Methods Behind A Polymorph Engine (Black Baron)

A GENERAL DESCRIPTION OF
THE METHODS BEHIND A POLYMORPH ENGINE
by
The Black Baron

This .DOC attempts to provide an insight into the workings of a Polymorph Engine. It assumes you are familiar with 8086 assembler and the logic functions XOR, AND & OR. To this end, no explanation of logic or assembler will be included in this text! Also note, no SEGMENT stuff will be included in any of the assembler listings, it is assumed that you know which segments are in play. The methods described in this .DOC are the ones used in my SMEG (Simulated Metamorphic Encryption Generator) Polymorph Engine and are by no means the only way to do it!

A small glossary of terms used in this document:

```
      ENCRYPT     = Transform from it's original form to an altered form.
      DECRYPT     = Transform from it's altered form to it's original form.
      KEY         = The register or value used to encrypt/decrypt with.
      SLIDING KEY = A KEY value that is INCREASED or DECREASED on each loop.
      COUNT       = The number of bytes in the encrypted code or data.
      INDEX       = A pointer to the encrypted code or data.
      SIGNATURE   = A unique group of bytes that can be used to check against
                    a programs content in the hope of detecting a particular
                    program.
      HEURISTIC   = A set of well defined rules to apply to a problem in the
                    hope of achieving a known result.
```

Question: What is a Polymorph?

Answer: Well, the Longman English Dictionary defines it as:

```
   "POLYMORPHOUS also POLYMORPHIC adj fml or tech.
     EXISTING IN VARIOUS DIFFERENT FORMS."
```

In other words, something that has the ability to change it's shape. Other ways to describe such a thing might be; Mutable, Metamorphic, Etc...

Question: What is a Polymorph Engine?

Answer: A program with the abilities to encrypt (or jumble up) another program or data and provide a unique decryptor for it, it must do this in such a way that no two encryptions of the same program or data will look alike.

Example: Take the following ultra-simple decryptor:

```
               MOV     SI,jumbled_data     ;Point to the jumbled data
               MOV     CX,10               ;Ten bytes to decrypt
main_loop:     XOR     BYTE PTR [SI],55    ;XOR (un_scramble!) a byte
               INC     SI                  ;Next byte
               LOOP    main_loop           ;Loop for the 9 remaining bytes
```

This small program will XOR the ten bytes at the location pointed to by SI with the value 55. Providing the ten bytes were XORed with 55 prior to running this decryptor the ten bytes will be restored to their original state. If you are unsure as to why this is, brush up on your XOR logic!!

Ok, so you might say that if you change the KEY value on each generation it will become Polymorphic? Well, yes and no! If you did that, the encrypted portion would be Polymorphic, but the decryptor would still remain mostly the same, the only change begin the KEY value! So, a signature scanner that allows WILDCARDS (and most do!) would still be able to find your decryptor!

One way you could fool some signature scanners is to swap around some of the instructions. So, with this in mind, the above decryptor might look like:

```
               MOV     CX,10
               MOV     SI,jumbled_data
main_loop:     XOR     BYTE PTR [SI],55
               INC     SI
               LOOP    main_loop
```

As you can see, still not much of a change, not really enough to fool some of the better signature scanners.

"GET TO THE POINT! WHAT IS A TRUE POLYMORPH?", I hear you cry!

Well, a "true" Polymorph would be a decryptor that looks completely different on each generation! Take the following decryptor:

```
               MOV     CX,10
               NOP
               NOP
               MOV     SI,jumbled_data
               NOP
main_loop:     NOP
               NOP
               XOR     BYTE PTR [SI],55
               NOP
               INC     SI
               NOP
               NOP
               NOP
               NOP
               LOOP    main_loop
```

This decryptor is the same as the one before it, but it has has a few random NOP instructions peppered throughout itself. On each generation you would vary the amount of NOPs after each instruction. This is a Polymorph in it's simplest form. Still, most of the good signature scanners would have no problem with such a simple Polymorph. They would simply skip the NOPs, thus having a clear view of the decryptor, to which they could apply a signature!

