POLYMORPHIC AND METAMORPHIC CODE APPLICATIONS IN PORTABLE EXECUTABLE FILES PROTECTION

Liviu PETREAN
"Emil Racoviţă" High School Baia Mare, 56 V. Alecsandri, tel. 0262 224 266

Abstract: Assembly code obfuscation is one of the most popular ways used by software developers to protect their intellectual property. This paper is reviewing the methods of software security employing metamorphic and polymorphic code transformations used mostly by computer viruses.

Keywords: *code, polymorphic, portable*

I. INTRODUCTION

The illegal copying of computer programs causes huge revenue losses of software companies and most of the time these losses exceed the earnings. As a consequence the software companies should use strong protection for their intellectual property, but surprisingly, we often encounter the absence of such protection or just a futile security routine. Many software producers argued these frailties affirming that sooner or later their product will be reversed with or without protection [1], [3], [6]. They are right but only partially, because even if everything that can be run can be reversed, the problem is how long is the reversing process going to take and how experienced must the reverser be?

After studying many reversing tutorials on different websites we reached the conclusion that many reversers have a minimal knowledge of assembly language and they have no idea why or how this process works, calling themselves newbies. Most of the time the easiest trick or diversion from the standard protection procedures can drive them out or make them quit.

In this paper we are reviewing self-modifying code techniques used to improve computer programs' security helping the software developers to add one more protection layer to their products.

II. SELF-MODIFYING CODE

In computer science terminology, **self-modifying code** [5] is a code that alters its own instructions at runtime. It was used in the early days of computers in order to save memory space, which was limited. It was also used to implement subroutine calls and returns when the instruction set only provided simple branching or skipping instructions to vary the flow of control. Self-modifying code was used to hide copy protection instructions in 1980s DOS based games. The floppy disk drive access instruction "int 0x13" would not appear in executable program's image but it would be written into the executable's memory image after the program started to execute. Nowadays self-modifying code is used by programs that do not want to reveal their presence such as computer viruses and executable compressors and protectors.

Self modifying code is quite straightforward to write when using assembly language but it can also be implemented in high level language interpreters as C and C++. The usage of self modifying code has many purposes. Those which present an interest for us in this paper are mentioned below:

1. Hiding the code to prevent reverse engineering, through the use of a disassembler or debugger.
2. Hiding the code to evade detection by virus/spyware scanning software and similar programs.
3. Compression of the code to be decompressed and executed at runtime, e.g. when the executable file is protected by a compressor.

The self-modifying code is used by executable compressors and computer viruses in combination with the following types of assembly language obfuscations:

Polymorphic code [5] is the code that mutates while keeping the original algorithm intact. This technique is sometimes used by computer viruses, and computer worms to hide their presence. Most anti-virus software and intrusion detection systems attempt to locate malicious code by searching through computer files and data packets sent over a computer network. If the security software finds patterns corresponding to known computer viruses, it will apply a scanning strategy by rewriting the unencrypted decryption engine each time the virus or worm is propagated. The first known polymorphic virus was written by Mark Washburn in 1990, and the virus was called "1260". Later in 1992, a more well-known polymorphic virus was invented by the Bulgarian reverser Dark Avenger as a means of avoiding pattern recognition from antivirus-software.

Metamorphic code [5] is the code that can reprogram itself. This is possible often by translating its own code

Manuscript received December 26, 2009; revised March 21, 2010

into an equivalent temporary representation; the code edits this temporary representation of itself, and then writes itself into the normal code again. Metamorphic code is more effective than polymorphic code. This is because most anti-virus software will try to search for known virus-code even during the execution of the code.

Diversion code [5] is the code that is not executed at runtime and is meant to draw the attention of reversers. Most of the time it is composed of junk code instruction generated by a polymorphic engine.

The three terms defined above usually appear in articles related to computer viruses whose major problem was to hide their bodies from anti-virus programs. But the polymorphic and metamorphic codes have many applications in software security when the developers want their intellectual property hidden from the prying eyes of the reverse engineers.

III. DESIGN AND USAGE OF SELF MODIFYING CODE

Self-modifying code represents one of the strongest weapons used by the developers of the commercial executable programs compressors today. In our previous sections we looked at the definitions and the basic principles of self-modifying code. In this section we will explore more deeply advanced techniques like polymorphism and metamorphism.

As we have already mentioned, polymorphism was used for the first time to write the body of a virus and was meant to hide this body through encryption. The principle is simple: the virus starts with a decryption routine that decrypts the body of the virus in memory and after that, the actual virus code is run. The same concept may be used when protecting an executable program encrypted using polymorphism. The executable starts with a decryption routine that decrypts the rest of the program in memory then the actual code of the application runs normally. The decryption routine must be injected in the code of the protected executable by the compressor.

