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/
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 3 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:

1. 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.

**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:
Whenever the code introduced by a rule modifies a register, say \( \text{reg} \), which was not modified by the original instruction, the mutated code is wrapped between 'push \( \text{reg} \)' and 'pop \( \text{reg} \)' 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.

<table>
<thead>
<tr>
<th>Location</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>004023B0</td>
<td>cmp al, 0x4A</td>
</tr>
<tr>
<td>004023B2</td>
<td>jnz 004023CE</td>
</tr>
<tr>
<td>004023B4</td>
<td>add esi,1</td>
</tr>
<tr>
<td>004023B7</td>
<td>mov eax, 83068A50</td>
</tr>
<tr>
<td>004023BC</td>
<td>stos dword ptr es:[edi]</td>
</tr>
<tr>
<td>004023BD</td>
<td>mov eax, 7801C6</td>
</tr>
<tr>
<td>004023C2</td>
<td>stos dword ptr es:[edi]</td>
</tr>
<tr>
<td>004023C3</td>
<td>mov eax, 5801C783</td>
</tr>
<tr>
<td>004023C8</td>
<td>stos dword ptr es:[edi]</td>
</tr>
<tr>
<td>004023C9</td>
<td>jnz 00401FF8</td>
</tr>
</tbody>
</table>

Table 1. A record in the buffer BUF2.

The first entry of the table is Source, this points to the address of the n\textsuperscript{th} 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 \textit{reg}, \textit{instructions}, pop \textit{reg}, where the \textit{instructions} does not contain \textit{push} or \textit{pop}. The \textit{push} and \textit{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.

\textbf{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.