

The Viral Darwinism of W32.Evol
An In-depth Analysis of a Metamorphic Engine
Written By: Orr, November 2006

Introduction

As a part of my everlasting interest in mutation of executable code, I decided to reverse-engineer the metamorphic engine of the **W32.Evol** virus. In short, a metamorphic engine is used in order to transform executable (binary) code. The behavior of such an engine varies from virus to virus, but many elements remain the same. A metamorphic engine has to implement some sort of an internal disassembler in order to parse the input code. After disassembly, the engine will transform the program code and will produce new code that will retain its functionality and yet will look different from the original code. There are many ways to achieve such a goal, and this paper will deal specifically with the metamorphic engine of Evol and its inner mechanics.

Information regarding the behavior of the virus itself is not included in this paper and is available on many Antivirus websites on the Internet, namely the Symantec website.

Legend:

Examples in this paper include shortened naming of assembly language expressions:

```
Reg - Register (i.e. EAX, EBX)
Mem - Memory address (i.e. [EAX])
r/m - Register or Memory
imm - Immediate Value (i.e. OP Reg, ACABh)

OP = {ADC, ADD, AND, CMP, OR, SBB, SUB, XOR}
OP1 = {DIV, IDIV, IMUL, MUL, NEG, NOT, TEST}
OP2 = {RCL, RCR, ROL, ROR, SAL, SAR, SHL, SHR}
```

I included an HTML dump from the IDA Pro disassembler, with the source of the metamorphic engine fully commented and named. This paper simply documents the aspects of the engine, as presented in the source.

Calling

The engine is called in the following way:

```
push    [ebp+var_14]           ; *outBuf (EDI)
call    GetSizeOfCode
push    eax                   ; sizeOfCode
call    SeekStartOfVirus
push    eax                   ; *inBuf (ESI)
call    MetaEngine
cmp     eax, 0
jz      short EngineFailed
```

Or, in psuedo-C:

```
MetaEngine(*inBuf, sizeOfCode, *outBuf);
```

Where, *inBuf is a pointer to the code, sizeOfCode is the size, *outBuf is the output to the destination buffer where the mutated code will be stored.

Code-Analysis

The engine will perform a analysis over the given code. Aside from on-the-fly disassembly, the virus will allocate 4 table entries for each instruction it analyzes. Each of the entries is a double-word. The structure is accessed in the following way:

+00	InputIP	Pointer to Instruction in the Input Buffer
+04	OutputIP	Pointer to Mutated Instruction in Output Buffer
+08	OffsetNewRelative	Pointer to offset of New Relative Branch Value
+0C	NewRelative	New Relative Value for a Branch Instruction

When the engine first loads, it will use allocate SizeOfCode*16 bytes using VirtualAlloc for the purpose of the above-mentioned table. In the end, theses bytes will be freed using VirtualFree. The virus itself uses internal 'caller' functions (callVirtualAlloc / callVirtualFree), and doesn't call the API's directly.

Every time the engine loads a new instruction for analysis, the first two members of the structure are filled, and the 3rd member is zeroed for later use. The 3rd and 4th fields will only be filled in case the engine analyzes a branch instruction (JMP/Jcc/CALL), to be used when the relocations will be fixed, after the mutation process is complete.

The engine will disassemble only instructions that the author had included, meaning it would fail with unrecognized / unsupported instructions.

Sample Disassembly:

```
cmp     al, 8Ah                ; MOV r8, r/m8?
jz      short _Mutate?
cmp     al, 8Bh                ; MOV r32, r/m32?
jz      short _Mutate?
cmp     al, 8Dh                ; LEA r32, mem?
jz      short _Mutate?
```

As you can see, the engine simply checks for the current opcode, and if it is recognized by the engine, it will take an action accordingly.

Code Transformations

1. Instruction Transformation

The engine supports several kinds of instruction mutations, meaning it will write different code with the same functionality. The defined transformations are divided into two parts:

- Inter-Engine Transformations: These transformations are inlined inside the engine, and are a part of the engine's core.

Original	Transformed
<ul style="list-style-type: none"> - PUSH r/m8 - PUSH r/m32 	<pre>MOV EAX, r/m PUSH EAX</pre>
<ul style="list-style-type: none"> - MOV reg, imm 	<pre>a. MOV reg, Random ADD reg, imm-Random b. MOV reg, Random SUB reg, -(imm-Random) c. MOV reg, Random XOR reg, Random^imm</pre>

- External Transformations: These transformations are called like that due to the fact that their ‘physical’ location is outside the main engine function. Despite this fact, these routines act as if they are inside the engine itself, and when they are finished they jump back to the engine. There is no visible point behind this, and my guess is that they were added after the initial coding process.

