FEATURE 1

ARE METAMORPHIC VIRUSES REALLY INVINCIBLE? PART 2

Arun Lakhotia, Aditya Kapoor, Eric Uday
University of Louisiana at Lafayette, USA

Metamorphic viruses thwart detection by signature-based (static) AV technologies by morphing their code as they propagate. The viruses can also thwart detection by emulation-based (dynamic) technologies. To do so they need to detect whether they are running in an emulator and change their behaviour. So, are metamorphic viruses really invincible?

In part one of this article (see *VB*, December 2004 p.5) we presented an overview of mutation engines, followed by a discussion of the Achilles' heel of a metamorphic virus: its need to analyse itself. In this part of the article we present a case study in which we look at the metamorphic engine of the virus W32/Evol. This leads to a discussion on developing 'reverse morphers' to undo the mutations performed by a mutation engine. The article closes with our conclusions.

W32/EVOL: A CASE STUDY

W32/Evol is a relatively simple metamorphic virus. Nonetheless, it is a good example for a case study since the virus demonstrates properties that are common to all metamorphic viruses, i.e. it obfuscates calls made to system libraries and it mutates its code prior to propagation.

The rest of this section describes the details of these methods.

OBFUSCATING SYSTEM CALLS

In order to perform a malicious act, a program must access the disk or the network. Access to these resources is controlled by the operating system. A quick way to determine whether a program is malicious is to look at the system calls it makes.

W32/Evol does not use a 'normal' procedure to make system calls – it obfuscates its calls, which means that a disassembler such as *IDAPro* cannot determine directly the system calls it makes.

W32/Evol uses the following strategies to obfuscate its calls:

1. It computes the address of the kernel32.dll function GetProcAddress() by searching for the eight-byte sequence [0x55 00 01 F2 51 51 ec 8b] on *Windows 2000*. (The W32/Evol binary at http://vx.netlux.org/

looks for the byte sequence [0x55 00 00 0f 51 51 ec 8b], which is probably for a different version of *Windows*.)

2. It keeps the address of GetProcAddress() in its stack-based global data store, maintained at a certain distance from a magic marker pushed on the stack.
3. It uses a 'return' instruction to make a call to GetProcAddress().
4. It maintains the names of functions to be called as immediate, double-word operands of multiple instructions, not as strings in the data store.

MUTATION ENGINE

The mutation engine of W32/Evol is a function consisting of the Disassembly and Transform modules described in part 1 of this article (see *VB*, December 2004, p.5). It does not have a Reverse Engineering module since it transforms one instruction at a time.

The mutation engine is located at address 00401FD7. The engine takes three inputs (all values quoted here are the values recorded during a test run of W32/Evol in a debugger):

1. The Relocatable Virtual Address (RVA) of loaded virus code: RVA = 401000.
2. The length of the original virus code: LEN = arg_4 (1847).
3. Pointer to buffer (BUF1) to store the transformed code: (Max Size buffer = 4 x LEN = arg_8 (7F0000).

The output of the engine is the transformed program, which is placed in the buffer BUF1.

DISASSEMBLY MODULE

The disassembly module of W32/Evol uses the linear sweep algorithm. It checks whether a byte starts an instruction, if it does then it gets the size of the instruction, and disassembles the byte following the instruction. If, during disassembly, the program comes across a byte that is not an instruction, the mutation process is abandoned (see Figure 3).

The mutation engine processes only a limited range of opcodes of the x86 instruction set. For instance, it does not process floating-point instructions. The mutation is abandoned if an instruction outside its accepted range is encountered.

Figure 4 shows the code fragment from W32/Evol performing the instruction range check.

