with itself. This method allows the virus body to be placed to an unpredictable

Chapter 7—Advanced Code Evolution Techniques and Computer Virus Generator Kits

position within the host program. The main entry-point method of the host program is replaced by the entry-point method of the virus. When the infected host is executed, the virus code is invoked, which will run the original main entry-point method eventually.

In addition, some viruses do not rely on the use of .NET Framework namespace to implement parasitic MSIL infection. For example, the MSIL/Impanate virus, written by roy g. biv, is fully aware of both 32-bit and 64-bit MSIL files and infects them using the EPO (Entry-Point Obscuring) technique. Thus next generation MSIL metamorphic viruses might not rely on the use of System.Reflection.Emit and similar namespaces anymore.

7.7 Virus Construction Kits

Virus writers continuously try to simplify the creation of virus code. Because most viruses were written in Assembly language, writing them remained out of reach for many kids. This inspired virus writers to create generators that can be used by just about anyone who can use a computer.

Virus construction kits evolved over the years as viruses targeted new platforms. Construction kits and virus mutators were built to generate a wide variety of malicious code, including DOS COM and EXE viruses; 16-bit Windows executable viruses; BATCH and Visual Basic Script viruses; Word, PowerPoint, and Excel viruses; mIRC worms; and so on. Recently, even PE virus code can be generated by such kits.

Virus construction kits are a major concern of antivirus producers. It is impossible to predict whether or not virus writers will use a particular kit. Thus the virus researchers need to spend time with even the most primitive kits to see what kinds of viruses they can produce and to design detection and repair capabilities against all possible cases. To add to this problem, many kits generate source code instead of binary. Novice attackers can change the source code further, beyond the capabilities of the kit itself, so it is not always possible to develop perfectly adequate protection.

To make the job of antivirus scanners more difficult, kits deploy armoring techniques such as encryption, antidebugging, antiemulation, and anti-behavior blocking techniques. Furthermore, some kits can mutate the code of the viruses similarly to metamorphic techniques.

7.7 Virus Construction Kits

7.7.1 VCS (Virus Construction Set)

The first virus generator was VCS, written in 1990. The authors of the kit were members of the Verband Deutscher Virenliebhaber (the Association of German Virus Lovers).

VCS was a rather simple kit. All viruses that the kit can generate are 1,077 bytes and saved to VIRUS.COM on the disk. The user's only options are to specify the name of a text file with a message and to set a generation number to display that message. The VCS viruses can only infect DOS COM files. The payload of the virus is to kill AUTOEXEC.BAT and CONFIG.SYS files and display the user's message.

VCS viruses are simple but encrypted. The only remarkable feature of VCS viruses is that they can check whether the FluShot behavior blocker is loaded in memory and avoid infecting if it is.

7.7.2 GenVir

In 1990–1991, a kit called GenVir was released as a shareware in France by J. Struss. The original intention was to “test” antivirus product capabilities with a tool that could generate newly replicating viruses. Very few viruses were ever created with GenVir. In 1993, a new version of GenVir was released to support newer versions of DOS.

7.7.3 VCL (Virus Creation Laboratory)

In 1992, the VCL virus construction kit was written in the U.S. (see Figure 7.11). The author of this kit was a member of the NuKE virus-writer team who called himself Nowhere Man.

VCL looked rather advanced at the time because it supported a Borland C++ 3.0–based IDE with mouse support. Instead of creating binary files, VCL creates Assembly source code of viruses. These sources need to be compiled and linked afterwards by the attacker to make them live viruses. VCL supports a large selection of payloads, as well as encryption and various infection strategies. Not surprisingly, not all viruses that VCL generates are functional. Nevertheless, the viruses look rather different because of the generator options, and thus their detection is far less straightforward than VCS. Several VCL viruses managed to become widespread. For the first time, VCL indicated that even beginner attackers could manage to cause problems in the field.

Chapter 7—Advanced Code Evolution Techniques and Computer Virus Generator Kits

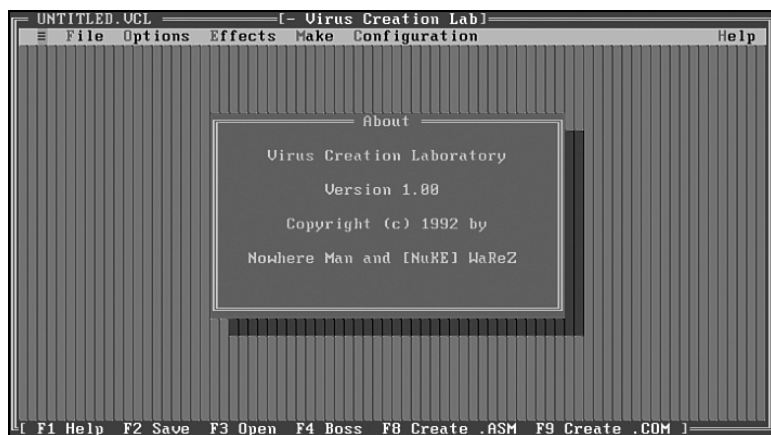


Figure 7.11 The GUI of the Virus Creation Laboratory.

