A idiot guide to writing polymorphic engines

An idiot guide to writing polymorphic engines
by Trigger [SLAM] '97

1. INTRODUCTION

Polymorphism is, in my opinion, very hard to explain *well* in just one little tutorial. You can find tute's about almost everything, including polymorphism, but I personally have never seen a tute covering ALL aspects of (writing) polymorphic engines. Why? Because there's so much to it, and most important, there's not *one* specific way of doing it. Many tutorials will explain and give many examples of how generated decryptors will look like, and why a random layout for decryptors is important. I will try to explain you more than the average text does, but I will still cover just the basics of polymorphism, so you can get a good idea what it takes. Also you should get an idea whether you'd really want to try to make one. Note that this is an "idiot guide to writing polymorphic engines" so the basics of *writing* one, and the practical problems you will come across will be covered. I will give you examples of constructions, not because those are the most efficient ones to use, but to teach you how you can start trying to build your OWN constructions. For that purpose a full working source code is not necessary and therefore not included. Maybe this will disappoint some readers, but I don't believe a ready-to-rip source code contributes to the creativity of the learning reader.

Throughout this article I will refer to a polymorphic engine just as an "engine". Although I will cover some basics, I'm assuming you know what is meant with things like XOR-encryption, scan strings, anti-heuristics, etc.

2. WHY POLYMORPHISM?

Nowadays, considering the virus-awareness, unencrypted viruses don't stand a chance in the wild: a virus is naked without some form of encryption. Unless it contains some nice anti-heuristic tricks, it will easily be spotted by even the lamest AV product available. It's easy to implement encryption in a virus, e.g. with a simple XOR-loop with varying immediate (=key). The gain you get out of using even a simple form of encryption is definitely worth the effort.

However, this yields just little improvement: nowadays, virus scanners are 'taught' what the average decryption loop looks like, and because those instructions are very typical for viruses, the scanner will alarm the user (=heuristic detection). A solution for this problem is to create loops with instructions which don't immediately betray their purpose. In other words, create a decryption loop which doesn't *look* like one. This will fool most heuristic engines.

At this point the virus can't be spotted as an unknown virus (this type of virus scanning is called 'generic' detection, of which heuristic detection is an example). But this still is not enough. Once the virus reaches an AV person, he can easily make a scan string from your 133+ anti-heuristic decryption routine; all your effort down the drain. Your virus suddenly becomes detectable by every AV product known to mankind...

To prevent the AV community from using scan strings to detect your virus, you should avoid any constant values in the virus. We already encrypted the virus body, but the decryptor always remains constant, and can therefore be used for a scan string. Of course, we can't encrypt the decryption loop, but a possible solution is letting the virus produce different decryption loops for each (number of) infection(s). This way, one copy of the virus will never look the same as another one. In other words: an AV product can impossibly detect the virus with a scan string.

Nice, but is this really worth all the hassle (considering e.g. most of the time engines are larger than the viruses itself) ? Most of the time: yes. First of all, a polymorphic virus is very annoying for the AV people, and assuming the engine is complex enough, they have to design a detection method dedicated to your particular engine. This will take some time, which gives your virus some more time to spread. Second of all, designing a detection method which cathces 100% of all possible generations is very hard.
The more complex your engine is, the harder it will be. If your polymorph is not 100% detectable, it still stands a chance to survive, EVEN if AV say they can detect it! At this point I agree with Bontchev: any detection rate below 100% is practically useless, as one wrongly left infected file can re-infect the whole system/network.

Concluding, it's fair to say that (well implemented) polymorphism is a very powerful tool as both a pre- and post-discovery strategy.

==========
3. HOW TO CALL AN ENGINE
==========

The first thing you will have to understand is how an engine normally is implemented in a virus. At the time, I didn't really get that part, I still don't know if it's because it actually *is* hard to understand, or just because I'm a complete moron. :) Hell, I'll explain it anyway. Here's an example of how an engine could want to be called:
(if you already understand this, you can skip this chapter, but keep these parameters in mind, as I will be referring to them in the following example engine.)

```
Entry parameters:
       DS:SI = code to be encrypted (virus body)
       ES:DI = place to put decryptor and encrypted code
       CX    = code length
       AX    = offset in memory of virus entry

Returns:
       CX    = length of decryptor + encrypted code
       DS:DX = offset of decryptor and encrypted code
```

What the engine will do, is encrypt a piece of code (the virus) with some nice scheme, and place a matching decryptor in front of it. As you can see, a user-friendly engine returns the parameters in i21/40 format: after the engine is called, the virus needs only to set AH=40 and call Int21, assuming the handle is in BX. Note that the input ES:DI and the output DS:DX are the same.