Location	Instruction
0040227A	cmp al, 0FEh
0040227C	jz short loc_402282 ; If the byte under analysis is FE ; goto 00402282
0040227E	cmp al, 0FFh ; If the byte is FF goto 00402282
00402280	jnz short loc_4022B5 ; compare al with next opcode.
00402282	mov al, [esi+1] ; If byte is either 0xFE or 0xFFload ModR/M ; byte in al
00402285	and al, 38h
00402287	ror al, 3
0040228A	cmp al, 7
0040228C	jz loc_402532 ; If value of bits 3, 4, 5 of ModR/M byte are ; 1 the instruction does not exist ; Exit mutation process

Figure 3. Invalid instruction check.

Location	Instruction
00402118	cmp al, 0Fh ; Checking for two-byte opcode.
0040211A	jnz short loc_402152 ;compare al with next opcode.
0040211C	mov cl, [esi+1]
0040211F	cmp cl, 80h
00402122	jb loc_402532 ; If byte following 0x0F is less than 0x80 ; then exit mutation process
00402128	cmp cl, 90h
0040212B	jnb loc_402532 ; If byte following 0x0F is greater than ; 0x90 then exit mutation process

Figure 4. Invalid instruction 'range' check.

TRANSFORM MODULE

The Transform module maps an instruction into one or more instructions. A detailed list of all the transformations is given in the appendix, which can be found at http://www.virusbtn.com/magazine/blahblah.

The transformation rules can be classified into two categories: deterministic and nondeterministic. A deterministic rule always transforms an instruction to the exact same sequence of instructions.

For example, the following rule for transforming the instruction movsb (opcode 0xA4) is a deterministic transformation rule:

```
movsb  ➔     push    eax
             mov     al, [esi]
             add     esi, 1
             mov     [edi], al
             add     edi, 1
             pop eax
```

Figure 5 shows the procedure for generating a fixed transformation for byte 0xA4 representing movsb.

A non-deterministic rule may transform an instruction to a different sequence of instructions. The following two rules demonstrate non-deterministic rules:

Location	Instructions
004023B0	cmp al, 0xA4
	;if byte is not 0xA4 goto next step
004023B2	jnz 004023CE
004023B4	add esi,1
	; Increment esi to analyze next byte
004023B7	mov eax, 83068A50
004023BC	stos dword ptr es:[edi]
004023BD	mov eax, 78801C6
004023C2	stos dword ptr es:[edi]
004023C3	mov eax, 5801C783
004023C8	stos dword ptr es:[edi]
	; If al contains 0xA4, insert the equivalent byte
	; sequence 50 8A 06 83 C6 88 07 83 C7 01 58
	; at the buffer location pointed to by edi
004023C9	jmp 00401FF8
	; goto analyze next byte

Figure 5. Transformation of byte 0xA4.

```
mov eax, [ebp+4]        →    push ecx
(8B 45 04)                   mov  ecx, ebp
                             add  ecx, 41h
                             mov  eax, [ecx-3Dh]
                             pop  ecx

mov eax, [ebp+4]        →    push esi
(8B 45 04)                   mov  esi, [ebp+4]
                             mov  eax, esi
                             pop  esi
```

Whenever the code introduced by a rule modifies a register, say *reg*, which was not modified by the original instruction, the mutated code is wrapped between 'push *reg*' and 'pop *reg*' instructions.

PATCHING RELOCATABLE ADDRESSES

W32/Evol does not contain any jump and call instructions that use absolute addresses, rather all the branching instructions use relative jumps. The virus also contains no indirect jumps and calls, where the target address is available in a register or some other memory location.

Since the transformations replace one instruction with multiple instructions, the mutation engine must also modify the relative addresses of the jump and call instructions.

In order to update the relative addresses, the mutation engine maintains another buffer, BUF2, of size 16 x [length of virus code]. For each instruction of the virus program, BUF2 has four entries, as shown in Table 1.

Entry 1 (DWord)	Entry 2 (DWord)	Entry 3 (DWord)	Entry 4 (DWord)
Source	Dest	Next address following opcode	Original offset

Table 1. A record in the buffer BUF2.