No, a "true" Polymorph has to be far far more complex then this! Instead of peppering NOPs throughout the decryptor it would pepper totally random amounts of totally random 8086 instructions, including JUMPS and CALLS. It would also use a different main decryptor (possibly from a selection of pre-coded ones) and would alter all the registers that the decryptor uses on each generation, making sure that the JUNK code that it generates doesn't destroy any of the registers used by the real decryptor! So, with these rules in

mind, here is our simple decryptor again:

```
               MOV     DX,10              ;Real part of the decryptor!
               MOV     SI,1234            ;junk
               AND     AX,[SI+1234]       ;junk
               CLD                        ;junk
               MOV     DI,jumbled_data    ;Real part of the decryptor!
               TEST    [SI+1234],BL       ;junk
               OR      AL,CL              ;junk
main_loop:     ADD     SI,SI              ;junk instruction, real loop!
               XOR     AX,1234            ;junk
               XOR     BYTE PTR [DI],55   ;Real part of the decryptor!
               SUB     SI,123             ;junk
               INC     DI                 ;Real part of the decryptor!
               TEST    DX,1234            ;junk
               AND     AL,[BP+1234]       ;junk
               DEC     DX                 ;Real part of the decryptor!
               NOP                        ;junk
               XOR     AX,DX              ;junk
               SBB     AX,[SI+1234]       ;junk
               AND     DX,DX              ;Real part of the decryptor!
               JNZ     main_loop          ;Real part of the decryptor!
```

As you should be able to see, quite a mess!! But, still executable code. It is essential that any junk code generated by the Polymorph Engine is executable, as it is going to be peppered throughout the decryptor. Note, in this example, that some of the junk instructions use registers that we are using in the decryptor! This is fine, providing the values in these registers aren't destroyed. Also note, that now we have random registers and random instructions on each generation it makes signature scanning (even for the clever signature scanners) impossible! Instead, an HEURISTIC method must be used, which can lead to false alarms.

So, a Polymorph Engine can be summed up into three major parts:
---

1 .. The random number generator.
2 .. The junk code generator.
3 .. The decryptor generator.

There are other discrete parts but these three are the ones where most of the work goes on!

How does it all work? Well, SMEG goes about generating random decryptors in the following way:

1 .. Chooses a random selection of registers to use for the decryptor. Leaving the remaining registers as "junk" registers for the junk code generator.

2 .. Chooses one of the compressed pre-coded decryptors.

3 .. Goes into a loop generating the real decryptor, peppered with junk code.

To understand how the selected registers are slotted into the decryptors and the junk code you must look at the 8086 instructions from a binary level:

```
XOR   AX,AX    =    00110001 11000000
XOR   AX,CX    =    00110001 11001000
XOR   AX,DX    =    00110001 11010000
XOR   AX,BX    =    00110001 11011000
```

You should be able to see a pattern in the binary code for these four 8086 instructions? Well, all 8086 instructions follow logical patterns, and it is these patterns that tell the 8086 processor which registers/addressing mode to use for a particular instruction. The total amount of instruction formats and the precise logic regarding the patterns is too complex to go into here. However, all good 8086 tutorials/reference guides will explain in full.

SMEG exploits this pattern logic to generate junk code and decryptors with random registers, as the patterns directly relate to the registers Etc.

SMEG generates junk code in the following way:

Inside SMEG there is a table of the basic binary patterns for all of the 8086 instruction set, but with one important difference, all the register/address mode bits are zero. This is called the SKELETON INSTRUCTION TABLE. The table also contains various other bytes used by SMEG to determine the relevant bit positions to "plug in" the register bit patterns. These patterns are plugged in via the logic processes OR and AND. Using this method, SMEG can generate endless amounts of random 8086 instructions without destroying any of the registers used by the decryptor proper.
SMEG also contains some discrete logic for producing false CALLS to dummy subroutines and also false conditional JMPS around the junk code.

SMEG generates the decryptor proper in the following way:

Inside SMEG there is a table containing a selection of common 8086 instructions used in decryptors, such as XOR [index],reg Etc. These are, again, stored in SKELETON FORM with some control bytes used by the decryptor generator. Also, inside SMEG, there are several pre-coded decryptors stored in a compressed form. On average, a complete decryptor can be described to the decryptor generator in as few as 11 bytes and adding to the list of pre-coded decryptors is both painless and economical with space!

SMEG generates the Polymorphed decryptor in the following way:

First it chooses, at random, one of the pre-coded compressed decryptors. Next it goes into a loop uncompressing each decryptor instruction, plugging in the required registers, storing it and then generating (for each real instruction) a random amount of random instructions. This loop repeats until the complete decryptor has been constructed. The final result is a random size, random register, random patterned decryptor!