So, what to do is the following:

(1) Set the correct parameter to point to the virus body, here DS:SI;
(2) Set the correct parameter to a free space in memory, big enough to hold the decryptor and the encrypted virus body (don't forget to include the engine in the size; it's a part of the virus!) Here this parameter is ES:DI. Mostly the free space used is after the virus;
(3) Set the correct parameter to the size of the virus (again include the size of the engine, as it is a part of the virus)
(4) Set the memory offset parameter (here AX) to it's correct value. This is the offset (in memory, when the file is loaded) at which the decryptor will start. This requires an example: When infecting a 200 (decimal) bytes long COM, the offset at which the decryptor (the beginning of the virus) will start would be 100h + 200d = 1C8h. Note that the offset in the FILE would be 200d = C8h, but as you know a COM file is loaded in memory after PSP, ie. starting from offset 100h. With EXEs, this value would be the EXE's new IP you're storing in the file header, as DOS doesn't mess with this offset.
(5) We're almost ready to call the engine. The last thing to check is if the engine destroys registers that aren't used as parameters. It will save you some debugging. :) After that, call poly!

4. AN EXAMPLE ENGINE

A polymorphic engine produces (from a given amount of variations) a random decryptor which matches the (also random) encryption method on each infection. How does it know how to do that? The programmer of the engine thought of a couple of decryptor layouts, from which the engine chooses one, and then it randomizes any variable bytes in it. This requires explanation: let's just assume that a (primitive) engine has only one decryptor layout. Let's say it looks like this:

```
[1]    MOV     <reg16> , offset_of_first_encrypted_word
    :decrypt_next_byte:
[2]    <crypt> [<reg16>] , random_immediate
[3]    <add2>  <reg16>
[4]    CMP     <reg16> , offset_of_last_encrypted_word
[5]    JB      decrypt_next_byte
```

```
Legenda: <reg16>        = a random 16 bit register (AX, BX, SI, BP, etc.)
         [<reg16>]      = pointer to contents of offset stored in <reg16>
                          eg: BX=1234, [BX]=what's at offset 1234
         <crypt>        = a crypt instruction, eg: XOR, ADD, SUB
         <add2>         = add 2 to the <reg16> (add 1 for byte encryption)
```

The engine will fill in all variable values:

[1] It selects (from a table given by the programmer) a random register to

work with. In this example the engine will store the MOV instruction encoded with the correct register (more on this later), after which it will store the offset of the first encrypted byte (or word) of the virus.

[2] After that, it selects (again from a table given by the programmer) a decrypt operation. Note that the engine has to remember which decrypt operation it chose, because later on, the virus has to be encrypted in such a way that the previously generated decryptor will correctly decrypt the virus. That's right, the encryption method is derived from the generated decryptor. The engine stores the crypt-instruction, and then it will generate a random value (=immediate) which will accompany the crypt operation. Again this value needs to be remembered, otherwise later on, the virus can't be encrypted so that the decryptor works. For example, if the random value is 1234 and the crypt-operation is XOR and the register is BX, the crypt-instruction is XOR [BX],1234 (duh). The engine stores the value, and repeats [3] for a (random) amount of times, as one single encryption operation yields a weak encryption.

[3] Add 2 to <reg16>. Advanced engines make a choice whether to use byte or word encryption, but I'm assuming this engine uses word encryption. The register needs to be increased by two, so that in the next decryption loop the decryptor will decrypt the next word. (why am I explaining this.. :))

[4] Compare the register to EOV (end of virus). Speaks for itself. (I hope :))

[5] If we haven't reached EOV yet, jump and decrypt the next byte/word.

4.1 REGISTER ENCODING

In this example the engine wants to pick a register. To understand how this is done we need to take a look at instruction encoding of the 80x86. At the time I was working on my engine, I didn't have technical documents so I grabbed SoftIce and a Hex-Bin calculator and started investigating. Here's a piece of my investigation on MOV reg16,reg16:

INSTRUCTION		HEX	BINARY
MOV	AX,AX	8BC0	10001011 11000000
MOV	AX,BX	8BC3	10001011 11000011
MOV	AX,CX	8BC1	10001011 11000001
MOV	AX,DX	8BC2	10001011 11000010
MOV	BX,AX	8BD8	10001011 11011000
MOV	BX,BX	8BDB	10001011 11011011
MOV	BX,CX	8BD9	10001011 11011001
MOV	BX,DX	8BDA	10001011 11011010
MOV	CX,AX	8BC8	10001011 11001000
MOV	CX,BX	8BCB	10001011 11001011
MOV	CX,CX	8BC9	10001011 11001001
MOV	CX,DX	8BCA	10001011 11001010

(...)