The first entry of the table is Source**,** this points to the address of the nth instruction in the virus code. The second entry, Dest, points to the address in BUF1 where the transformed virus code is stored. (*Note that the mutation engine takes BUF1 as input.*)

The other two entries in the table are zero unless the instruction carries a relocatable offset, in which case the third entry points to the address where the calculated offset is to be stored. The last entry stores the value of the current offset.

The change in the length of the code results in a change of relative addresses. To update the relative offsets, the algorithm searches for all the non-zero 'Entry 3' locations, i.e. instructions that have offsets.

If an instruction, *I*, with a non-zero offset is found, it adds the original offset (Entry 4) to Source (Entry 1), to obtain address *a*. Address *a* is the original destination address in the W32/Evol code. Since this destination address should start a valid instruction, there should be a valid record in BUF2 such that Source is equal to *a*. (*Note that BUF2 has records corresponding to each valid instruction in virus code.*)

The difference between the values of Dest at the location of instruction *I* and Dest at location *a* gives us the new offset. This offset gives the number of bytes that have been added in the transformed code. The offset is then patched back to the location pointed by Entry 3 at the location of instruction *I*.

DEFEATING W32/EVOL

W32/Evol is no longer considered to be a major threat – most of the current AV scanners can catch it because of its relatively simple morphing engine. Yet it may be worth contemplating how this virus could be defeated. The insights could lead to the development of methods for defeating other metamorphic viruses.

W32/Evol uses some very interesting techniques to obfuscate system calls. It is probably beyond the scope of current static analysis techniques to undo these obfuscations and identify the system functions being called by the virus. It appears to be futile to follow that direction.

However, the limitations of the metamorphic engine of W32/Evol are clearly its weaknesses.

- It uses linear sweep for disassembling itself. Hence, it can be disassembled by most disassemblers.
- It cannot use indirect jumps and calls because it cannot transform them correctly. Thus, its control flow graph can be created easily, thereby simplifying its reverse engineering.

- Its deterministic transformation rules essentially replace a certain byte with a certain fixed sequence of bytes. These rules can be applied in reverse.
- The code generated by non-deterministic transformation rules follows the pattern: push *reg*, *instructions*, pop *reg*, where the *instructions* does not contain *push* or *pop*. The *push* and *pop* instructions form a pair of parenthesis. All such pairs are properly matched in the generated code. It should be possible to undo the transformation using a parenthesis-matching algorithm.

Now consider a program Undo.Evol that does the following: it disassembles a program using linear sweep and then applies the transformations of W32/Evol in reverse. The program continues to apply the transformations until none of the transformations can be applied.

Will Undo.Evol program help in detecting versions of W32/Evol?

Since the transformations of W32/Evol always result in an increase in the code size, when they are applied in reverse they will always decrease the code size. Thus, Undo.Evol will always terminate.

It is a matter of further study whether Undo.Evol will always terminate on a single program. If it can be shown that Undo.Evol terminates on a single program, say Min.Evol, then to detect W32/Evol one may apply Undo.Evol on a binary and check for the signature of the Min.Evol.

CONCLUSIONS

Anti-virus scanner technology is constrained by the theoretical limits of program analysis techniques. A metamorphic virus is a manifestation of these limits. In fact, metamorphic viruses also depend on program analysis techniques, because in order to mutate, a metamorphic virus must analyse its own code. Thus a metamorphic virus cannot use tricks that will fool its own analyser.

This handicap of metamorphic viruses can potentially be exploited to develop AV scanners. However, to reverse the mutations in order to defeat a virus, the AV research community faces several key questions, such as: How does one extract the assumptions of a virus and the transformations it performs? Will reverting the transformations lead to a single result? Will the reverse transformations terminate in polynomial time? And how does one separate virus code from the code of the host?

The answers to some of these questions would be crucial in developing technology that takes advantage of the Achilles' heel of a virus.