It should also be noted that whenever SMEG generates an INDEXed instruction it uses either SI, DI or BX at random, also it sometimes uses a random offset.

For example, say the encrypted code started at address 10h, the following could be used to index this address:

```
       MOV   SI,10h     ;Start address
       MOV   AL,[SI]    ;Index from initial address
```

But sometimes SMEG will generate something like the following, again based on the encrypted code starting at address 10h:

```
       MOV   DI,0BFAAh      ;Indirect start address
       MOV   AL,[DI+4066h)  ;4066h + 0BFAAh = 10010h (and FFFF = 10h)!!
```

These indexed and initial values are picked at complete random, and the examples of 0BFAAh and 4066h are valid, but next time they will be completely different!

The following are two decryptors that were generated with my SMEG Polymorph Engine. It should be noted that I generated 4000 examples with no two alike! Unfortunately I ran out of hard drive space! But it is fairly safe to say that the total number of decryptor combinations would run into the BILLIONS!

All the lines marked with ";junk" in the following listings indicate random junk instructions that were inserted throughout the actual decryptor, note that SMEG has the ability to generate junk CALLS to false SUBROUTINES, as well as general junk conditional jumps! All lines marked with a * indicate an actual part of the decryptor proper. I chose the two generations shown because their sizes were similar, 386 and 480 bytes. SMEG produces decryptors ranging in size from as little as 288 to as much as 1536 bytes. Even if two decryptors are generated that are the same size the chances of them being the same are, literally, billions to one!

```
;Assembler listing for decryptor 1, size 368 bytes.
;--------------------------------------------------
;Size of the encrypted code was 07DBh (2011 bytes)
;The encrypted code started at address 0270h

;This decryptor was generated to use the following registers:
;
;   DX = Count of bytes in the encrypted code
;   BX = Index pointing to the encrypted code
;   AL = The encryption key
;   CL = General work register

0100    JNS     0103            ;junk
0102    CLD                     ;junk
0103    SAR     SI,CL           ;junk
0105    CMP     BP,0708         ;junk
0109    STC                     ;junk
010A    JG      010E            ;junk
010C    OR      SI,CX           ;junk
010E    XOR     DI,3221         ;junk
0112    ADD     BP,0805         ;junk
0116    AND     BP,3512         ;junk
011A    SHR     SI,CL           ;junk
```

```
011C    MOV     SI,1B04         ;junk
0120    SAR     DI,CL           ;junk
0122    ADC     SI,2506         ;junk
0126    ADC     DI,1F11         ;junk
012A    SBB     BP,[0F3E]       ;junk
012E    CMP     BP,3F1E         ;junk
0132    DEC     SI              ;junk
0133    NOT     DI              ;junk
0135    AND     SI,083D         ;junk
0139    INC     SI              ;junk
013A    SBB     DI,0103         ;junk

013E    MOV     DX,1791         ;*  Set up the COUNT register
                                ;   3x Actual number of bytes!

0141    CLD                     ;junk
0142    JB      0146            ;junk
0144    TEST    SI,AX           ;junk
0146    SBB     DI,SP           ;junk
0148    TEST    DI,[251B]       ;junk
014C    TEST    CL,[SI]         ;junk
014E    SHL     BP,1            ;junk
0150    MOV     BX,017D         ;junk
0153    CMC                     ;junk
0154    MOV     DI,1218         ;junk
0158    JO      015C            ;junk
015A    RCR     DI,1            ;junk
015C    STC                     ;junk
015D    CMP     BP,DI           ;junk

015F    MOV     AX,CS           ;*  Get CODE SEG in AX

0161    TEST    CH,[BX+17]      ;junk
0164    SBB     BP,3107         ;junk
0168    INC     DI              ;junk
0169    RCR     BP,1            ;junk

016B    MOV     DS,AX           ;*  Make DATA SEG = CODE SEG

016D    ADD     DI,[3B04]       ;junk

0171    MOV     AL,50           ;*  Set up decrypt KEY reg

0173    JNB     0179            ;junk
0175    MOV     SI,1439         ;junk