The easiest way to approach polymorphism looks like this:

```
            pushad
            mov edi, pStart
            mov ecx, dwLen
decr_next_byte:
            xor byte ptr [edi],byteKey   (1)
            dec ecx
            jz Done
            inc edi
            jmp decr_next_byte
            Done:   popad
```

This kind of polymorphism is easy to implement but let us take a look at this code from a reverser's point of view: the code starting with the byte indicated by the pointer pStart with the length in bytes of dwLen is encrypted. The encryption can be made by a standalone encrypting application or even easier with a hex-editor. In this example we have chosen a simple XOR encryption with the key byteKey. Starting with the byte at pStart all the following bytes are decrypted in memory at runtime revealing the real assembly code hidden by the encryption. The easiest way to go through this routine using a debugger is to place a memory breakpoint on the POPAD instruction and wait for the code to decrypt.

The point of polymorphism is to force the reverser to run our program, static disassembling being impossible because of the encryption, and thus useless. The problem with this small polymorphic engine is that it is too obvious and an experienced reverser doesn't even have to run the program. All he has to do is study the decryption routine and write a small program in assembly language designed for IDA Pro Disassembler, Olly Debugger, or a similar disassembler to decrypt the hidden code. To patch the application the reverser will use the NOP assembly language instruction to decrypt the code of the application.

In the following paragraphs we will refer to Olly Debugger, this being the most popular debugger among the crackers' communities on the web. Although designed to help programmers to debug and repair programs' flaws, Olly is used by reversers to crack and patch commercial applications and almost all the reversing tutorials available on the web today are based on Olly Debugger.

IV. ADVANCED POLYMORPHISM

A more powerful polymorphic engine not only randomly generates the encryption key protecting the encrypted code but it also alters the encryption routine. The features of a strong polymorphic engine are:

1. Generating different instructions to do the same thing.
2. Swapping sets of instructions.
3. Creating calls to diversion routines.
4. Generating lots of conditional jumps.
5. Encapsulating of complicated anti-debugging tricks.
6. Inserting a lot of junk code into the real code.

A combination of the features above results in a more complicated debugging, drastically increasing the reversing time or even make it impossible for less experienced reversers.

Generating different instructions to do the same thing is based on the fact that in computers programming there is always more than one way to write a code. For example, let us consider the following instruction:

```
mov eax, 33                    (2)
```

Equivalent forms of instruction (2) are:

```
push 33          or        mov eax, 56
pop eax                    xor eax, 65
```

The representation using an intermediate language can also assist us in implementing the remaining features of a polymorphic engine. The theory behind representation using an intermediate language is explained in books about compiler design [2], [8].

Nevertheless, there is a vulnerability of classical polymorphism which consists in the fact that after finishing the polymorphic routine, the sensitive code is left decrypted in memory. This means that if the reverser manages to get through the hard-to-debug mechanism of the polymorphic engine and all the computer-generated code, he will find the comprehensible original code. This is something that the protection must prevent.

The solution of this problem is the division of our code into smaller modules and put each of them into its own polymorphic envelope. This will make reversers life harder, because he will never see the whole code at once and he will have to trace through the polymorphic decryptors annoyingly often.

To make the polymorphic encryption more secure, each block of code will be encrypted using algorithms similar to Tiny Encryption Algorithm (TEA) based on Feistel cipher with at least three rounds, see [17].

V. METAMORPHISM AS AN EFFECTIVE WEAPON

The solution to the problem exposed in section 5 is called metamorphism. From the outside it is similar to polymorphism because it creates a different code for each application. The key difference between polymorphism and metamorphism is that while polymorphism encrypts the sensitive code and creates a unique decryptor for it, metamorphism morphs the sensitive code to make it almost impossible to understand by a human. With metamorphism it is possible to create kilobytes of morphed code from several bytes of original code. Manual tracing of such code can easily take days or even weeks of difficult debugging with poor results (the reverser will never see the original code as with polymorphism).

