ARE THERE ANY POLYMORPHIC MACRO VIRUSES AT ALL? (…AND WHAT TO DO WITH THEM)

Gabor Szappanos
VirusBuster Ltd., H-1116 Budapest, Vegyesz u. 17-25, Hungary
Tel +36 1 382 7000 • Fax +36 1 382 7007 • Email gszappanos@virusbuster.hu

ABSTRACT

This paper will investigate how the currently known polymorphic macro viruses fit into the usual terms used for binary polymorphic viruses and what special detection procedures have been implemented and should be developed to fight them.

The first question to answer is whether there are any of them that can be qualified as polymorphic. Most of them are very simple, and would not be polymorphic if VBA were a compiled language. There are several more complicated, encrypted viruses, some of them even with polymorphic encryptors.

Several code normalization techniques – most of which have been used in virus scanners for several years – exist, that transform the code into more-less identical form. The main objective is to use these normalization techniques so that the normalized code of all the replicas of the same virus will be exactly the same – this way even static CRC over the normalized code can be used for identification.

However, these methods lose information that could be important, especially in the case of encrypted macro viruses.

There are even more complicated viruses where these simple procedures are not sufficient. In these cases the development of advanced techniques is required, which include intelligent code parsing and analysis, leading the AV products very close to actual macro code emulation.

USING MIRA

In order to examine the polymorphic nature of viruses I used MIRA, with the permission of its author, Costin Raiu. A very detailed explanation of MIRA was presented by Costin Raiu at the AVAR 2001 conference [1], therefore there is no need for me to enter into its details.

This useful tool is aimed as a classification tool for new macro viruses by determining how similar the new sample is to the already classified known virus families. As a side-effect, it can measure how successful the so-called polymorphic macro viruses are in morphing their code. We will use only this side-effect throughout this presentation.

MIRA uses an opcode apparition frequency matrix constructed from four-byte opcode fragments. The difference between two matrices is calculated using a simple neural network and weight function, that eliminates the effect of zero values in the matrix.

Throughout the presentation I will use the output generated by MIRA to examine the similarity between replicas of the examined viruses. This way we will receive a partial answer to the question appearing in the title of this presentation.

I should note that MIRA is based on the opcodes, its results show how different the opcodes are. However, the macro virus detection is (at least) two-fold, some virus scanners base detection on source code, other on opcodes. We will see that these different approaches give different opinions on some particular virus' polymorphic nature.

EARLY RIDICULO-MORPHS

The first baby-step to polymorphism was ***Outlaw***. Frankly, it should not be called polymorphic at all. The virus code itself did not change a bit, only the name of the macro carrying it.

The only reason this virus was described as polymorphic (not that deserved it) was that, in the early months right after the appearance of Concept, there were several (mostly WordBasic-based) macro virus protection products that based their detection on macro names. As these products were surpassed by OLE2 parsing scanning routines, this group of viruses sank back to non-morphism.

JUNK INSTRUCTION INSERTERS

Another old virus, ***FutureNot*** consists of two modules, AutoOpen and FileSave. The latter, responsible for infecting further documents is gathered during the infection of the normal template and it is not present in the infected document.

The original AutoOpen macro is saved into the global template with a randomly selected five-character name. Additionally, the virus inserts to a random location in the code the text *1 Gen*. This serves the role of the generation counter – the number of such comments equals the number of infected computers along the current infection chain.

The virus line by line gathers the FileSaveAs macro. During this process the virus slightly mutates its code. It inserts two random numbers as comments, and a couple of extra line feeds. The resulting macro looks like this fragment:

```
dlg.Format = 1
‘ 0.38007424142506
jqp$ = FileName$()

‘ 0.12508087603783
```

Other good examples are the members of the WM97/Class family. These fill every other line of the macro code with random comment, usually combining the user name, the date, the installed printer, etc.

As a solution to overcome these simple viruses, scanners introduced several code normalization measures. The comments are removed from the code, as well as the white-space characters and the empty lines. After this code transformation the replicas of these types of polymorph viruses will have exactly the same look. Thus the identification is the same as with any other macro virus, even static CRC can be used for that.

Not in the case of junk-code inserters.

Good examples for this group are the members of the ***Coke*** family.

```
vHyjSvWnd.DeLEteLIneS 1, vHyjSvWnd.CoUntOFLinEs
fNRkKKWMuuLdd$ = “DJVDMFff”
WGfPYTIRnLGRtSS$ = “nWcgjbb”
Rem shxLccWVlHYY
Qtt = 66
vHyjSvWnd.inSErtliNes 1, ppgt.liNES(1, ppgt.CoUntOFLinEs)
For BGDD = 3 To 1
If juu = TVvhDCC Then
Rem SWdYRXEcDsMLJJ
Rem YuKGXHxxMprhwo
Rem hFlxnnDDqMM
End If
Next BGDD
xnrycNKOpyxtuss$ = “XLCkk”
tMNLLmbbtt$ = “GYtJyjA”
End If
wLxggSQQugGMVMpvv$ = “HshVHH”
nkkbnMVV$ = “JtlcNN”
If ppgt.liNES(1, 1) <> “‘gqAfatQEb” Then
If Spuu > KWvdd Then
‘mDlKjbXXTRWTRRR
If kpWI = FTSyy Then
SVmQRNN = 46
If Wsuu = TvpyuuJJ Then
If kPVTspp = tvMcKRFxx Then
DVrujFPP = 18
If khDrXJJ < yYPo Then
If JTTNnff > xPLTT Then
End If
Rem hWJCJeJutqq
jnDDxWusQFYY$ = “QQde”
End If
End If
```

A fragment of the virus code is illustrated above. Apart from the variable-name changing, junk instructions and instruction blocks are inserted. The meaningful lines are highlighted; the remaining lines are random variable assignments, combined with (possibly nested) conditional statements.