Look at the binary values; do you notice anything? Obviously, the first byte in a MOV reg16,reg16 instruction always is 10001011b or 8Bh. The second byte only changes a little if we alter the registers. A closer look reveals that bit 5-7 always remain 110.

NOTE: For those of you that don't know anything, the bit most to the right is called bit 0, so the bit most to the left is the last bit: bit 7. A byte consisting of 7 bits instead of 8? No, bit 0 is the 1st bit, so bit 7 is the 8th bit.

Where was I... Oh yeah, the last bits remain 110. Now it looks like the rest of the byte changes all the time, which is indeed the case. I hear the attentive reader shouting: but what about the 5th and the 3rd bit?! Well, in this example I didn't use the 16 bit registers SI, DI, SP and BP. Those ones can also be encoded, and they require that third bit.

Conlusion: 10001011 11..... = MOV reg16,reg16

Note that the first three dots are filled by bits representing the TARGET register, ie. the one to the left, and the second three are representing the SOURCE register, the one to the right.

The attentive reader probably noticed that I just explained the MOV reg16,reg16 instruction, while the MOV in the example engine doesn't need the two registers after the MOV, but only ONE register and an immediate (the starting offset). Let's take a look at that:

INSTRUCTION	HEX	BINARY
MOV AX,1234	B83412	10111000 00110100 00010010
MOV CX,1234	B93412	10111001 00110100 00010010
MOV DX,1234	BA3412	10111010 00110100 00010010
MOV BX,1234	BB3412	10111011 00110100 00010010
MOV SP,1234	BC3412	10111100 00110100 00010010
MOV BP,1234	BD3412	10111101 00110100 00010010
MOV SI,1234	BE3412	10111110 00110100 00010010
MOV DI,1234	BF3412	10111111 00110100 00010010

This time it may be more interesting to look at the Hex values than to look at the binary values. I have sorted the instructions a bit, so it shouldn't be hard to see a pattern. :) The difference with the previous example is that the 'register encoding' in the instruction is now done in the first (and only) byte of the instruction itself. The whole instruction takes three bytes: one for the instruction itself with a register encoded in it, and in the other two the immediate is stored, in Intel little-endian format.

When we further analyse this example, we see that (in the first byte) only the last three bits change every time. That's not so weird, considering the fact that 3 bits are necessary to encode 16 bit registers in instructions.

Conclusion: 10111... 00110100 00010010 = MOV <reg16>, 1234

Now it's time to list the registers and how they're encoded in

instructions. Looking at both examples we can easily see that:

```
AX = 000b = 0h
CX = 001b = 1h
DX = 010b = 2h
BX = 011b = 3h
SP = 100b = 4h
BP = 101b = 5h
SI = 110b = 6h
DI = 111b = 7h
```

All usable 16bit registers are listed here (IP doesn't count), and we can see that they all just fit in the available 3 bits. What a coincidence :) Write this list down, you'll be needing it.

The question is now, how do we tell the engine to do this? Very simple: the boys at Intel invented the OR instruction. I think many beginner virus writers do not know the exact purpose of this instruction, while for simple COM/EXE techniques this instruction is seldomly used. I think you SHOULD know that:

```
0 OR 0 = 0
0 OR 1 = 1
1 OR 0 = 1
1 OR 1 = 1
```

But you're probably wondering: for what purpose would I ever want to use this instruction? Well, what it *actually* does, is making sure that in the target byte all bits are SET (=1), which are set either in value 1 OR in value 2. And this is VERY useful for register encoding: remember our 'wildcard' for MOV <reg16>,<reg16>? No? It was: 10001011 11..... = MOV <reg16>,<reg16>, with the first three dots for the target reg16, and the second three for source reg16. Say we fill the dots with zeros, and then OR them with the encoding value for registers. Say we wanted MOV BX,CX. First we'd want to move the value 10001011 11000000 (the basic value for MOV <reg16>,<reg16>) in a register, otherwise the CPU can't work with it. We'll put it in AX:

```
MOV AX, 8BC0h   (use your bin2hex calculator)
```

