Polymorphism: Level 6B (Polymorphism: Chaotic Permutations)

vxug.fakedoma.in/archive/VxHeaven/lib/vch03.html

VX Heaven

[I'd like to point out that there's a large number of variations to this idea and I'll leave it to your creativity to figure out what hides in the shadows.]

1. Introduction

Definition by Eugene Kaspersky:

> **Level 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.

Consider a code of three instructions A, B and C. If these instructions are written in a row, one after another, there're six (3!=3*2*1) possible arrangements: ABC, ACB, BAC, BCA, CAB, CBA. This method of permutation I call level 6a. So what is level 6b?

The three instructions A, B and C occupy spaces that we label **os**. Now, let's add one more space, but it'll always be unoccupied so we'll label it **us**. How many permutations are now possible?

(os+us)!=(3+1)!=4*3*2*1=24

The powers of this become obvious when **(os+n*us)!**, where **n** is any number. This method I call 6b polymorphism.

The idea behind polymorphism at level 6 is to divide our code into blocks that can be permutated so that scanners can't detect them with search strings with/without wildcards. There're various approaches to level 6 code, f.ex:

```
[loop method]      [sequence method]     [internal method]
@00: cmp bl, 01    @00: ...              @00: ...
     jne @01           jmp @01              jmp @01
     inc ebx       @01: ...                 ...
     ...               jmp @02          @01:jmp @02
@01: cmp bl, 00                         @02:...
     jne @00                                jmp @03
     inc ebx                                ...
     ...                                @03:jmp @04
```

['...' is in each example the same, I've only changed the logical structure.] Whatever logical structure all methods require that you pack your code into blocks that can be permutated. Some viruses that use level 6 techniques are f.ex TMC and Ply viruses.

2. Approaching Level 6B

One way to pack your code is to use 'minimal blocks'. Minimal blocks just consist of 1 instruction followed by an uncoditional jmp (short/near/far). F.ex:

```
push 0                 ; 2b   -->   push 0                     ; 2b  | block 00
push offset filename   ; 5b         jmp @01                    ; 2b  |
call _lcreat           ; 5b         @01: push offset filename  ; 5b  | block 01
                                    jmp @02                    ; 2b  |
                                    @02: call _lcreat          ; 5b  | block 02
                                    jmp @03                    ; 2b  |
```

The point with this is to let us be able to spread our program randomly over/into another program. This example executes its instructions consecutively, for the moment let's forget about loops and calls, so if we've spread our program in a random fashion we only have to link the jmps together so the program will execute in the right order.

The first code had a total size of 12b while the packed code became 18b long. One way to minimize the packed code size is to bundle several instructions together, but then there's the risk of becoming recognizable with search strings.

The blocks have a variable size and to make it easier to deal with them I introdcue the term 'invisible boundary'. The **ib** ('invisible boundary') value is the size of our largest block. In our previous example the largest block was 01/02 and then the value of **ib** becomes 7.

If our code consists of 10 blocks and the **ib** value is 7 then the minimum size of another code (of the program we want to infect) must be at least 70b. So if another code is 77b long then it has for our code 10 **os** and 7-1 **us** which may result in 16! permutations. Wow!

But why 7-1? We 'must' reserve the first block in the other program for our first block, the execution follows a critical sequence.

Remember, when writing to the blocks we don't write as many bytes as ib's value. F.ex, when **ib** equals 7:

```
[our code]        [other code]       [new code]        [overwritten code]
push 0  ; 2b   +  inc eax ; 1b  =    push 0  ; 2b
jmp @01 ; 2b      inc ebx ; 1b       jmp @01 ; 2b
                  inc ecx ; 1b       dec eax ; 1b
                  inc edx ; 1b       dec ebx ; 1b
                  dec eax ; 1b       dec ecx ; 1b
                  dec ebx ; 1b       dec edx ; 1b
                  dec ecx ; 1b                    >   inc eax   | put
                  dec edx ; 1b                        inc ebx   | them
                                                      inc ecx   | somewhere
                                                      inc edx   | else
```

So, what does our program have to do?