Metamorphic engines first take existing code, analyze it using an internal disassembler, morphs the internal representation of code then generate the morphed native code. Let us have an example:

```
mov eax, 23
mov ecx, 3
```

The resulting morphed code can look like this:

```
xor eax,eax
or eax, 22
inc eax
sub ecx, ecx
inc ecx
add ecx, 1
inc ecx
```

The major difference between the implementation of polymorphism and metamorphism lies in the fact that polymorphism does not change the original code. It only hides it. On the other hand metamorphism alters the original code and thus has to cope with several problems:

1. **Code flow:** because each instruction is replaced with several new instructions, the length of the code blocks changes. The engine has to detect and repair all jumping coordinates or function calls within the code to match new positions of code blocks.
2. **Registers used as pointers:** it basically represents the same problem like the code flow.
3. **Detecting data in code:** most compilers today place some data in the text section of executable, together with the code. An attempt to handle data as code would have fatal consequences. This is the reason why metamorphism is never used for the whole application but only for the part of it containing the sensitive code like the protection itself.

Because of great complexity of the task of writing a metamorphic engine, many of the available commercial protectors resort to partial metamorphism. They decide not to write a full morpher but select only a small subset of instructions that will be morphed. The other instructions are left without change.

This approach fulfills the goal of metamorphism only partially. Even though it is difficult to understand the generated code, it is even more difficult or impossible to understand it with full metamorphism. The reason for this is that the subset of affected instructions is usually too small to generate sufficient amount of confusing code.

Metamorphic code may be written using the following techniques:

1. **Permutations:** By permutating n different sequences of code among themselves we may obtain n! different copies that perform the same task.
2. **Transformations:** Experienced programmers know that there are many ways to write the code for a special task. Assembly code transformations or metamorphic code transformations are special sequences of code injected into the assembly code of a portable executable file to make the code more difficult to follow and to discourage reversers. The transformations that we used in our project are based on the 8086 instructions set, see [4], [7], [8], [9], [15].
3. **Garbage instructions:** Garbage or junk code is the code that performs practically nothing. It is inserted in the assembly code of portable executable programs to make debugging a long and painful task. Experienced reversers identify garbage code and avoid it by setting breakpoints to bypass its execution.

For example, the insertion of unconditional jumps to the very next instruction alters the program's code as we may observe in the figures 1 and 2 below captured from OllyDebugger.

```
00401000  r$ B8 64000000   MOV EAX,64
00401005  .  BB C8000000   MOV EBX,0C8
0040100A  .  03C3          ADD EAX,EBX
0040100C  .  33DB          XOR EBX,EBX
0040100E  .  33C0          XOR EAX,EAX
```

Figure 1. Original assembly code before inserting the jump instruction

```
00401000  $ B8 64000000   MOV EAX,64
00401005  . BB C8000000   MOV EBX,0C8
0040100A  .v EB 00        JMP SHORT skeleton.0040100C
0040100C  > 03C3          ADD EAX,EBX
0040100E  . 33DB          XOR EBX,EBX
00401010  . 33C0          XOR EAX,EAX
```

Figure 2. Transformed assembly code after inserting the jump instruction

Using arithmetic and logical operations, certain pieces of code may be transformed as in the example below. Let us consider the original instruction:

mov eax, 100

This instruction can be altered using a table containing more metamorphic transformations equivalent to the original one like in table 1 below:

T #1: xor eax,eax add eax, 100

T #2: push eax
mov eax, 35
pop eax
mov eax, 123
sub eax, 23

T #n: mov eax, A4
add eax, B4
sub eax, 14
xor eax, 120
(144H XOR 120H = 64H = $100_{(10)}$)

Table 1. Transformations equivalent to instruction mov eax,100

where T#i means transformation number 1, and $100_{(10)}$ means the decimal integer 100 equal to the hexadecimal 64H. The transformations can be implemented in a text file and a metamorphic engine will be designed to insert randomly such transformations in the protected code.

VI. THE EFFICIENCY OF ENCRYPTION

The most commonly method to achieve polymorphism is code encryption [1], [2], [3], [6]. To evaluate the advantages and disadvantages of using self-modifying (polymorphic) code and thus encryption, we develop a small assembly program designed to encrypt/decrypt itself at runtime and thus altering its own instructions. The program is implemented using Winasm Studio 5.1.3.0 and its execution at runtime is studied with Olly Debugger 1.10. Both of these products are distributed freely on the internet and can be downloaded from www.winasm.net and www.ollydbg.de.

Our application starts with a message box that lets us know that we have only fifteen days left to use it. After pressing OK, the rest of our small program is a window containing some text. This is a typical pattern used in many trial versions of commercial computer applications. From a reverser's point of view we are particularly interested in the starting message box, called "nag-screen", and we want to remove it from the execution of our target and thus have a clean copy of the program for ourselves.

From a developer's point of view we want our software tried for only fifteen days and then registered or removed from the hard disk. To reach this goal we must be able to hide the sensitive code specific only to trial versions in the best possible way. This code launches the message box when running the unregistered program every time.