Then we want to have the encoding value for the target register in bit 3-5, and the source register in bit 0-2, because the MOV instruction requires us to do so. BX=011b and CX=001b, so when we put this together we get: 011001b What happens when we OR this value with AX (which contains the value for a MOV instruction): OR AX, 19 (again use the bin2hex calculator)

```
AX = 8BC0 = 10001011 11000000
       19 = 00000000 00011001  OR
            ---------------------
            10001011 11011001      which is 8BD9, which is MOV BX,CX!
```

Mission accomplished, you now know how to encode registers in instructions.

```
========================
```
4.2 PICKING A REGISTER
```
========================
```

One of the first things an engine needs to do, is to pick a register to work with. Here we will see a big drawback of this particular decryptor layout example. In this example we are using registers to point to memory locations, that means e.g. XOR [BX],1234. Due to the 80x86 architecture, we are limited to but four registers to do that, namely BX, SI, DI and BP. So XOR [AX],1234 for example is an illegal instruction. Only those four registers are capable of this addressing mode.

There's one problem with BP though. Let me show you something:

```
81 37 34 12             XOR  WORD PTR [BX],1234
81 77 01 34 12          XOR  WORD PTR [BX+1],1234
```

In the first instruction, we're addressing with just the register, without a relative offset. In the second instruction, we DO have a relative offset, and it is encoded in an extra byte. Note that the second byte changes from 37 to 77 when we introduce the relative offset. On bitlevel, 37 and 77 only differ 1 bit, namely the sixth. This is of course to let the CPU know whether there will be a relative offset or not.

```
81 34 34 12             XOR  WORD PTR [SI],1234
81 74 01 34 12          XOR  WORD PTR [SI+1],1234
```

The same story goes for SI. When there is no relative offset accompanying the register (in other words: relative offset = 0), the instruction looks different. This same story also goes for DI, but not for BP!

```
81 76 00 34 12          XOR  WORD PTR [BP+00],1234
81 76 01 34 12          XOR  WORD PTR [BP+1],1234
```

As you can see, when there is no relative offset accompanying the register, it STILL is encoded, simply as a zero byte. Why? No idea.

What does this have to do with the engine? Well, in the decryptor in the example, we're addressing without a relative offset. For SI, DI and BX goes that there will not be a zero byte to let the CPU know, but just a bit set or unset in the instruction byte. This is a good thing, because otherwise we would have a stable byte (i.e. a byte that's the same in each decryptor generated), namely the zero. So if we wanted the engine to be able to pick registers from a list which included BP, we would have to write a routine that checks if he picked BP, and if so, adjust the instruction byte, and write the extra zero. Of course, for the purpose of creating decryptors as random as possible, this is a good idea. This, however, is an advancement to be worked on when you finished a working engine. I'm giving you a good advice here: don't try to create a complex engine right away, first start with a simple but working 'framework', which you can expand later on. If you start including all kinds of nice features, you'll probably just end up debugging your ass off.

Back to the example. I'm assuming we'll just use BX, SI and DI, so we don't have to deal with that relative offset shit. Err.. what should I explain further.. Just put the register encoding values for BX, SI and DI in a table, and let the engine choose from 'em.

Maybe you're wondering why I told you this whole story about BP and

it's relative offset. It's not really that important and specifically a problem for this type of decryptor layout, and I also just could've told you: don't use BP in this example. I didn't, because I wanted to let you know you can't just 'assume' stuff, and you have to check everything.

==========
4.3 MOVING
==========

After reading 4.1 this shouldn't be really tough I guess. :)
In the example engine, we want to have MOV <reg16>, imm16, which looked like:

10111... 00110100 00010010 = MOV <reg16>, 1234

We don't want the 1234, so forget the last two bytes. Important is that the instruction byte (the first byte) is 10111..., with a register encoding value filling the last three dots.

Example: how would we make the engine generate a MOV SI,0120 in the decryptor? We'd first have to fill the dots in the unencoded value for MOV <reg16>, imm16 with zeros, so we can OR the value to the correct opcode. From now on, when I'm talking about an unencoded value for an instruction, I'm assuming the dots are filled with zeros. Okay, so the unencoded value for MOV <reg16>, imm16 is B8h (converted in hex), let's put that value in AX. Then, we'd have to OR AX with the correct register encoding value. We wanted SI, whose value is 6h (110b) so we will perform: OR AX,6h.