1. Permutate the code
2. Modify the blocks

The easiest thing to do when permutating the code is to generate a new sequence, like this:

```
[old sequence]   -->   [new sequence]
01020304050607   -->   01070305040602
```

When generating a new sequence you'll quite likely have to use some kind of sorting algorithm that sorts some random numbers that you'll 'generate'. Now we can simply write the blocks to their new locations and all we've to do now is to link the jmps together.

When the **ib** is equal to 7 and we've got 3 blocks and the original sequence looks like '123' and the new sequence looks like '132' then we've to link 1 to 2 to 3. But, since the blocks are of variable size we've to find how many byte precede a jmp instruction. In this case block 1 has a jmp after 1b and block 2 after 2b. So:

```
Block 1 jmps to Block 3 = (7b - 3b) + 7b = 11b jmp ; here
3b = 1b + 2b (jmp + coordinates)
Block 3 jmps to block 2 = -3b - 7b = -10b jmp      ; here 3b = 2b + 1b (jmp
only)
```

But let's assume we've managed to perform our permutation and our code is now all over the place. How can we read the blocks in the right sequence? We must trace the code with the help of the jmp coordinates. To place our first block at the Program_Entry_Point of the other program might, for the beginning, be a good idea.

And yes, code like 'mov al, 235' must be avoided when using jmps since, in this case, jmp=235.

3. Internal Jmps and Calls

Now we've a block that looks like this:

```
loopw [email protected]   ; a type of jmp    or   call MessageBox   ; a call
jmp @06         ; the regular
jmp        jmp @06
; the regular jmp
```

My first advice is that you avoid them as much as possible. If you've got to use calls and especially APIs then I suggest you import them. To get some ideas you should hav a look at the article 'Accessing Windows 95 API's by scanning PE-tables' by Lord Julus, which can be found in VDAT 1.9.

The easiest way around the jmp and call problem is to write your entire program to the beginning of the other program and to place its code to the end of itself, then you don't even have to manipulate the PE headers etc. You just have to swap the code around and execute the other program... and instead of hiding in the other program's code we can now generate random code to fill our code or rip it (more believable).

If you don't use the methods mentioned above it'll get more complicated. There're many different types of jmps and scanning for them is just like scanning for the end-of-block jmp, but it's more work. Calls are different, but they've also got some coordinates that need to be updated. F.ex, when calling an API the first instruction 'in' the procedure is usually a 'jmp dword [...]'. These instructions exist at the end of the regular code and when moving a block with a 'call' then we only have to add/substract the number of bytes we've moved up/down, and since we're dealing with blocks this is really easy because of the **ib** method.

You may f.ex hardcode all the coordinates of the blocks or search through the code.

4. Problems with Level 6

I suggest you use near jmps everywhere and skip all loop types, rewrite them as jmps, and you should also try to write as few loops as possible. Having strings/variables in the data section becomes more complicated. And, don't forget to deal with the 'jmp dword [...]' at the end of the code. There's also a significant increase in program size. With only structural changes the size increase will definetely be greater than 200%. If, f.ex, there comes a scanner that actually traces through the code, much like we do, then the code will be identifiable with a simple search string. I suggest you combine level 6 polymorphism with the usual encryption/decryption routines used in standard polymorphic code (level 1-5). I believe that this method will yield quite good results.

5. The End and Future

This method is rather difficult to implement and I suggest that all beginners first try other ways of coding polymorphic programs, like oligomorphic programs (level 1).

This method has a lot of potential and I hope that many viruses will make use of it. Best of luck.

[This paper is a work in progress and you may send comments/questions to . If I get some positive feedback on this paper I'll consider writing several guides to all levels of polymorphism. I apologize if there're still unclear issues in this paper (this all represents my first tries at asm/virus/polymorphism), but I wrote this paper for me and you just happen to come second, for now. Hope you got some ideas.]