The sensitive code characterizing only the trial version may contain also other important sections like a registration routine or time based verifications.

Now, back to our program, as a method of protection for the code that handles our "nag-screen" we choose polymorphism. The unprotected debugged message box code looks like this:

```
00401032  . 6A 00         PUSH 0
00401034  . 68 9E304000   PUSH Window.0040309E
00401039  . 68 86304000   PUSH Window.00403086
0040103E  . 6A 00         PUSH 0
00401040  . E8 DD010000   CALL <JMP.&user32.MessageBox
```

Figure3 . MessageBox assembly code in OllyDebugger

Figure 3 shows how the parameters of MessageBox function are pushed to the stack from left to right starting at the relative virtual address 1032H accordingly to the "STDCALL" model. We must choose if we want to encrypt just the code handling the message box or the whole application's code. Let us apply both methods to observe the results better. The encryption starts at address 1032H.

We encrypt the nineteen bytes of the message box code by XOR-ing each one with 51H. To be able to study the application's execution correctly we must insert a decryption routine before the encryption one. Then, before quitting the program we encrypt one more time the same nineteen bytes. The encryption/decryption routine (1) has been already presented as a small XOR encryption with a static key of byteKey=51H. After encryption, the bytes of the message box code look like in figure 4 below:

```
00401032  . 59            POP ECX
00401033  ? 335B AD       XOR EBX,DWORD PTR DS:[EBX-53]
00401036  ? 0373 33       ADD ESI,DWORD PTR DS:[EBX+33]
00401039  . 5B            POP EBX
0040103A  ? B5 03         MOV CH,3
0040103C  ?v73 33         JNB SHORT Window.00401071
0040103E  . 59            POP ECX
0040103F  ? 33DB          XOR EBX,EBX
00401041  ? EE            OUT DX,AL
00401042  ? 3233          XOR DH,BYTE PTR DS:[EBX]
00401044  ? 336A 0A       XOR EBP,DWORD PTR DS:[EDX+A]
```

Figure 4. Encrypted MessageBox assembly code in OllyDebugger

First, we must observe, like in figure 4, that all the code generated by the polymorphic routine and replacing the original one, starts at the same relative virtual address 1032H. The instructions that follow are completely altered but this method does not provide a strong protection because encrypting only a part of the code leaves the rest of it unprotected at the reverser's discretion [16].

To improve the previous method we will encrypt all the executable code, thus providing a stronger stealth factor for our sensitive code. This time the first executed routine will be the decryption routine, to be able to normally run the application. Let us examine with Olly Debugger the first unencrypted instructions of our program:

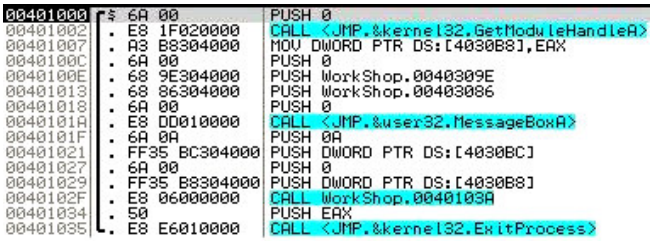

Figure 5. Entire program assembly code in OllyDebugger

In figure 5 we may notice the entry point of our executable program with the relative virtual address 1000H, where the API function GetModuleHandle is called with the NULL parameter. GetModuleHandle then returns a handle to the current running process whose address is stored in the EAX registry. The message box code starts at address 100CH. The parameters of the MessageBox function are visible between the addresses 100CH and 1018H. Then the assembly code contains the parameters of the routine that calls the main body of our application at address 102FH, and the exit function at address 1035H. By applying the polymorphic routine to all this assembly code we obtain the code in figure 6 below:

Figure 6. Entire program encrypted assembly code in OllyDebugger

After the encryption of the whole application's body the only instructions left visible are the instructions that prepare the polymorphic routine. These instructions start at the executable's entry point with the relative virtual address 1000H and are followed by the call at the address 100EH. Below this call we may see only the junk code generated again by our polymorphism. This is the call of the routine that decrypts the rest of the application which is necessary to run it correctly.

None of the presented encryption methods is not recommended to be used in commercial applications. Such protection methods might thwart the plans of a beginner reverser but would not stand a chance in front of an experienced one because of the static encryption keys used in each example presented in figures 3, 4, 5, and 6.

Instead, we may use encryption keys generated using serial numbers of hardware components belonging to the computer where the protected program will be used.

