Tutorials - Resource Based Polymorphism [rbp]

> [This is an unfinished product (CCTX), author's quote:".....just a bunch of ideas really just for my own use. Half the stuff contradicts each other, I added on to it as I came up with new stuff so it doesn't make a lot of sense and its pretty much just another idea put on the scrap heap or at least on hold.....]

This document currently holds a bunch of ideas based on 'resource-based' polymorphism, some ideas mentioned aren't very efficient and have been surpassed by other ideas yet they still exist in this text. 'Offset-Based' polymorphism is the best method so far that I've come up with in the area of 'Resource Based' polymorphism.

This is an idea I have been thinking about for a while, but have only recently put much thought into. Originally, this idea was dubbed 'Linked list polymorphism', however, with some thought, I realise that the method could be applied in many other ways which wouldn't necessarily involve linked lists, as a matter of fact, the best way I can think of applying it doesn't involve linked lists at all. I then dubbed it 'clean-slate' polymorphism, due to its nature, but then I realised it can be implemented even more ways, and decided on simply 'resource-based' polymorphism. The key to this new form of polymorphism is in mutating the entire code rather than just generating decryptor/encryptors. It is similar to Rajaat's (29a) idea in that it doesn't use decryptor/encryptor pairs, but instead mutates the code, however it achieves this without the use of an emulator, or the need for a 'shrink' engine, which was the major stumbling block to this original idea. It is also similar to my own now discontinued 'HOPE' engine in that it heavily uses tables, linked lists, control codes and parsing etc, however as stated before, it mutates the entire code rather than just generating decryptor/encryptors. It is based on keeping a 'clean' copy or reference of the virus and engine, however, in a mutated state.

One possible method, but not the best, of mutating this 'clean' copy of the virus+engine is through the use of linked lists. A linked list is essentially a structure which 'points' to the next structure, which in turn, 'points' to the next... This enables the items to be 'physically' in any order, yet still logically in order. They are similar to arrays however they have both advantages and disadvantages, the advantages far outway the disadvantages in this case however and they are very useful.

- Items can be dynamically created and inserted into the 'chain'
- Each item can be 'physically' located anywhere, yet still be in order
- Each item can be of variable size
- Disadvantages:
- Each item must also contain a pointer to the next
- Items can't be directly indexed, and you must step through the entire chain

The disadvantages aren't that much of a problem, it means an extra word or byte for each item, which although means a lot of wasted space, it also means fairly thorough polymorhpism. As in this case the items will be accessed sequentially, the second disadvantage isn't much of a problem. If you think about it, linked lists enable us to have ordered randomness, a very useful thing indeed for a polymorphic engine. We can have items in the linked list in a random order, and yet they can still be accessed in the original order, through the use of chaining and pointers. This should all be fairly obvious, however, it is with this idea that 'Linked List Polymorphism' is based on, so you should have a good idea on what it is.

The idea behind the engine is to store a duplicate or reference of the entire virus code plus engine into a table or linked list, which can be mutated yet still be readable by the engine. Ideas like this have been made before, but there has always been the problem of somehow getting further than 'first generation' without any static bytes, as it is necessary to somewhere keep a 'clean' copy of the engine and virus to mutate it from. This has been accomplished however with 'linked lists' or encryption of the table. Obviously, it is necessary to duplicate this table of code and data into each generation, and in most other engines this would remain constant. However, with the features of linked lists, we are able to also mutate this table so that there are no constant bytes within the code.