Original	Transformed
<ul style="list-style-type: none"> - MOV r/m, reg - MOV reg, r/m - TEST r/m, reg - LEA r32, mem - OP r/m, reg - OP reg, r/m 	<pre>PUSH RandomReg MOV RandomReg, OriginalReg ADD RadnomReg, RandomImm8 OP r/m - RandomReg, OriginalReg POP RandomReg</pre>
<ul style="list-style-type: none"> - MOV r/m, reg - TEST r/m, reg - OP r/m, reg 	<pre>PUSH RandomReg MOV RandomReg, OriginalReg OP OriginalR/M, RandomReg POP RandomReg</pre>
<ul style="list-style-type: none"> - MOV reg, r/m - LEA reg, mem - OP reg, r/m 	<pre>PUSH RandomReg MOV RandomReg, OriginalReg OP RandomReg, OriginalR/M MOV OriginalReg, RandomReg POP RandomReg</pre>
<ul style="list-style-type: none"> - OP r/m8, imm8 - MOV r/m8, imm8 - TEST r/m8 	<pre>PUSH RandomReg MOV RandomReg8, Imm8 OP OriginalR/M8, RandomReg8 POP RandomReg</pre>

The engine's decision whether to transform a given instruction or not is based upon a random factor. The engine asks for a random number between 0 and 7, and the transformation will be applied only if it is 0 – meaning a probability of 1/8.

2. Alternative Instruction Encoding

The Intel instruction format allows different binary encoding for the same action. The engine supports the following alternative encodings:

Original Encoding	Modified Encoding	Mnemonics	
7x imm8	0F 8x imm32	Jcc short	Jcc near
EB imm8	E9 imm32	CALL short	CALL near
A8 imm8	F6 C0 imm8	TEST AL, imm8	TEST AL, imm8
A9 imm32	F7 C0 imm32	TEST EAX, imm32	TEST EAX, imm32
3F imm8	80 ModRM imm8	OP AL, imm8	OP AL, imm8
3F imm32	81 ModRM imm32	OP EAX, imm32	OP EAX, imm32
83 ModRM imm8	81 ModRM imm32	OP r/m32, imm8	OP r/m32, imm32

3. Fixed Transformations

The engine will replace the following bytes with the corresponding sequences:

Original Opcode	Transformation	Mnemonics	
A4	50 8A 06 83 C6 01 88 07 83 C7 01 58	MOVSB	PUSH EAX MOV AL, [ESI] ADD ESI, 1 MOV [EDI], AL ADD EDI, 1 POP EAX
A5	50 8B 06 83 C6 04 89 07 83 C7 04 58	MOVSD	PUSH EAX MOV [EAX], ESI ADD ESI, 4 MOV [EDI], EAX ADD EDI, 4 POP EAX
AA	88 07 83 C7 01	STOSB	MOV EDI, [AL] ADD EDI, 1
AB	88 07 83 C7 04	STOSD	MOV EDI, [EAX] ADD EDI, 4
AC	8A 06 83 C6 01	LODSB	MOV AL, [ESI] ADD ESI, 1
AD	8A 06 83 C6 04	LODSD	MOV EAX, [ESI] ADD ESI, 4

As you can see, these instructions do not have any parameters passed onto them, thus simply being replaced with their corresponding functionality.

4. Junk-Code Insertion

The engine will generate instructions that are not reliant upon the original code, and their functionality is essentially “do-nothing”. The junk instructions will only be added if the last written byte is between 50h to 52h (PUSH EAX/ECX/EDX).

Junk Instructions	
MOV	r32, [ebp+Random8]
MOV	r32, Random32
OP	r32, Random32 ;ADC/ADD/AND/OR/SBB/SUB/XOR
MOV	RandomReg8, Random8

It may be noted, however, that these instructions actually **do** alter the original code flow as they are random and inserted in places in which they will be executed, but these instructions are inserted after PUSH instructions, so we assume the registers will be modified later on.

