# The Complete Re-write Engine



## VX Heaven

MidNyte
*Final Chaos [1]*
*May 1999*

## Introduction

First of all, let me explain a Complete Re-write Engine (CRE) as I see it. I have thought of it as something in the 'would be good if it could be done' category, and is a form of metamorphism. A CRE would read in a piece of code, and work out a (random?) functional equivalent, that is, produce a new and different piece of code that would produce the same results. This has been talked of as a replacement for encryption in viruses. To write a CRE you will need at least a basic understanding of polymorphism, emulation and disassembly. You need to understand how the opcodes are formed to be able to manipulate them properly.

Also, let me explain that if you're looking for a step by step guide to writing a CRE, you're not going to find one here, this is only a discussion of the main problems of creating one. I am certainly not of the coding ability to create one yet, if one is even possible in practise. If you think you can do one, go ahead, metamorphism is a unique alternative to polymorphism, but don't think this is going to teach you how to do it. This is just my explanation of how I plan to do it when I do have the ability.

The algorithms, methods and code produced by the end of this article will be messy and awkward, but hopefully functional. The price to pay for code that works is often code that's neat, at least for those who do too much writing and not enough coding like me. Also, please note that the second and subsequent generations will be very difficult to read and/or debug, because (obviously) the re-write engine will re-write itself along with the rest of the virus. Any bugs in a later generation must be traced back to their creation, but this may be through a buggy routine produced by a buggy routine etc.

The practical values of a CRE are that no scan string can be made from the code, as every part of it can be changed from one generation to the next. Most heuristics are often defeated by simply changing from obvious ways of doing things to an alternative, so it should be possible for a CRE virus to evade them. The disadvantages are that some (most?) forms of anti-debugging are code specific, so might be difficult or even impossible to incorporate (the stack

checking trick for instance), and that keeping data within the CRE virus can be tricky. Anyway, It's up to you to evaluate the usefulness of CRE. My personal opinion is that it will only ever be a proof of concept thing, and that the limitations to it will always outweigh the advantages. I'm hoping someone can prove me wrong. Anyway.....

## The First Step

The first step in any major programming project is (or at least should be) planning the algorithm. Our CRE would examine each instruction in turn and work out an equivalent instruction or series of instructions. This alternative generating is much like conventional polymorphism.

My approach to writing the algorithm would be too write the basic virus, then write the instruction handlers for every instruction in the virus. Then when that's done, go back and add handlers for every instruction that's in the instruction handlers we've just written. This should require no new instructions to achieve, but if it does, make a note and write handlers for them aswel. This is going to take time.

So our basic model is: (in pseudocode & assuming all instructions are one byte each, this is only a demonstration.)

```
START:
   <get byte>
   <X = 1 to number_of_instructions>
   <is byte a type X insruction?>
      Y - jump to type_X_handler
      N - next X
   <else jump to data_handler>
   <all bytes done?>
      Y - exit
      N - get next byte

TYPE_X_HANDLER:
   <Y = random number (1 to number_of_type_X_alternatives)>
ALTERNATIVE_GENERATOR:
   <generate alternative Y>
   <return to start>

DATA_HANDLER:
   <copy data>
   <return to start>
```

Ok, by now you've noticed a few obstacles to overcome and things that need to be explained more fully before we can move on.

First, data is data and cannot be changed without changing the functionality of the code. This can easily be overcome in some cases however by loading the values into registers (ie, cmp ax,80h could be cmp ax,bx instead if bx was first loaded with 80h). Some cases are more difficult, like having a search string of '*.com' in amongst our code for instance. One possible

way to deal with this would be to store the string encrypted in a certain way. This series of instructions (the decryption method) could then be handled by our CRE. This does however lead to the difficulty of keeping track of where the string is within the file, but this can be done by pointing to the address of the string from a set location. As the string will always move, this will not create a stable byte. Or we could just put all the strings/data at the end of the file and read them from there, using the filesize as a base reference to find them all.

Secondly, what if a data byte is the same number as the ordinal value of an instruction we scan for? well, this is an inherent problem to the CRE concept (and emulators and debuggers), but a quick work-around is to check each piece of randomly generated data against a list of 'forbidden values', ie, our instruction set, and choose another if it will cause a conflict. This does unfortunately need more data to be stored/created. Maybe those who have knowledge of emulation systems could provide a better solution (hint hint), but at least in the meantime we have a work-around.

(What if the pointer mentioned in the first point is the same number as an ordinal value of one of our instructions? I guess we'd have to use two pointers added together to create the address of the string to be safe, then at least we could keep altering them until neither contained a hidden value.)

Thirdly, (not really a problem) our instruction handlers need to know what bytes that come after the instruction byte are data. This is simply a matter of knowing the instruction's layout, then loading the stack or registers with the appropriate numbers ready for use by the alternative generator. This is where metamorphism differs from polymorphism quite significantly. In polymorphism the code generator knows what values in has to attach to instructions, in metamorphism it has to work it out for itself from code that it has previously written.