0179    JB      017D            ;junk
017B    ADC     DI,AX           ;junk
017D    JMP     0185            ;junk
0180    MOV     BP,1B36         ;junk
0184    RET                     ;junk
0185    RCR     SI,1            ;junk

0187    MOV     BX,842D         ;*  Set up the INDEX register

018A    SUB     SI,CX           ;junk * Decryptor MAIN LOOP
```

```
018C    OR      DI,0B0F         ;junk
0190    MOV     BP,1E3E         ;junk
0194    RCL     DI,CL           ;junk
0196    SUB     BP,2E12         ;junk
019A    ADD     DI,[2E2A]       ;junk
019E    ROL     SI,CL           ;junk

01A0    MOV     CL,[BX+7E43]    ;*  Get next encrypted byte
                                ;   NOTE:  original index 842Dh plus 7E43h =
                                ;   10270h AND FFFFh = 0270h!  Which is the
                                ;   start of the Encrypted code!

01A4    JZ      01AC            ;junk
01A6    TEST    BH,[DI+2B3B]    ;junk
01AA    CMP     [BP+SI],DL      ;junk
01AC    ROL     DI,1            ;junk
01AE    SBB     DI,263A         ;junk

01B2    DEC     DX              ;*  Dec the COUNT register (x1)

01B3    CALL    0180            ;junk
01B6    MOV     DI,CX           ;junk
01B8    ADC     BP,282E         ;junk

01BC    SUB     CL,AL           ;*  Decrypt byte using KEY reg

01BE    MOV     SI,372A         ;junk
01C2    TEST    BP,3A10         ;junk
01C6    CALL    0180            ;junk
01C9    ADC     SI,1317         ;junk
01CD    CLD                     ;junk

01CE    INC     AX              ;*  Increase the KEY reg

01CF    XOR     SI,203D         ;junk
01D3    JMP     01E1            ;junk
01D6    DEC     DI              ;junk
01D7    CMC                     ;junk
01D8    SUB     BP,[3624]       ;junk
01DC    XOR     SI,0200         ;junk
01E0    RET                     ;junk
01E1    CMP     [SI+13],BH      ;junk

01E4    SUB     DX,0001         ;*  Dec the COUNT register (x2)

01E8    CMP     AX,0517         ;junk
01EC    SUB     BP,2816         ;junk
01F0    AND     SI,0807         ;junk
01F4    SUB     SI,2E03         ;junk
01F8    ROR     BP,1            ;junk
01FA    INC     DI              ;junk
01FB    RCR     SI,CL           ;junk
01FD    TEST    CH,DH           ;junk
01FF    SUB     BP,1026         ;junk
```

```
0203    MOV     [BX+7E43],CL    ;*  Store the decrypted byte

0207    JNB     020D            ;junk
0209    XOR     DI,1B30         ;junk
020D    CLD                     ;junk
020E    ADD     SI,3C38         ;junk

0212    INC     BX              ;*  Increase the INDEX reg

0213    XOR     DI,0B2C         ;junk
0217    JMP     022F            ;junk
021A    OR      BP,1C18         ;junk
021E    JLE     0221            ;junk
0220    DEC     BP              ;junk
0221    ADC     SI,0E32         ;junk
0225    AND     DI,1522         ;junk
0229    CMP     [BP+SI+36],BH   ;junk
022C    ROL     SI,1            ;junk
022E    RET                     ;junk
022F    SHL     DI,1            ;junk
0231    SHR     DI,1            ;junk

0233    DEC     DX              ;*  Dec the COUNT register (x3)
                                ;   Hence the 3x original size!

0234    JNZ     023F            ;*  Not zero then jump to 023Fh

0236    TEST    CL,[BP+DI]      ;junk
0238    ADC     BP,012D         ;junk

023C    JMP     025B            ;*  Finished decrypting!

023F    INC     BP              ;junk
0240    JNB     0246            ;junk
0242    CMP     BX,0E2E         ;junk
0246    TEST    DI,SI           ;junk
0248    SBB     SI,3233         ;junk

024C    MOV     CX,018A         ;*  Set address of MAIN LOOP

024F    ROL     DI,1            ;junk
0251    SUB     DI,BX           ;junk
0253    SHR     DI,1            ;junk
0255    TEST    BL,[BX+DI+1C2E] ;junk

0259    PUSH    CX              ;*  Stack LOOP address
025A    RET                     ;*  RETurn to MAIN LOOP

025B    MOV     SI,211F         ;junk
025F    CMP     BL,[BX+DI]      ;junk
0261    SUB     BP,2D33         ;junk
0265    MOV     BP,3735         ;junk
0269    XOR     SI,SI           ;junk
026B    MOV     BP,[0A38]       ;junk
```

```
026F    INC     DI              ;junk