5. General Instructions

In any other case the engine will store the instruction as is, aside from exceptional opcodes:

Opcode	Mnemonics	Action
90	NOP	Don't store
0F xx (80 > xx > 90)	Special Opcode 0F Not supported by engine	Abort Engine
CC	INT 3 (Debugger Breakpoint)	Anti Debug
81 C4	ADD ESP, imm32	Store
81 EC	SUB ESP, imm32	Store
83 C4	ADD ESP, imm8	Store
83 EC	ADD ESP, imm8	Store
C0	OP2 r/m8, imm8	Store
D0	OP2 r/m8, imm8	Store
CD	INT	Store
8B EC	MOV EBP, ESP	Store
F3	REP Prefix	Store
C3	RET	Store
50 - 5F	PUSH r32 / POP r32	Store

Relocations Fixup

After the mutation process is completed, the engine fixes instruction relocations. Due to the fact that many times the transformation process results in growth of code, most (if not all) of the branch instructions will lead to an incorrect place in the destination buffer. The engine will utilize the relocation-table it created during the mutation process, and it will patch the new address into place. First, it will loop through the table. For every instruction it will add the 1st and 4th fields (InputIP + NewRelative), thus calculating a virtual original destination. It will then set a second loop that will search for that destination, and patch the entry using the 3rd and 4th fields.

Other Features

1. Anti Debugging

If the engine will detect a breakpoint over the code it mutates, it will jump to the following routine:

```
AntiDebug:
    cmp     byte ptr [ebx+7], 0BFh      ; are we in kernel mode?
    jnz    short ret_AntiDebug
    mov    ecx, 1000h                  ; counter = 1000h
    mov    edi, 40000000h
    or     edi, 80000000h
    add    edi, ecx                    ; edi = C0001000h
    rep stosd                          ; copy bytes to [edi]

ret_AntiDebug:
    retn                               ;This will result in a crash
```

The above routine can also be considered as 'external', as it is called from the main virus body as well as from the engine.

2. Internal Functions

The engine contains several functions that it uses for many actions:

- **Random** – Returns a random number in EAX.
- **Rnd7** – Returns a random number between 0-7 and checks if it's 0.
- **CheckDisplacement** – Returns the displacement of a given opcode in CL.
- **InvertSign** – If necessary, the function inverts the sign (- / +) of AL.
- **GetRandomDword** – Returns a random dword in EAX.
- **GetWBit** – Extracts the W bit from the opcode into DL.
- **ModifyDH** – Modifies the DH according to the W bit.
- **GetRandomReg8** – Returns a random 8 bit register in BL.
- **GetRandomReg32** - Returns a random 32 bit register in BL.
- **MakePushRandomReg** – Generates a PUSH RandomReg instruction.
- **MakePopRandomReg** - Generates a POP RandomReg instruction.

Sample Transformations

Presented below are the actual transformations performed by the engine on itself.

B9 00 10 00 00	mov	ecx, 1000h	
B9 10 B2 00 3C	mov	ecx, 3C00B210h	
81 C1 F0 5D FF C3	add	ecx, 0C3FF5DF0h	; ecx = 1000h

Some of the “external” transformations:

8B 45 0C	mov	eax, [ebp+0Ch]	56 89 EE 83 C6 56 8B 46 B6 5E	push mov add mov pop	esi esi, ebp esi, 56h eax, [esi-4Ah] esi
89 43 08	mov	[ebx+8], eax	51 8B C8 89 4B 08 59	push mov mov pop	ecx ecx, eax [ebx+8], ecx ecx
33 C0	xor	eax, eax	51 89 C1 33 C8 8B C1 59	push mov xor mov pop	ecx ecx, eax ecx, eax eax, ecx ecx
80 F9 50	cmp	cl, 50h	52 B2 50 38 D1 5A	push mov cmp pop	edx dl, 50h cl, dl edx

As you can see in the above examples, the mutated byte-sequences are entirely different then the original ones.

Ending

The analysis of the virus engine took me a lot of time, mainly due to the fact that it was done statically, without running the code. I hope this paper helps to shed more light on the idea of how metamorphism is done, as well as the aspects involved in the design of such an engine. Further, I’d like to thank the author of this engine, for creating this piece of code that enhanced my interest in this particular field. I encourage you to look over the reversed source-code of the engine, as it will probably make all things written above a little bit more clear.

Thanks for reading this.
As always, any feedback is always welcome.

Orr,
November 2006

OrrIscariot@gmail.com