VII. IMPROVEMENTS AND CONCLUSIONS

This article represents a review of the basics of the polymorphic and metamorphic codes and their implementation in assembly language. To improve such protection methods based on self-modifying code we should use more complex encryption and compression algorithms. The following set of features adds considerably more improvement when protecting an application:

1. Application integrity check.
2. Compression of the application.
3. Counteraction to debuggers and disassemblers.
4. Creation and verification of registration keys using public keys algorithms.
5. Counteraction to memory patching.
6. Generating of registration keys based on the computer systems.
7. Possibility to create trial versions, that limit application functions based on evaluation times and number of runs.
8. Deletion of import information and API redirecting.

When deciding to use a commercial compressor it is very important to make sure that there are no automatic unpacking programs for it or that methods of unpacking are not made public. The self-modifying code and assembly code obfuscation represents the ultimate weapon of a software developer against reversing. For the better use of executable programs compressors and the features above, developers should program their own packers providing different protection schemes for different products of the same developer.

The bottom line of using polymorphic and metamorphic code as methods of protection should be the delay of the reversing process or even the impossibility of reversing the protected program.

This article is also one of the results, see [10], [11], [12], [13], [14], of the study of more than 800 commercial applications during the last 4 years among which we enumerate, photo, video, sound converters and players, educational software, anti-spyware and anti-virus programs, business and office programs, games, etc developed by more than 210 software companies. If only half of this information falls into the wrong hands, the prejudice produced to the programmers who dedicated time and money to their products would be devastating in time. This is the major reason why we must emphasize the security problems, their solutions and vulnerabilities and we must bring new solutions to combat software piracy more efficiently.

REFERENCES

[1] Eldad Eilam, *Reversing – Secrets of Reverse Engineering*, Wiley Publishing, Inc. 2005.
[2] Garry McGraw, *Software Security: Built Security In*, Pearson Education, Inc. 2006.
[3] Kris Kaspersky, *Hacker Disassembling Uncovered*, A-LIST Publishing, 2003.
[4] Michael Howard, David LeBlanc, John Viega, *19 Deadly Sins of Software Security – Programming Flaws and How to Fix Them*, The McGraw-Hill Companies, 2005
[5] Michal Strehovsky - "Advanced Self-Modifying Code", http://www.free-articles-zone.com/print.php?id=42171
[6] Pavol Cerven, *Crackproof Your Software – The Best Way to Protect Your Software Against Crackers*, No Starch Press, Inc. 2002.
[7] R. Hyde, *The Art of Assembly Language*, No Starch Press, 2003
[8] S. Steven Muchnik, *Advanced Compiler Design and Implementation*, Morgan Kaufmann Publishers, Inc. 1998.
[9] G. Toderean, A. Caruntu, O. Buza, A. Nica, *Microprocesoare – Indrumator de Laborator, editia 2007,* Risoprint 2007
[10] L. Petrean, "Software Security – Polymorphic and Metamorphic Code Applications*", SIITME 2007 International Symposium for Design and Technology of Electronic Packaging,* pp. 212-217, Baia Mare, Romania, ISSN 1843-5122
[11] L. Petrean, "Software Security – General Method of Literal Strings Obfuscation*"*, *IWCIT 2007 International Workshop Control and Information Technology*, pp. 41-46, Ostrava, Czech Republic, ISBN 978-248-1567-1
[12] L. Petrean, G. Toderean, "Portable Executable Format Security Patterns*"*, *IWCIT 2008 International Workshop Control and Information Technology*, pp. 170-174, Gliwice, Polland, EAN-978839047438
[13] L. Petrean, G. Toderean, "Portable Executable File Format – Import Address Table Redirection and Reconstruction", *microCAD 2008 International Scientific Conference 20-21 March 2008*, pp. 49-54, Miskolc, Ungaria, ISBN 978-963-661-812-4
[14] L. Petrean, G. Toderean, "Assembly Metamorphic Code Transformations as a Protection Method Of Portable Executable Files"*, microCAD 2009 International Scientific Conference 19-20 March 2009*, Vol. XXIII, pp. 35-40, Miskolc, Ungaria, ISBN 978-963-661-866-7
[15] G. Muscă, *Programare în Limbaj de Asamblare,* Teora, 1996
[16] C.E. Landwehr, "A taxonomy of computer program security flaws, with examples", *ACM Computing Surveys*, vol. 26, no. 3, pp. 211–254, September, 1994
[17] B. Schneier, *Applied Cryptography, Second Edition: Protocols, Algorithms and Source Code in C,* John Wiley & Sons Inc., 1996