The engine looks at this linked list, going from each _logical_ item to the other, adding the code contained within the block, and in between each block of code added, some random junk is added which obviosuly must have no affect on the surrounding code, including memory, registers, flags and everything else. This pretty much limits the junk code to just 'NOP's as most other operations modify registers or flags. However if you code your block carefully, it would be possible to have more complicated junk which modifies flags etc, by making sure no junk is inserted between a comparison and a conditional jump, this can be achieved in two ways, either by coding the instructions into one block, or by using control codes to specify that no 'flag-modifying' junk be inserted after the block. eg:

Which may generate unwanted junk between the two instructions, you would code just one block of:

Which would be inserted wholly with no junk inserted between. You may have noticed the problem of offests at this point, I will get to that later. Instead of just inserting random junk between instructions, its also possible to do such permutations on the code that are available to my 'HOPE' engine, such as changing opcodes and so on. Obviously you couldn't use random registers etc, without highly complicating the engine, in some cases however this may be possible. Instead of having each block containing a single instruction, it could contain multiple instructions, or variations of the same instruction, to be randomly chosen, permutated, and used. This would require an extension to the engine to process special 'control-codes' for each block/item. Once again, take a look at my 'HOPE' engine to see one method of how to achieve this. It would be possible to add many different features to the engine, to allow for highly polymorphic viruses, there are however some points to remember.

The entire engine plus virus code has to be duplicated into the linked list, with all the chain information plus control codes if they are used. This amounts to an extremly huge amount of data, the HOPE engine alone is 3k or so, this would roughly triple and much more if it was to be converted to an engine such as this. Also, the larger the blocks/items, the more chance of a scan string to be created. Although the block itself can be relocated anywhere within the linked list area, the actual block itself with control codes etc doesn't change, it is possible to change it however, this would take yet more code. For example, a block with:

Which when read and parsed in to re-create the code, could be converted to a variation yet again for the actual code, still without the requirement to have a 'shrink' engine as the junk code is never contained within the linked list of raw, clean code. This would complicate the code even more, however it is theoretically possible. With just small blocks and no control codes, this probably isn't necessary, as the scan string would have to be small and able to be located anywhere, which would cause a lot of false alarms. So the idea is to keep the blocks as small as possible, preferably as small as just one instruction. If it is necessary to have blocks of more than about 10 bytes, its probably advisable to also mutate the actual blocks within the linked list themselves, unless you can keep the blocks as generic as possible so that they would be likely to also exist in other programs. Another option would be to mutate the blocks only on parsing to the virus code and engine, but then compressing the linked-list. This is a very attractive option, as it not only shrinks the very large list, but the compression would hide the larger blocks, they would exist differently each time because the list would be in a different order. The only option would be to decompress the list, then scan for a block within the decompressed list, which could exist anywhere. This is an unlikely option for an anti-virus scanner, they would be more likely to attempt to detect you virus using 'x-ray' technology on the actual permutated virus code, which is far from fool-proof.

To permutate each block, one would probably use technology similar to the 'HOPE' engine, which uses 'control-codes' heading each block, indicating where the code may be changed with each block. An example may be:

Obviously, this would only be necessary if it was compulsory that these instructions be placed together with no junk inserted in between, a block wouldn't normally be this large, only the inc ax/jz label would need to be in the same block, however it demonstrates how to mutate a block if it needs to be this large. An example may be if you have a string which you want to randomly change to uppercase or lowercase. eg.

A better alternative to the previous block would be to include codes specifying what kind of junk may be inserted after each block, so that blocks doing comparisons don't have flag modifying junk inserted after they are parsed in, because we know the order in which the blocks will be added in, we don't have the problem that I faced with 'HOPE' in which what block gets added afterwards was completely random. This would mean blocks can be as small as one instruction, comparison / conditional statements won't be affected. I have already

demonstrated the use of 'control-codes' with my 'HOPE' engine, so it is quite possible. Therefore, the best method I can think of is to head each block with just one 'control-code' byte, which should indicate something like the following:

This is just one example of a control byte that could be used, it would satisfty most engines needs. The "Further control bytes follow" flag would indicate a secondry control byte, or more, which would specify yet more options if necessary. I personnaly wouldn't use more than one control byte as it would complicate the engine a great deal if anymore were added. It would be better to use the 8th bit for something more useful, for example, the bits 5,6,7 and 8 could be instead used to help the engine recognise the instruction in the block and permutate it, otherwise the engine must recognise the instruction and know how to possibly modify it.