Fourth, with all this generating of numbers instead of storing, aren't we going to get longer each generation? Without a doubt. Every piece of code in the virus will be taken up by an equivalent of at least the same length, probably more. Each piece of this new code will be replaced in the next generation by an equivalent of at least the same length, probably more in the generation after that, and so on. This would mean an exponential increase in the size of the code. This is why the first advancement on the basic system is:

## Code Shrinking

Code shrinking is the replacement of a long set of instructions with a shorter set of an equivalent function. Our instruction handler is going to have to be able to recognise instruction sequences that can be shrunk in the first place, so how about scanning the next 20 or so bytes for particular sequences relevant to the instruction we're on? The CRE would produce most random code if the shrinking does not always shrink as much as possible. Maybe you could give it a percentage chance of shrinking and play with that value until the optimum is reached. If we get this part right, our code should grow from it's first generation

(because it's in it's simplest state when it's written) until it reaches an average size that it fluctuates around (when the shrinking it does roughly balances the increases it makes). If it doesn't become stable, then either your alternatives are too lengthy, your shrinker isn't good enough or both. If the fluctuation in size is too drastic or just not to your liking, you could always bias your shrinking dependant on the size of the code, ie, more shrinking when the filesize is high. As long as the biasing is not too strong it should calm down the fluctuation too, brush up on your chaos theory if you want to know why. Chaos theory also tells us that too strong a bias may well lead to a predictable oscillation by the way, and the last thing we want is anything predictable. Anyway, We now have:

```
START:
   <get byte>
   <X = 1 to number_of_instructions>
   <is byte a type X insruction?>
      Y - jump to type_X_shrinker
      N - next X
   <else jump to data_handler>
   <all bytes done?>
      Y - exit
      N - get next byte

TYPE_X_SHRINKER:
   <push si>
   <Y = 1 to 20>
   <read next byte>
   <is byte related to instruction X>
      Y - SHRINKABLE=TRUE ; save shrinkable byte to mem
      N - ignore
   <loop Y>
   <is shrinkable=true>
      Y - jump to shrink_X
      N - continue
   <Y = random number (1 to number_of_type_X_alternatives)>

ALTERNATIVE_GENERATOR:
   <generate alternative Y>
   <return>
SHRINK_X:
   <calculate alternative from saved bytes>
   <return>

DATA_HANDLER:
   <copy data>
   <return>
```

So, now we have a structure to work on. This is the part where each instruction must be dealt with individually. You'll notice I wrote 'is byte related to instruction X' in the above pseudocode. Well, that's up to you to decide how all the instructions are related, or more accurately, up to your alternative generator. For example if your CRE's only alternatives to 'MOV AH,80h' are 'MOV AL,80h / XCHG AL,AH' then for your MOV_SHRINKER you will

only have to check for the MOV and XCHG instructions. If you also have the possibility of 'MOV AH,40h / ADD AH,10h / ADD AH,10h / ADD AH,10h / ADD AH,10h' then you'll have to check for ADD's aswell.

One thing to watch though, is that every alternative that you make that increases the amount of instructions should be recognised and handled by your shrinker. If you don't, it can happen that your code makes an alternative that the shrinker doesn't recognise, and then that particular bit of code can never be as small as it used to be. This makes the code less random and larger, not a real problem for just one instance, but obviously the more it happens the more it becomes a problem. This will cause a particular problem if a longer replacement for a common instruction that your generator makes isn't recognised by your shrinker and if the replacement contains similar instructions to the original (ie, making 'ADD AX,40h' into 'ADD AX,20h / ADD AX,20h' in one generation, each 'ADD AX,20h' in the next becoming 'ADD AX,10h / ADD AX,10h'). This would create a sequence that could only ever get bigger as generations went by, each one expanding the sequence but never shrinking.

Other advancements would be to have your alternative generators or shrinkers be aware of consecutive instructions, ie, 'MOV AH,80h / MOV BH 40h' could become 'MOV AH,80h / MOV BH,AH / SHL BH,1', or you could interleave the equivalents of consecutive instructions, ie, 'MOV AH,80h / MOV BH 40h' could be 'XOR AH,AH / XOR BH,BH / ADD AH,80h / ADD BH,40h'. I have already incorporated a mechanism for skipping unrelated instructions (ie, searching ahead for relevant instructions, not just assuming that they're the next ones you come across), but haven't put in any way of ensuring that the same instruction is not taken by the shrinker to be part of two instruction sets, ie, using it twice. I'll leave it to you to sort all that out, and I'll concentrate on learning to code properly while you're doing all that.

Do be aware that any complications to the layout of instructions your alternative generator makes, again your shrinker should be able to handle if you don't want your filesize to (sooner or later) increase out of control.

## Conclusion

There we are then, I've just told you what needs doing, admitted couldn't do it, and now I'm sitting back waiting for a CRE with a big credit to me to appear..... no, seriously, a CRE is possible in theory as long every instruction can be emulated by using other instructions, in practice though there are a lot of problems. Whether or not these can be overcome enough to create a fully functioning CRE-equipped virus or not is another matter, especially if the practical value of this idea isn't much in the eyes of the reader. Personally, I see polymorphism as a middle ground (although more than adequate, providing the encryption is strong) between a simple encrypted virus and metamorphism, so a CRE should at least be

created and tested. I will eventually learn enough to do one, and I will do one, but until then I don't want to be keeping these ideas to myself, just in case anyone finds something useful in all that. Good luck!

```
- MidNyte
```

As always, I welcome ANY feedback, good or bad, as long as it is reasonable.