0270    The encrypted code starts here.

;****************** End of decryptor 1 assembler listing. ******************


;Assembler listing for encryptor 2, size 480 bytes.
;-------------------------------------------------
;Size of the encrypted code was 07DBh (2011 bytes)
;The encrypted code started at address 02E0h

;This decryptor was generated to use the following registers:
;
;   AX = Count of bytes in the encrypted code
;   BX = Index pointing to the encrypted code
;   DL = The encryption key
;   CL = General work register

0100    NOT     SI              ;junk
0102    TEST    CH,[BP+DI+0F]   ;junk
0105    INC     DI              ;junk
0106    CLD                     ;junk
0107    ADC     DI,132A         ;junk
010B    JPE     0111            ;junk
010D    OR      DI,332E         ;junk
0111    INC     SI              ;junk
0112    TEST    AL,CH           ;junk
0114    JMP     0120            ;junk
0117    JPE     011D            ;junk
0119    CMP     DX,1909         ;junk
011D    RCR     DI,CL           ;junk
011F    RET                     ;junk
0120    INC     DI              ;junk
0121    TEST    DI,BP           ;junk
0123    JMP     0133            ;junk
0126    TEST    DI,0E24         ;junk
012A    TEST    DI,093A         ;junk
012E    AND     DI,SP           ;junk
0130    CMP     [BP+SI],BH      ;junk
0132    RET                     ;junk
0133    MOV     BP,0C28         ;junk
0137    TEST    DH,CH           ;junk
0139    TEST    BP,1C16         ;junk
013D    ROR     BP,CL           ;junk
013F    JZ      0145            ;junk
0141    TEST    DH,[BX]         ;junk
0143    ADD     DI,SP           ;junk
0145    TEST    CL,[SI+3435]    ;junk
0149    MOV     BP,2E08         ;junk
014D    TEST    CX,DI           ;junk
014F    CLD                     ;junk
0150    MOV     SI,3831         ;junk
0154    AND     BP,363E         ;junk
0158    ROR     DI,CL           ;junk
```

```
015A    CLC                     ;junk
015B    JNS     0163            ;junk
015D    SAR     SI,1            ;junk
015F    SBB     DI,3308         ;junk
0163    SBB     DI,362B         ;junk

0167    MOV     AX,07DB         ;*  Set up the COUNT register

016A    AND     DI,0F1E         ;junk
016E    JMP     0182            ;junk
0171    MOV     DI,2F31         ;junk
0175    CMP     CX,2212         ;junk
0179    SBB     SI,2E14         ;junk
017D    TEST    BL,[SI+341D]    ;junk
0181    RET                     ;junk
0182    CMP     BH,19           ;junk

0185    MOV     BX,B977         ;*  Set up the INDEX register

0188    TEST    AL,[DI+072C]    ;junk
018C    TEST    DI,2306         ;junk
0190    SHR     SI,1            ;junk

0192    MOV     DX,CS           ;*  Get CODE SEG in DX

0194    CALL    0171            ;junk
0197    TEST    SI,1410         ;junk
019B    CLC                     ;junk
019C    SHL     DI,CL           ;junk

019E    MOV     DS,DX           ;*  Make DATA SEG = CODE SEG

01A0    NEG     SI              ;junk
01A2    CALL    0171            ;junk
01A5    TEST    CH,[BP+DI+070F] ;junk

01A9    MOV     DL,8D           ;*  Set decrypt KEY register

01AB    MOV     DI,3A30         ;junk
01AF    JMP     01B9            ;junk
01B2    JBE     01B5            ;junk
01B4    INC     DI              ;junk
01B5    NOT     DI              ;junk
01B7    CMC                     ;junk
01B8    RET                     ;junk
01B9    XOR     CX,DX           ;junk

01BB    CALL    01B2            ;junk * Decryptor MAIN LOOP

01BE    TEST    SI,3029         ;junk
01C2    INC     DI              ;junk
01C3    SBB     DI,1E19         ;junk
01C7    MOV     DI,0038         ;junk
01CB    RCR     DI,CL           ;junk
01CD    MOV     BP,1809         ;junk
```

```
01D1    NEG     BYTE PTR [BX+4969]      ;*  NEG the byte at [BX + 4969]
                                        ;   NOTE:  original index B977h plus
                                        ;   4969h = 102E0h AND FFFFh = 02E0h!
                                        ;   Which is the start of the
                                        ;   encrypted code!

01D5    TEST    BP,2A37         ;junk
01D9    CMP     CX,2B37         ;junk