================
4.4 DECRYPTING
================

Let me first of all mention once again the fact that the strategy behind most engines is that *first* the decryptor is generated, and after that the correct (matching) encryption is applied to the virus. Just to prevent some confusion.

Now, we want to make a decryptor, and we want each decryptor to be as random as possible. So we have to give the engine as much as possible crypt operations. XOR, ADD and SUB are good examples, because they have the same instruction layout. They work with 2 operands, for our purpose we will have the pointing register, and a random 16 bit immediate, so e.g. SUB [BX],1234. It's a Good Thing (tm) to include more crypt operations like NEG, NOT, INC, DEC, but the disadvantage of these other types is that they have other instruction layouts, and therefore require separate routines for including in the decryptor. If you decide to include more crypt operations in your engine, these four are a good choice, because they all require one operand, and for our example engine that would be a pointing register, e.g. NOT [SI]. Let's just say our example engine uses only XOR, ADD, and SUB.

Again, to build an instruction, we need to know the unencoded value for the instruction. Assuming the dots (where the register encoding should take place) are filled with zero-bits, here are the values:

```
XOR:    81 30 xx xx
ADD:    81 00 xx xx
SUB:    81 28 xx xx
```

Please note that these are the values for use with *pointing* registers! So with these values, you would be able to produce an instruction like XOR [SI],1234 but not XOR SI,1234 ! We need pointing registers for our example engine, so the above values will do.

Another important note: the first byte is 81, no matter what crypt instruction the engine picks! That's not good, because the whole idea of the engine is that we wouldn't have 'stable bytes' in the decryptor. So you can see you simply cannot do with just these three crypt operations, simply because it'll create a stable byte. As said, it is wise to ignore this for the time being, and first try to create a working engine. Then you can start working on problems like these. Well, you actually *need* to work on them, because if you leave stable bytes in your decryptor, you still haven't disabled the possibility for scan strings.

Okay, back to the program, say the engine picked ADD to include as crypt operation. We can simply store the 81, while it doesn't need any register encoding. Then we have to encode the correct register in the second instruction byte. Earlier, we encoded just the registers in the instruction, and now we need to encode pointing registers in the instruction. I haven't given you the encoding values for the pointing registers, so here they are:

```
000b = 0h = [BX+SI]
001b = 1h = [BX+DI]
010b = 2h = [BP+SI]
011b = 3h = [BP+DI]
100b = 4h = [SI]
101b = 5h = [DI]
110b = 6h = [immediate]
111b = 7h = [BX]
```

You should see a possible advancement: addressing using two registers. For example, set BX to zero, and let SI be the pointing register, but now use [BX+SI] instead of just [SI] to address the correct byte/word. More randomness!

Okay, I think the rest is pretty easy. OR 30h with the correct register encoding value, store the byte, and store a random 16 bit immediate. Don't forget to 'remember' which cryptop the engine stored, and which immediate it used. Most engines create an encryptor along the way, which can be executed when the decryptor is generated.

This is only one crypt operation. Uhm.. like, just repeat this step a couple of times, depending on how much crypt operations you want in your decryptor.

4.5 INCREASING

Now it's time in the decryptor to increase the register. Since we're

using word encryption, the register needs to be increased by two. Obvious instructions are two INCs; ADD reg, 2; or maybe even SUB reg, -2. But also consider string instructions like SCAS, CMPS, STOS, LODS and MOVS. They perform an action, and then increase SI, DI or both. That could suit our purposes quite well. However, be careful with these, as for instance MOVS may have some unwanted results. I think you should be able to find the correct values for these instructions. To make this tutorial not too longwinded I will only shortly explain the use of the two INCs, as you now should be able to figure out the rest yourself.

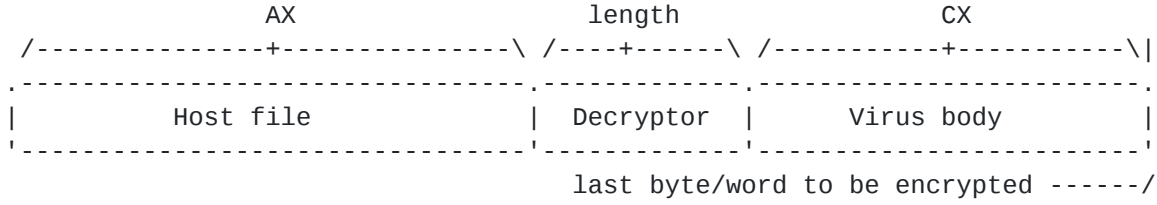
The unencoded hex value for INC is 40hex (hi P/S) and you should encode it with the normal register encoding values (not the *pointing* registers, because that would be a crypt operation! (compare INC BX and INC [BX])) to get the necessary result. E.g. INC SI would be OR 40, 6. With 6 being the register encoding value for SI. This should be clear by now :)