7.7.4 PS-MPC (Phalcon-Skism Mass-Produced Code Generator)

Because virus-writing groups are in competition, we did not have to wait long before another group announced its kit. PS-MPC was created in 1992 in the United States, by the virus writer, Dark Angel.

PS-MPC does not have the fancy user interface of VCL, which actually makes it a much more dangerous kit. Because PS-MPC is script-driven, it can be much simpler to generate hundreds of copycat viruses. PS-MPC is also a source generator, just like VCL. However, the PS-MPC viruses are much more functional than VCL's. Not surprisingly, virus writers utilized this particular kit to generate more than 15,000 PS-MPC variants and upload them to several antivirus companies' FTP sites.

Although the initial versions of PS-MPC could only create direct-action viruses, other versions were released that could create memory resident viruses as well. Furthermore, later versions support infections of EXE files also.

PS-MPC is one of the reasons I decided to introduce kits in this chapter. The creator of PS-MPC realized that to be successful, generators must create different-looking viruses. To achieve this, PS-MPC uses a generator that effectively works as a code-morphing engine. As a result, the viruses that PS-MPC generates are not polymorphic, but their decryption routines and structures change in variants.

PS-MPC was quickly followed by G2, a second-generation kit. G2 adds anti-debug/antiemulation features to PS-MPC, with even more decryptor morphing.

7.7 Virus Construction Kits

7.7.5 NGVCK (Next Generation Virus Creation Kit)

NGVCK was introduced in 2001 by the virus writer, SnakeByte. NGVCK (see Figure 7.12) is a Win32 application written in Visual Basic. This kit is very similar to VCL in many ways, but it can generate 32-bit viruses that infect PE files. There are well over 30 versions of NGVCK alone.

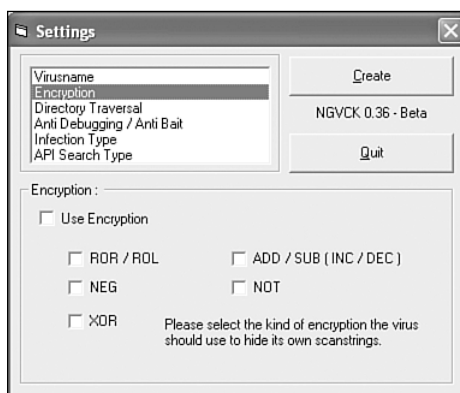


Figure 7.12 The main menu of NGVCK.

NGVCK has a rather advanced Assembly source-morphing engine. The viruses that it creates have random function ordering, junk code insertion, and user-defined encryption methods. It also attacks the SoftICE debugger on Windows 95 systems. Later versions of the kit also have support to infect files using different techniques.

The viruses that NGVCK creates are automatically morphed, so every single time somebody uses the kit, a new minor variant of the virus is generated. The code generation uses principles described in Chapter 6, “Basic Self-Protection Strategies.” Unfortunately, it is much simpler to develop source morphing than real code-morphing engines, so such techniques are likely to be widely adopted by the attackers in the future.

7.7.6 Other Kits and Mutators

Several other copycat generators have been used by amateur virus writers. Amateurs have many choices—well over 150 kits and code mutators are available, and many of them have been used to produce working viruses. In 1996 such tools became extremely popular. In particular, we have received new viruses from

Chapter 7—Advanced Code Evolution Techniques and Computer Virus Generator Kits

English schools that were generated by the IVP (Instant Virus Production Kit). IVP was written by Admiral Bailey of the YAM (Youngsters Against McAfee) group in Turbo Pascal. It supports EXE and COM infection, as well as encryption and code mutation—and kids loved it because of its Trojan payloads.

One of the most infamous examples of a virus generator-based attack was known to the general public as the “Anna Kornikova” virus outbreak. This worm was created by the VBSWG kit, and the kit’s user, a 20-year-old Dutch man, admittedly did not know how to write a program. Nevertheless, the mass-mailing ability of the VBS worm, combined with social engineering, worked effectively. Many users were eager to see a recent picture of Anna Kornikova—but executed the script instead.

Table 7.1 lists some other common generators.