01DD    JMP     01E2            ;junk
01E0    DEC     DI              ;junk
01E1    RET                     ;junk

01E2    MOV     CL,[BX+4969]    ;*  Get the NEGed byte into CL

01E6    CMC                     ;junk
01E7    ROR     DI,CL           ;junk
01E9    INC     BP              ;junk
01EA    TEST    DI,281E         ;junk
01EE    JZ      01F3            ;junk
01F0    TEST    BH,[BX+DI+05]   ;junk
01F3    MOV     DI,160C         ;junk
01F7    SUB     BP,BP           ;junk

01F9    XOR     CX,DX           ;*  XOR byte with the KEY

01FB    TEST    BL,[BP+DI+3C]   ;junk
01FE    JNB     0204            ;junk
0200    ADD     BP,0A13         ;junk
0204    CMP     [BX+DI],CL      ;junk
0206    CALL    01E0            ;junk
0209    CALL    01E0            ;junk
020C    DEC     DI              ;junk
020D    AND     DI,073A         ;junk

0211    DEC     AX              ;*  Decrease the COUNT register

0212    XOR     DI,2036         ;junk
0216    NEG     BP              ;junk
0218    ADC     DI,SP           ;junk
021A    CMC                     ;junk
021B    CMP     BL,[BX+SI]      ;junk

021D    DEC     DX              ;*  Decrease the KEY register

021E    ADC     BP,1821         ;junk
0222    SHL     DI,CL           ;junk
0224    CMP     AX,1816         ;junk
0228    SHL     DI,1            ;junk
022A    CMP     AL,[BP+DI+1A]   ;junk
022D    MOV     SI,1819         ;junk
0231    ADD     SI,063B         ;junk

0235    DEC     DX              ;*  Decrease the KEY register
```

```
0236    SUB     BP,0028         ;junk
023A    AND     BP,1930         ;junk
023E    CLD                     ;junk
023F    ADC     BP,2D1D         ;junk
0243    SAR     DI,CL           ;junk

0245    XCHG    CX,DX           ;*  Swap CX & DX

0247    TEST    CX,DX           ;junk
0249    MOV     SI,CX           ;junk
024B    XOR     SI,030D         ;junk
024F    SUB     DI,311C         ;junk

0253    XCHG    DL,[BX+4969]    ;*  Swap [index] & DL
                                ;   NOTE:  This restores the decrypted byte!

0257    ADD     DI,0E13         ;junk
025B    CMP     BL,[BP+DI+33]   ;junk
025E    CLD                     ;junk
025F    NOT     SI              ;junk
0261    MOV     SI,3F1C         ;junk

0265    XCHG    CX,DX           ;*  Swap CX & DX, restoring the KEY in DL

0267    MOV     SI,221A         ;junk
026B    OR      BP,0D2C         ;junk
026F    MOV     DI,231B         ;junk

0273    ADD     BX,0001         ;*  Increase the INDEX register

0277    JMP     0288            ;junk
027A    ADC     BP,AX           ;junk
027C    TEST    BL,[DI+19]      ;junk
027F    TEST    DI,0321         ;junk
0283    NEG     DI              ;junk
0285    ROL     SI,CL           ;junk
0287    RET                     ;junk
0288    SBB     BP,1B0D         ;junk
028C    XOR     BP,2A23         ;junk
0290    CMP     DL,3A           ;junk
0293    TEST    BH,[DI]         ;junk

0295    AND     AX,AX           ;*  Test if COUNT is zero
0297    JNZ     02AD            ;*  Jump to 02ADh if not

0299    CALL    027A            ;junk
029C    AND     DI,291F         ;junk
02A0    JA      02A6            ;junk
02A2    MOV     DI,0514         ;junk
02A6    ADC     SI,1F2A         ;junk

02AA    JMP     02BC            ;*  Finished decrypting

02AD    JMP     02B2            ;junk
02B0    CLC                     ;junk
```

```
02B1    RET                     ;junk
02B2    SHL     DI,CL           ;junk
02B4    CLD                     ;junk
02B5    ADD     SI,2C1A         ;junk

02B9    JMP     01BB            ;*  Jump to MAIN LOOP

02BC    TEST    BH,BL           ;junk
02BE    MOV     DI,210C         ;junk
02C2    SUB     SI,1600         ;junk
02C6    CALL    02B0            ;junk
02C9    XOR     SI,2F1D         ;junk
02CD    MOV     BP,0430         ;junk
02D1    TEST    BH,[DI+362A]    ;junk
02D5    OR      DI,1C21         ;junk
02D9    STC                     ;junk
02DA    CMP     DI,2828         ;junk
02DE    CLC                     ;junk
02DF    DEC     BP              ;junk

02E0    The encrypted code starts here.

;****************** End of decryptor 2 assembler listing. ******************
```

