Tutorials - Do Polymorphism

ivanlef0u.fr/repo/madchat/vxdevl/vdat/tumisc76.htm

This tutorial isn't to discuss about any polymorphism matters, or to just explain you it's basics. It's fully oriented for you to at last *learn* how to write a polymorphic engine, with useful tips on how to implement it.

Some coders, even some really good ones, feel it so difficult when it's time to come into polymorphism. "Ok, I have to swap instructions, but how the hell do I make that, how do I control the decryptor length and that the decrypting instructions are on their place ?" and so on.

We start from the very beggining. What is polymorphism ? The abitily from a virus to change it's shape each time it infects a file. As you know, even if your virus is encrypted, the bytes from the routine that decrypts the main body will still remain static, so the AV can detect you easily. So, the main idea about polymorphism is changing that static shape so AV have to change their way to detect your virus.

After reading this, you should be able at least to make an "oligomorphic" engine, which is better than no polymorphism at all: oligomorphism is some kind of despective word used with polymorphic not-very-polymorphic viruses.

```
    .-----------------.
    |                 |<-- Virus decryptor
    |-----------------|
    |                 |<-- Virus body ( encrypted )
    '-----------------'
```

Let's imagine you use simple XOR encryption. Then your decryptor would look as this:

```
        call delta
delta:  pop bp
        call decrypt
        [...]                   ; encrypted code
decrypt:
        mov dl,byte ptr ds:[encryption_key+bp]
        lea si,encryptinit+bp
        mov cx,encryptend-encryptinit
loop_on_decryption:
        mov al,byte ptr ds:[si]
        xor al,dl
        mov byte ptr ds:[si],al
        inc si
        loop loop_on_decryption
        ret
```

You will notice that this decryptor is very unoptimized, and that thing would be so bad for permutating instructions as it's easier to get fixed bytes this way. We can calculate the delta offset before infecting even in win32 - by looking at the base address -, and make this a bit smaller:

```
        mov bp,XXXX
newdelta equ $-2        ; So you just have to place new delta there
        lea si,bp+encryptinit
        mov cx,encryptend-encryptinit
decr_loop:
        xor byte ptr ds:[si],XX
        inc si
encrvalue equ $-1       ; Again you place it in the decryptor
        loop decr_loop
```

Now we have just five instructions. A lot better, no ? We can even optimize itmore by just eliminatin the "mov bp,XXXX", and precalculate bp+cryptbegin loading it to si, loading the delta offset inside the encrypted code:

```
        mov si,XXXX                             ; 3 bytes
        mov cx,cryptend-cryptbegin              ; 3 bytes
decr_loop:
        xor byte ptr ds:[si],XX                 ; 3 bytes
        inc si                                  ; 1 byte
        loop decr_loop                          ; 2 bytes
```

Wow, just we have five instructions to make a decryptor, not just that but look at their length: we can easily divide them in four blocks of three instructions, as all the blocks have same length !. Of course inc si and loop decr_loop can be in the same block, as really it is as if they were one, the two are needed there.

Then, just remember you have to keep the original instructions anywhere if you change them, or generate them independant from what was generated earlier.

Now let's see what can we permutate. Look at the two first instructions. Does anything matter if we change their place ? Not really, they aren't dependant on each other, so their position can be changed. Your engine could do that easily.

We go to the XOR. More fixed bytes, but who said you can only encrypt with XOR ? You can change that XOR for an ADD in example, making SUB when encrypting. You should make a NOT, ROR, etc:

```
       xor byte ptr [si],XX                 80h 34h XXh
       add byte ptr [si],XX                 80h 04h XXh
       sub byte ptr [si],XX                 80h 2ch XXh
```

So, just changing the second opcode we can permutate among different encryption schemes. But again who said that SI has to be the pointer ? Here we have another change we could make. Look at this:

You just have to change also the SI loading to play with a lot of registers (and the inc si, of course). Also Intel cared about us virus writers, and when using 32bit access you can use any register as a pointer, which gives you much more possibilities.

It's easy to imagine other permutations even in this little bunch of code. You could load in SI or whatever not the encryption start but the end and change the INC SI for a DEC SI. You could change the LOOP for a DEC REGISTER/JNZ and then change the CX register for another one. No limits for your imagination.