Obviously the use of 'linked lists' is just one option of mutating the 'clean' code, another option is to simply use a table that is encrypted using a variable key. This wouldn't be able to be decrypted by AV scanners as the decryption code would only be called when the virus attempts to replicate. Use of encryption or compression however has its drawbacks, which are quite significant. By not encrypting the 'clean' code, the virus can load itself into memory and run the engine etc without having to allocate space to store the encrpyted data, and having to somehow re-encrypt it when replicating, and also by using linked-list mutation, the entire virus is mutated without relying on protection such as previous polymorphic viruses used, which was overcome. The advantage of this type of polymorphism is that the virus can duplicate itself, generate new code easily and shuffle the tables, without ever having to have a decrpyted form of itself as it is all fully functional code. This means the possibility of slow polymorphism is quite easy, as the virus may choose to only mutate itself when loading into memory, then all infections can just be mirror images.

Another option for so-called 'Resource Based Polymorphism' is through the use of emulation, similar to Rajaat's method. There are at least two possibilites, either by keeping an encrypted 'clean' form of the virus without all the header information, and recognising each individual instruction through emulation, or a much shorter way, keep a list of offsets to the 'real' instructions in the actual permutated code, instead of an entire duplicate copy. This is probably the best method so far. The offsets can be shuffled around, then when its time to generate the code, it is read in logical order by going from lowest to highest eg..

The engine would seek through the offset list and look for the next largest offset to generate the code, there are problems with this method however. The emulator must be able to recognise instructions that need to be specially treated, ie, instructions that can't have 'flag-modifing' junk inserted between them and the problem of relocating offsets. This can be solved however in a similar method as before, and also can do away with the use of emulation and several other problems. Each offset need only be headed by one or two or more 'control-codes' that indicate the type of instruction and length. eg..

This means that each instruction need only have a 3 byte entry in the table. The control-code byte would probably take on something similar to, but not necessarily anything like:

This would allow one byte to indicate a maximum of an 8 byte block, of course more than one could be linked together to indicate anything larger, by specifying that no junk be inserted afterwards and adding another entry to the table. The problem with this approach however is that the tables must be modified on the fly to indicate where each instruction is placed and the new length if it is to be changed. Also, if instructions are changed at all, we still have the problem of expansion, but it is not as big a problem as with Rajaat's original idea as it is still only working with the actual code and not the junk. The best method may be to not expand instructions at all, but only modify where possible, and add junk. The second code example below gives a demonstration on sorting through a list to parse into logical order.

This is covered by another table or linked list, although there are alternative ways of solving this issue. This table of items would also need to be 'shuffled' or encrypted or both in some way so that it doesn't reamain constant.

Everytime a new item is parsed in, the relocation tables are searched and the appropriate offsets added to the items in the table, so that when finished the code can be modified appropriately. This table could also contain extra information for 'special' entries in the main list, such as the item containing the offset to the first item in the chain or other 'special' sections in the code which need to be handled differently to other items. Basically it is a way to add extra 'control-codes' to each block without wasting space.

Example code for linked list polymorphism (this is not the best method for resource based polymorphism, see below for a better approach)

Some example code for 'offset based polymorphism', this is a far better approach to 'Resource Based Polymorphism'. This is only example code, however it can be adapted further to provide full polymorphism.

The idea of the engine is to strip the old 'actual' code from each replication, and generate a new polymorphed version from the lists. See the diagram.

The lists are then duplicated and appended to the new virus+engine, however they are 'shuffled' around and modified so that they cannot be scanned. This provides full polymorphism to the entire virus+engine+list.

When creating any virus, its important to understand how the virus may be detected by the AV, by viewing the code from a critical AV perspective. One possible method the AV may use to detect a metamorphic virus is to use 'x-ray' technology to filter all your junk code then construct a scan-string based on whats left. For example, take a look at your mutated code and strip away all code except instructions such as, stosb, movsb, int, cmp [si], 'MZ', all memory writes etc.. you will nearly always be left with the same code, or very similar code, which a wild-card scan string can be constructed from. This highlights the importance of