The following are the HEX dumps for both of the above decryptors, decryptor 1 is on the left and 2 is on the right. These dumps are to show that it would be very difficult to find a signature that could be applied to each of these decryptors in the hope of detecting them both, this is the main purpose of a Polymorph Engine! To detect, therefore, you would have to write a program that tries to use intelligence to work out if what it is looking at is a Polymorph generated decryptor. This is prone to false alarms or, in certain cases, missing the decryptor totally!

HEX DUMP OF ENCRYPTOR 1, 368 bytes	HEX DUMP OF ENCRYPTOR 2, 480 bytes
7901FCD3FE81FD0807F97F020BF181F7	F7D6846B0F47FC81D72A137A0481CF2E
213281C5050881E51235D3EEC7C6041B	334684C5E909007A0481FA0919D3DFC3
D3FF81D6062581D7111F1B2E3E0F81FD	4785FDE90D00F7C7240EF7C73A0923FC
1E3F4EF7D781E63D084681DF0301BA91	383AC3C7C5280C84F5F7C5161CD3CD74
17FC720285F01BFC853E1B25840CD1E5	04843703FC848C3534C7C5082E85CFFC
BB7D01F5C7C718127002D1DFF939FD8C	C7C6313881E53E36D3CFF87906D1FE81
C8846F1781DD073147D1DD8ED8033E04	DF083381DF2B36B8DB0781E71E0FE911
3BB0507304C7C63914720213F8E90500	00C7C7312F81F9122281DE142E849C1D
C7C5361BC3D1DEBB2D842BF181CF0F0B	34C380FF19BB77B984852C07F7C70623
C7C53E1ED3D781ED122E033E2A2ED3C6	D1EE8CCAE8DAFFF7C61014F8D3E78EDA
8A8F437E740684BD3B2B3812D1C781DF	F7DEE8CCFF84AB0F07B28DC7C7303AE9
3A264AE8CAFF8BF981D52E282AC8C7C6	0700760147F7D7F5C333CAE8F4FFF7C6
2A37F7C5103AE8B7FF81D61713FC4081	29304781DF191EC7C73800D3DFC7C509
F63D20E90B004FF52B2E243681F60002	18F69F6949F7C5372A81F9372BE90200
C3387C1381EA010081F8170581ED1628	4FC38A8F6949F5D3CF45F7C71E287403
81E6070881EE032ED1CD47D3DE84EE81	847905C7C70C162BED33CA845B3C7304
ED2610888F437E730481F7301BFC81C6	81C5130A3809E8D7FFE8D4FF4F81E73A
383C4381F72C0BE9150081CD181C7E01	074881F73620F7DD13FCF53A184A81D5

```
4D81D6320E81E72215387A36D1C6C3D1
E7D1EF4A7509840B81D52D01E91C0045
730481FB2E0E85FE81DE3332B98A01D1
C72BFBD1EF84992E1C51C3C7C61F213A
1981ED332DC7C5353733F68B2E380A47
```

```
2118D3E781F81618D1E73A431AC7C619
1881C63B064A81ED280081E53019FC81
D51D2DD3FF87CA85CA8BF181F60D0381
EF1C318697694981C7130E3A5B33FCF7
D6C7C61C3F87CAC7C61A2281CD2C0DC7
C71B2381C30100E90E0013E8845D19F7
C72103F7DFD3C6C381DD0D1B81F5232A
80FA3A843D23C07514E8DEFF81E71F29
7704C7C7140581D62A1FE90F00E90200
F8C3D3E7FC81C61A2CE9FFFE84FBC7C7
0C2181EE0016E8E7FF81F61D2FC7C530
0484BD2A3681CF211CF981FF2828F84D
```

Well, I hope this brief insight into the workings of a Polymorph Engine have enlightened and possibly inspired you into having a go at writing one yourself?

(C) The Black Baron