==========
4.6 COMPARING
==========

Of course the decryptor has to know up to which offset it has to decrypt. In this example the only possibility of doing this is a CMP instruction, which is pretty weak because it gives the decryptor a stable byte. Another way to do the same thing would be SUB, but when SUBtracting the register you're using again in each loop, it will be destroyed. A way of doing this could be giving another register the end offset each loop, and then subtracting that register from the main register (used for the actual decryption). It's just an idea.

For this example we'll stick with the CMP. The unencoded value for CMP reg16, imm16 is 81F8h, 81 being the instruction byte which needs register encoding. So we can store the 81, and OR the F8 with the normal register encoding values (again *not* the pointing register values), and store the result.

After that, we'll have to store the imm16. This value is the last byte/word that needs decrypting. How do we calculate this value? Rather simple: starting offset + code length. Code length is passed to the engine using CX, and the starting offset can be calculated by memory offset + decryptor length. Memory offset was again a parameter for the engine, namely AX, and the decryptor length shouldn't be hard to calculate. If your decryptors are always the same length it's easy, but when you have variable length decryptors, you should keep track of how many bytes you're storing. So:



AX + decryptor length + CX = CMP operand

Pretty easy huh. Add the values, and store it as operand.

==========
4.7 JUMPING
==========

After the CMP comes a conditional jump. The obvious instruction would be JB, but that again introduces a stable byte, because we don't have a substitute for it. Yeah, JNA would be nice, but the problem is they have the same hex values, which is not so weird considering that they do exactly the same thing. What about JNZ? This could be interesting, but only if we'd add one/two to the CMP operand! Otherwise the last byte/word wouldn't get decrypted! Again, I'm assuming we're only using JB. (Note that IF you choose to use the JNZ feature the following explanation also sort of applies to it)

If the condition is met (condition = we're below the last byte/word) the jump should jump back to the first decryption operation. How do we 'encode' this? First of all, JB has 72h as hex value, so we can store that. As you all should know from COM infections, such a jump has a displacement byte right behind it. This displacement byte is the amount of bytes the CPU should move IP, where you should see the starting point at the offset of the next instruction (so 2 instructions behind the 72 of the JB). Here's an example:

```
0BF9:03B6   81 37 64 23         XOR [BX],2364
0BF9:03BA   (...)               (...)
0BF9:03C6   81 FB A4 06         CMP BX,06A4
0BF9:03CA   72 EA               JB 03B6
0BF9:03CC   (...)               (...)
```

In this example the first decrypt operation starts at offset 3B6. The "next instruction" of which I was talking would start at 3CC in this example. So the correct calculation for the JB displacement would be:

```
New offset  -  Offset of next instruction  =  Displacement
   3B6      -           3CC                =     FFEA
```

As you can see, we get FFEA as displacement. We can only use one byte, so that would be EAh. Now most of the time you would say that EAh = 234d, but there's also something to say for EAh = -16d, and that's exactly the amount of bytes the CPU should move. Note that you don't have to worry about things like the memory offset, because it doesn't matter where in memory this is located: the displacement will be the same every time, as the amount of bytes to move doesn't change.

==========
5. ENCRYPTING
==========

At this point we're done with the decryption loop. Along the way I was telling you your engine should 'remember' what it did. Sounds okay, but you're probably wondering how that is done. First of all, it is wise when making a table with crypt operations, to include the inverse crypt operations. E.g. store SUB along with ADD, ADD along with SUB, and XOR along with XOR.

In that way, you can store the decrypt operation in the decryptor, and the inverse crypt operation (for encrypting) in the encryptor. Note that the first instruction in the decryptor should have it's inverse operation as *last* operation in the encryptor.

==========
6. RANDOMIZE
==========

Throughout this tutorial I told you to let your engine pick something at random. Most high level languages have functions which return a 'random' value, but, as usual in assembly language, we get to do it ourselves.

As there is no way you can actually make a computer return a totally random value, because it just can't do that, we have to build a routine that gives us a value as random as possible. The most used way to get a random value is reading from port 40h. Because the value isn't really random, it's wise to perform some calculations, which would randomize the number some more. Another possibility is to let the decision depend on a value like system time or date. BTW: when you manage to do this correctly, you'll get a form of slow polymorphism.