Table 7.1

Examples of Virus Generator Kits

Name of Generator Kit	Description
NRLG (NuKE’s Randomic Life Generator)	Released in 1994 by the virus writer Azrael. Very similar to VCL.
OMVCK (Odysseus Macro Virus Construction Kit)	This kit was released in early 1998. It can generate Word Basic macro-virus source code.
SSIWG (Senna Spy Internet Worm Generator)	Released in 2000 in Brazil. This generator supports the creation of VBS worms.
NEG (NoMercy Excel Generator)	This was the first Excel macro virus generator kit (1998). It creates .bas files.
VBSWG (VBS Worm Generator)	Released in 2000 by [K]Alamar. Generates various script worms.
AMG (Access Macro Generator)	Created in 1998 by the virus writer Ultras to generate Access97 macro viruses.
DREG (Digital Hackers’ Alliance Randomized Encryption Generator)	Released in 1997 by Gothmog. Supports advanced code morphing and antiheuristics.

Evidently, in the future we can expect the tendency of virus construction kits to continue toward networked viruses, worms in particular. There are already a few kits that can be driven via a CGI script over a Web interface. This allows the attacker to avoid releasing the complete code of the virus generator, allowing fewer opportunities for antivirus vendors to test the capabilities of such tools.

References

7.7.7 How to Test a Virus Construction Tool?

The use of virus construction kits is not purely a technical question; it is also an ethical question. As Alan Solomon has argued, it is ethical for the antivirus researcher to use kits as long as the samples are stored on a dirty PC only and destroyed as soon as the researcher has developed a detection solution for them. Thus the generated samples shall not be stored anywhere, not even for further testing. This method is the only approach widely accepted as ethical among computer virus researchers.

References

1. Fridrik Skulason, "Latest Trends in Polymorphism—The Evolution of Polymorphic Computer Viruses," *Virus Bulletin Conference*, 1995, pp. I-VII.
2. Peter Szor and Peter Ferrie, "Hunting for Metamorphic," *Virus Bulletin Conference*, September 2001, pp. 123-144.
3. Tim Waits, "Virus Construction Kits," *Virus Bulletin Conference*, 1993, pp. 111-118.
4. Fridrik Skulason, "Virus Encryption Techniques," *Virus Bulletin*, November 1990, pp. 13-16.
5. Peter Szor, "F-HARE," *Documentation by Sarah Gordon*, 1996.
6. Eugene Kaspersky, "Picturing Harrier," *Virus Bulletin*, September 1999, pp. 8-9.
7. Peter Szor, Peter Ferrie, and Frederic Perriot, "Striking Similarities," *Virus Bulletin*, May 2002, pp. 4-6.
8. X. Lai, J. L. Massey, "A Proposal for New Block Encryption Standard," *Advances in Cryptology Eurocrypt'90*, 1991.
9. Peter Szor, "Bad IDEA," *Virus Bulletin*, April 1998, pp. 18-19.
10. Peter Szor, "Tricky Relocations," *Virus Bulletin*, April 2001, page 8.
11. Dmitry Gryaznov, "Analyzing the Cheeba Virus," *EICAR Conference*, 1992, pp. 124-136.
12. Dr. Vesselin Bontchev, "Cryptographic and Cryptanalytic Methods Used in Computer Viruses and Anti-Virus Software," *RSA Conference*, 2004.
13. James Riordan and Bruce Schneider, "Environmental Key Generation Towards Clueless Agents," *Mobile Agents and Security*, Springer-Verlag, 1998, pp. 15-24.

Chapter 7—Advanced Code Evolution Techniques and Computer Virus Generator Kits

14. Makoto Matsumoto and Takuji Nishimura, "Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudorandom Number Generator," *ACM Transactions on Modeling and Computer Simulations: Special Issue on Uniform Random Number Generator*, 1998, <http://www.math.keio.ac.jp/~nisimura/random/doc/mt.pdf>.
15. Peter Szor, "The Road to MtE: Polymorphic Viruses," *Chip*, June 1993, pp. 57-59.
16. Fridrik Skulason, "1260—The Variable Virus," *Virus Bulletin*, March 1990, page 12.
17. Vesselin Bontchev, "MtE Detection Test," *Virus News International*, January 1993, pp. 26-34.
18. Peter Szor, "The Marburg Situation," *Virus Bulletin*, November 1998, pp. 8-10.
19. Dr. Igor Muttik, "Silicon Implants," *Virus Bulletin*, May 1997, pp. 8-10.
20. Vesselin Bontchev and Katrin Tocheva, "Macro and Script Virus Polymorphism," *Virus Bulletin Conference*, 2002, pp. 406-438.
21. "Shape Shifters," *Scientific American*, May 2001, pp. 20-21.
22. Peter Szor and Peter Ferrie, "Zmist Opportunities," *Virus Bulletin*, March 2001, pp. 6-7.
23. Peter Ferrie, personal communication, 2004.