So what ? Let's say you want to implement this. It doesn't seem difficult, as you should just use random numbers to decide whether to do or not one change. To make it easier, I recommend:

- Remember to make a copy of the original instructions, or don't depend on that data when generating new ones for another infection. If you change the place where ESI is loaded with the delta shit, you wouldn't probably find it again. Take a buffer to mess up the instructions and change them, and each time you are going to infect replace them, in example.
- Try to do an infector that just uses this kind of oligomorphism/ polymorphism. When it's done, it will be time for more serious issues.

The polymorphic method before is a bit unnefective: you could eliminate some or all fixed bytes, but that's not enough. Also, any person with a middle knowledge in viruses that looks at that code will notice that it's a virus, and that he is watching a decryptor.

So you have to make garbage among the instructions. Which of these sounds better for you ?

```
mov dx,0ffffh
mov ax,bx
add ax,121Dh
> mov di,4355h
div di
inc dx
jnz 1211h
> mov cx,85Ch
[...]
```

The second style may look as if it was a legitimate program. Of course there are some points that have to be fixed, but this is the way to get it. As I surely conviced you on the neccesity of a decryptor, here is the main objective of this section, building the opcode generator.

First of all, you'll wonder how to build all that garbage. I recommend you making by yourself a list of opcodes in instructions. Then, you have to classify them. You can do it in example by the number of bytes, or by looking at if they need random values. To accomplish this just take debug or a good source as the Intel Developers Insight.

Let's think on the second one. Let's say we have the opcodes for a list of instructions which are based on 1 opcode plus a 2 byte value. This can be mov reg,imm16 in example. So, when you call the routine that generates that kind of instructions, the opcode for the desired

instruction plus the random value would be selected.

For this code it's supposed you've made a getrndnumber function which returns a number smaller than the parameter AX: you can easily achive this by dividing the random number by AX and storing the remainder which will be lesser than AX.

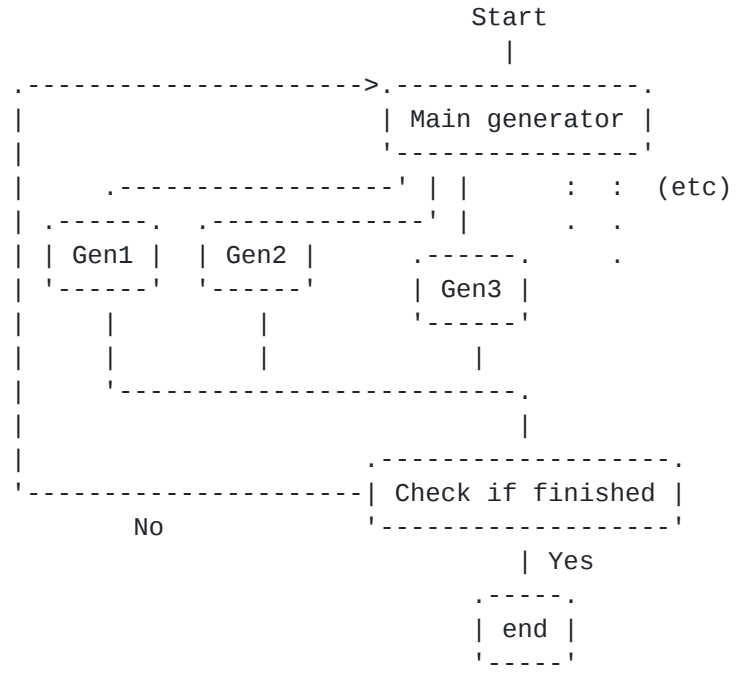
It's not this simple as you have to check if your decryptor uses one of this registers, but I think you'll get the idea. A good way to avoid overwriting useful registers can be keeping a byte which bits indicate if one register is used or not in this part of the decryptor, or in the whole decryptor if you don't want to make things more complicated.

```
 AX BX CX DX BP SI DI
.--------------------.
|  |  |  |  |  |  |  |
'--------------------'
  0  0  1  0  0  1  0
```

So, a 1 will tell us that it's used, a 0 that you can operate with it without messing the decryption: imagine the bad stuff that would be making a /mov si,startencr/add ax,1200h/xor si,1100h :)