Experiment with calculations and write a program which outputs the random values to screen *on bitlevel*. Investigate the values to check if they really are quite random. Note that this IS quit important, and if you rely on a bad randomizer, you'll end up with some options of your engine getting disabled, as they're never reached.

Once you have a random value, and you want to use it to decide e.g. which cryptoperation you will use, you have to AND it out to e.g. the offset of the last value of the table. (If you don't understand what AND is: AND makes sure that in the result byte only those bits will be set, which are both set in the first operand AND in the second operand. If you still don't get it, re-read the part about OR, and investigate AND in the same way, i.e. try to get to know what it REALLY does.) So, if you have 5 options, AND the random value out to 4, so you can add it to the starting offset of the table. Don't AND it out to 5, because the starting offset + 5 is just behind the table. This is because at offset 0 there is a value too. So, if you have x values, AND the random value out to x-1.

There is one problem though, which is overlooked by some writers. Get your hex2bin calculator, and start ANDing random values with 4h. Say you do it 100 times, you'll only get either 4 or 0 as a result. You'll NEVER get 1, 2 or 3. Hmm.. weird.. Let's try it with 5h. Perform AND xx,5 with 100 values, and you'll get 0, 1, 4 and 5. But NEVER 2 or 3. To understand this problem, we have to look at the values we're ANDing with at bitlevel:

```
0h = 00000000b
1h = 00000001b
2h = 00000010b
3h = 00000011b
4h = 00000100b
5h = 00000101b
6h = 00000110b
```

7h = 00000111b.

When ANDing with a value, bits in the result byte will only be set when they're also set in the value you're ANDing with. Say you're ANDing with 5h, and you'd want to get 3h as a result. 3h equals 10b so that means that in the value we're ANDing with, bit 1 should be set. Because the value we're ANDing with is 5h, and that equals 101b, bit 1 can NEVER be set. So you'll NEVER get 3h as a result. Again, this is quite important, because if you choose to neglect this problem, you'll end up with some features of your engine simply being disabled!

I hope you already 'feel' that values with all (necessary) bits set are suitable for ANDing with, as the result of the operation varies from 0 to the value you're ANDing with, which is what we want. So 1b (mostly unnecessary), 11b, 111b, 1111b and so on, are values suitable for ANDing with. If you have too few options in your table and can't think of another option to fill the table with (so you can AND with one of the mentioned values), you could simply put duplicate options in your table. This isn't as stupid as it may seem, as some options are more flexible than others: e.g. when deciding whether the engine should use normal INC or a string operation (LODS, STOS, etc.) for increasing, the option for "use a string operation" is attractive for duplicating in your table, as it has more sub-options (later on, the engine has to decide *which* string operation it should use) than the "use an INC" option.

Please note that this is just one way of working around this problem. You should be creative enough to think of another one, although I doubt it being more effective considering size.

==========
7. SIDENOTE
==========

One more important note. When working on your own engine, you're probably planning on adding some nice features. And when you're including new instructions, you need to know the unencoded values of these instructions. If you're not instantly running SoftIce in the background it's probably very tempting to use DOS DEBUG to check out the values. Never do that.... There is something very strange about the 80x86 of which I don't know the explanation: some instructions can take two different identities. E.g. the instruction "OR AX,AX" can take "09C0" as 'identity', but "0BC0" gives us the exact same instruction.

All 'official' assemblers like Borland TASM, Microsoft's MASM and all others, all pick the same of the two choices they can make, namely the 'highest' (so "0BC0" for "OR AX,AX"). In other words, if a certain instruction takes an identity which normally wouldn't be produced by regular assemblers, it is very likely this is the work of some polymorphic engine. Some (or most?) antivirus programs know this and alarm if they find such an instruction. I recommend using SoftIce anyway (although it's assembler also has some bugs), but if you for some strange reason don't or can't, use Borland's TurbodeBugger to check the values, I guess that should do the trick too.

A sidenote to this sidenote would be something PM pointed out to me:

the still ever popular A86 shareware assembler has a nasty side-effect to it. At times, it seems to pick the 'wrong' one of the two possible instruction codes, resulting in the same alarms being produced. This can easily be avoided by using the Real Thing (TASM). There's no reason for not using TASM anyway.