complicated junk which does pretty much all that your virus does, so that the AV will be left with nothing to scan with. It also suggests that you should code your virus using as generic procedures as possible, for example using loops instead of rep's, as 'LOOP' junk can more easily be generated as junk than rep opcodes. It also highlights the importance of op-code/instruction changing and address relocation, with this, the AV are left with less to scan with, meaning either less detection or more false positives. You should remember that they can only filter these instructions using emulation, so only the first part of your virus will be susceptible to this attack, so concentrate your mutation/generic effort there more than anywhere. With this in mind, the 'link- list' approach becomes more attractive, as it can provide alternative routines to 'compile' in, so there is more variation in the 'stripped' down virus. For example..

```
Block35:
        DW Offset to Block36
        DB 3                    ;3 alternative routines to choose from
        DB length of routine 1

        mov ax, [offset]
        cmp ax, 090
        jz label

        db length of routine 2

        cmp [offset], 090
        jz label

        db length of routine 3

        mov dx, [offset]
        cmp dx, 08F
        jb label
```

```
Block13Control:
        dw Offset Block14       ;Pointer to next block
        db length of block
        db number of control codes
        db offset within block to (mov bx,6)
        db 0x00110110           ;Flags indicating to mutate this
instruction
        db offset within block to other instructions
        db 0x01100011           ;Flags indictating to and how to mutate
this
instruction
Block13:
        mov bx, 6
        inc ax
        jz Label
        mov cx, 7
End Block13:
```

```
Bit

8       - 1     - Further control bytes follow
7       - 1     - Junk afterwards can modify DX
6       - 1     - Junk afterwards can modify BX
5       - 1     - Junk afterwards can modify AX
3&4     - 00    - Add any kind of junk afterwards
        - 01    - Don't add flag modifying junk afterwards
        - 10    - Don't add any junk afterwards
        - 11    - Add garbage numbers afterwards
1&2     - 00    - Don't modify block at all
          01    - Permutate this instruction
          10    - Block is a string (Randomly upper/lower case it)
          11    - Custom
```

```
Permutated Code:
Offset1:
        mov ax,090
        nop     ;Junk
        nop     ;Junk
Offset2:
        inc bx
        nop     ;Junk
Offset3:
        add cx,1

ShuffledOffsetList:
        Offset3
        Offset1
        Offset2
```

```
Permutated Code:
Offset1:
        mov ax,090
        nop     ;Junk
        nop     ;Junk
Offset2:
        inc bx
        nop     ;Junk
Offset3:
        add cx,1

ShuffledOffsetList:
        Offset3
        db ControlByte
        Offset1
        db ControlByte
        Offset2
        db ControlByte
```

```
Block145Control:
        dw Offset Block146      ;Pointer to next block
        db length of block
        db number of control codes
        db offset within block to string
        db 0x01100111           ;Flag indicating to randomly convert
string to
upper/lower
Block145:
        db 'chklist.ms'
End Block145:
```

```
Bit

7&8     -00     Add any kind of junk afterwards
        -01     Add no junk afterwards
        -10     Don't add flag modifying junk afterwards
        -11     Add garbage afterwards
5&6     -00     Don't modify block
        -01     Permutate Instruction
        -10     Block contains string
        -11     Block contains garbage
4       -1      Custom
3&2&1   -Length of instruction/data
```

```
;Format of Reloc list items
;       DB      //Offset into code/data of block to modify offset
;       DB      //Flags to indicate type of modification
;       DW      //Logical source item number
;       DW      //Logical destination item number
;       DW      //Source offset - used by engine
;       DW      //Dest offset - used by engine
```

```
|discarded
|   -----
|-> | V |
|   -----
|   |   |  virus+engine generated from lists by engine
|-> | E | -|
    -----  |     -----
    |   |  |     | V |
    | L | -----> -----
    |   |        |   |
    |   |        | E |
    -----        -----
      |          |   |
list duplicated  | L |
   & shuffled    |   |
      |--------> |   |
                 -----
```

The engine would then choose a random routine and use this in the generated code. Obviously, the list then becomes quite large, but with compression/shuffling/encryption, it will vary considerably making it not an option to scan for.