Getting deeper in this into advanced polymorphism, you shouldn't just store the opcodes of the instruction, as you're wasting lots of space. I recommend you Lord Julus's guide, where you can get a deeper view on this. I just remind you that you should watch out how redirection modes go when operating with registers: any r/m instruction's second byte is called the 'ModR/M' byte. There, the instruction stores the redirection mode: first two bits get in four groups, and the next three bit fields indicate which register is used; when one of the operands of the instruction is a register, it's AX = 000, CX = 001, DX = 010 and so on

Now you more or less know how to build garbage. You can use lots of routines as the one I wrote calling them randomly and create a shitload of garbage there. GenX represents one kind of garbage:



BTW,this isn't finished, and we are getting to the part some coders fear. Making things finally work.

Maybe another difficult thing is mixing the garbage generator and the decryptor. In this part, I've preferred to list some problems and give you some advice on how to solve them:

> a) How do I call a "generate decryption instruction"
>
> As well as you call any "generate garbage" one. The best way you can do this is by making generation in two phases: you generate X bytes of garbage and then one instruction, and so on until you reach the end. You should then make for example 30-50 bytes of garbage + 3 bytes of instruction + 30-50 bytes, etc, until you reach the end.
>
> And remember to put some garbage after the last instruction and before the first one !. Another good thing could be changing the entry point randomly also.
>
> b) Calculate the final loop
>
> Either if you choose a loop for decryption or a conditional jump, you should calculate the distance between that and the xor/add/sub/etc instruction as it's a relative jump. Of course that's easy for any coder - remember it's stored in two complement -, but you should keep in mind it's a short jump in DOS, so you can't place them too far from each other. Fortunately, since 486 you can make long jump structures, with 16bit or 32bit relative jumps as there are alternative conditional jump instructions. Coolio for your win32 virus polymorphic engine.
>
> c) What if I want stealth so I need a fixed length generator ?
>
> This is also very simple. You can make the generation variable in length, but correct it in the next call to garbage generation, fixing the bytes to the one you desire.

```
generate12:
  mov AX,7h                                   ; Seven registers
  call getrndnumber
  lea si, registertable
  add si,ax
  lodsb                                       ; We have the desired opcode
  stosb                     ; We suppose di points to the decryptor

  mov ax,0ffffh
  call getrndnumber
  stosw                     ; Now we store a two bit random number

registertable:
  db   0b8h,0bbh,0b9h,0bah,0bdh,0beh,0bfh   ; MOV reg,imm16
    ;   AX   BX   CX   DX   BP   SI   DI
```

So, this is the end. Following my advice you shouldn't find it difficult getting into one of the most personal and powerful parts in a virus, polymorphism.

- Try to use undocumented opcodes. Some info can be found in www.x86.org. Opcodes as 0f1h - or int1h/ICEBP, maybe 0d6h - SALC - should be used to kill code emulators which want to trace your code: bad stuff is it won't work with AMD or Cirix processors as they are undocumented.

- Avoid stupid looking stuff. That would be modifying a register and then loading any value on it without saving the last one in example. You can look which registers are touched when you generate them or by looking at the MODR/M byte of the instruction. You shouldn't have anything like this:

- If you want to make reentrant calls, one simple method is generating also inconditional jumps this way:

You can fill the garbage place then finish with a ret. Recursive calls to the polymorphic engine would be a good way to do that. So, as you have the beggining of the garbage, you can call it when you want.

- Don't place interrupts/long time wasting instructions inside the loop. Illustrating this is very simple: imagine you used some service from the int 21h, and you encrypted the code in a byte step. If there were four of this interrupts inside the loop and the virus is 2Kb long in example, you would execute 2048x4 = 8192 int 21h calls. But sure in most modern computers wasting a lot isn't really important.

- When writing a polymorphic engine for win32, one trick you can do is saving some data in the data section of the PE in your virus so you can write in the old one, giving your decryptor memory direct writing capability.

- Lord Julus polymorphism tutorial in 29A#2, so interesting low level shit.
- Intel Family Developer's Manual, complete opcode description in it's implementation: very useful.
- Any other VX article I didn't mention :)

I'm also working in some new theories about polymorphism: you'll hear soon about me I hope.