8. GARBAGE

Garbage instructions are instructions included in the decryptor, which serve no real purpose for the execution of the decryptor, but are only included to hide the so obvious real purpose of the decryptor. Without any garbage, the decryptor actually *looks* like a decryptor, which makes it very easy to spot it heuristically. Furthermore, garbage instructions will let the 'real' instructions be at random places (in a constant order of course). In one generation the increase and compare operation are right next to each other, while in another generation some other (do-nothing) instruction is included between them.

The classical concept of garbage instructions consists of including one-byte do-nothing instructions like NOP, CLC, STC, INT 3 (the only one-byte interrupt (CCh)) between the actual decryptor instructions. Nowadays, the use of these one-byte instructions has become rather trivial, as almost all AV programs just ignore these instructions when they encounter a decryption loop.

These one-byte do-nothing instructions have also lead to something AV calls variable scan strings. In these scan strings, it doesn't matter if you have zero or thousands of garbage instructions between two constant values, the AV program just ignores those, and just checks if the two constant values occur in the right order (so, first the increase, then the compare instruction, here assuming these are stable bytes in the decryptor). In other words, these one-byte garbage instructions really serve no purpose anymore.

Furthermore, AV has noticed that huge amounts of these kinds of garbage instructions in a relatively small area are very likely to be the results of a polymorphic engine. In other words, using these one-byte do-nothing instructions can result in heuristic flags being triggered.

Luckily, there are other ways of producing 'garbage code'. The basic idea behind all of them is producing code which COULD be code required for the program to run correctly. A nice example is including some stupid Int21 call, like GetVersion or something. There are a lot of possibilities here. Each time take two things in account:

- (*) Whatever garbage you use, make sure it doesn't influence the execution of the decryptor, but also make sure it LOOKS like useful code
- (*) When using large garbage constructions (like the setting up of an Int21 call), make sure you have enough variations, otherwise AV could make a scan string (or some more if that would be enough) just from your garbage code!

Things to be considered are the possibility of including some nice armor tricks in the garbage construction (I'm calling it construction because

sometimes the garbage doesn't consist of just one instruction anymore), but again, make sure to have enough variations, because armor code is of course uncommon code, which AV loves, because they can make scan strings of it!

For those wondering how to include garbage instructions/constructions at random: just place a call to your insert_garbage_at_current_offset before each major decryptor operation. An extremely simplified example:

```
start:
        call    maybe_insert_garbage
        call    set_up_registers
        call    maybe_insert_garbage
        call    perform_move_instruction
        call    maybe_insert_garbage
        call    do_first_crypt_operation
        call    maybe_insert_garbage
        (...)
```

9. MULTIPLE DECRYPTORS

A virus writer which for some reason doesn't want to include a complete engine, but still wants *some* variation in his decryptors may consider the use of a couple of constant decryptors. They all do the same thing, but they look different. Here, the variation isn't on instruction level, but on routine level, which is far more simpler to implement. This technique can be called polymorphic in it's true meaning (poly=many, morph=shape), but as just a few routines are generated, it's often called oligomorphic (oligo=few). On it's own this method isn't considered strong, as a set of scan strings can be used to identify every possible generation.

The combining of these two methods however gives a very large variety of both decryptor layouts and decryptor instructions, which can be considered strong polymorphism. Most of the more powerful engines combine the two methods, and are therefore very hard to detect reliably.

Throughout this tutorial I have tried to explain how to avoid constant values using variation on instruction level. If you want to extend your engine with variation on *routine* level you must use well-considered code. The engine should decide which decryptor layout (routine) it will generate, and then it should generate that particular routine. As some things have to be done in more decryptor layouts (e.g. increasing the register) it's often wise to make routines (for e.g. increasing the register) which are independant on from which decryptor-layout setup they are called from. So the increase_pointer routine (which on his turn chooses from a variation of instructions) must be able to be called from different layout setups. (hope this was clear.. :))

10. CONCLUDING

I hope I have given you a basic idea of writing polymorphic engines.

Once you have written one you have a pretty good idea about polymorphism and you should be able to enhance it with more Fancy Features.

Finally, I'd like to thank Case/IRG for explaining some stuff to me when I was starting out with polymorphism, and not to forget PM, who was willing to test the article.

If you like this tutorial (or have any questions for that matter), feel free to contact me at trg-@usa.net, on IRC #virus, or contact another SLAM member. Feedback will encourage me to write more tutorials. By the way, if you hate it, I also wanna know :)

Trigger [SLAM] '97