An even better example is ***Polymac.A (***also known as Chydow.A***),*** which also inserts nested junk code program structures, as illustrated below, with the meaningful instructions highlighted:

```
Loop While DjFeUOL2kkwJCIE6 < 44:
End If
Next
ActiveDocument.Save
DwFNV1qyGsV4 = 187
For LU1VyC6 = 8 To 54 Step 3:
If dDGlb7 <> Rnd * 28 Then
PjN7FDkGwkU7 = 9
Do
PjN7FDkGwkU7 = PjN7FDkGwkU7 + 6
Loop While PjN7FDkGwkU7 < 21

End If: Next
ActiveDocument.Close: TzLPGXn3lZ6 = 1
```

MIRA results for the three replicated samples are listed below. Even the highest score is below 0.8, which means that the virus is quite successful in morphing itself.

```
Top 10 matches for [c:\test\polymac\3\Normal.dot]:
0.99999 with [c:\test\polymac\3\Normal.dot] [394]
0.78263 with [intended://Word97Macro/Chydow.A] [369]
0.78189 with [c:\test\polymac\2\Normal.dot] [389]
0.78016 with [intended://Word97Macro/Chydow.A] [385]
0.77805 with [intended://Word97Macro/Chydow.A] [382]
0.77690 with [intended://Word97Macro/Chydow.A] [388]
0.77544 with [intended://Word97Macro/Chydow.A] [400]
0.77456 with [intended://Word97Macro/Chydow.A] [372]
0.77453 with [c:\test\polymac\1\Normal.dot] [343]
0.77439 with [intended://Word97Macro/Chydow.A] [391]
0.77138 with [intended://Word97Macro/Chydow.A] [381]
0.76509 with [intended://Word97Macro/Chydow.A] [383]
0.76480 with [intended://Word97Macro/Chydow.A] [391]
0.76166 with [intended://Word97Macro/Chydow.A] [388]
0.75810 with [intended://Word97Macro/Chydow.A] [384]
0.75360 with [intended://Word97Macro/Chydow.A] [363]
0.62516 with [virus://Word97Macro/Minimal.E] [36]
0.61481 with [virus://Word97Macro/Minimal.AU] [27]
0.60323 with [virus://Word97Macro/Minimal.F] [34]
0.58824 with [trojan://Word97Macro/Quitter.A] [53]
0.58599 with [virus://Word97Macro/Minimal.BK] [35]
0.58416 with [virus://Word97Macro/Minimal.D] [25]
0.57595 with [intended://Word97Macro/Sant.A] [15]
0.56755 with [virus://Word97Macro/Minimal.U] [29]
0.56603 with [virus://Word97Macro/Vp.D] [68]
```

None of the scores is higher than 0.8, and all come from different Polymac samples. It clearly indicates that this virus is quite successful in changing itself, the replicas are quite different from each other. Moreover, given the low ‘signal-to-noise ratio’ (i.e. the junk instructions outweigh the virus instructions), it is more likely that the similarity of 0.8 comes because of the junk instruction structures – the encryption engine uses a limited set of possible constructs, which will result in a more-less similar code.

Traditional code normalization techniques are not applicable there, still sometimes good scan strings are possible to find, or advanced techniques can be used.

Let’s take the above Polymac example. First, the code has to be precompiled, in which stage the VBA/application object variables are separated from the program variables. Both have to be replaced with variable tokens – this simple step will eliminate all the variable name changer polymorphs. After that the code flow has to be analysed. In this process the basic code structures (conditionals, loops,…) have to be identified. If we find that, within a structure, all variables are internal – meaning that no document/VBA objects are accessed and the used variables are not used in other parts of the code – then that code fragment has to be declared as an intrinsic junk code and can be eliminated. This will eliminate the macro viruses that insert random code line/structures into the virus code.

If we remove from the quoted code fragment the intrinsic junk code, only the following remains:

```
ActiveDocument.Save
ActiveDocument.Close
```

Which is invariant enough to CRC it. The speed of the indifferent code removal procedure is largely dependent on how far should it look. It may be very slow if long structures are processed, but junk code may remain if only short structures are handled.

VARIABLE NAME CHANGERS

Another interesting early attempts were the variable name changers. The best pure examples of this type are the members of the ***IIS*** family.

As an example, ***IIS.I*** uses the variable name changing polymorph method with some interesting twists. The author’s intent was to change each variable name that is used in the code but as usual, some of those were forgotten.

The virus creates variable names that use characters of high ASCII codes (from 130 to 204). The resulting code is rather difficult to read as the following code snippet illustrates:

```
If Left(+ÿ¦, 1) = “‘“ Then
«ÿæ¦-º+á¬â+¦¿+ = «ÿæ¦-º+á¬â+¦¿+ + 1
Pô½-Å-+-ù-Åè+¦º¢+òòñ(«ÿæ¦-º+á¬â+¦¿+, 1) = Mid(+ÿ¦, 2, Len(+ÿ¦))
```

In the preparation stage the virus collects the virus code from the Flitnic module in the global template into a temporary string buffer. Then it collects the variable names to be modified into a variable name array. It contains both the old names of the variables and the new names acquired during the mutation process. The variable names are collected from the ‘variable pool’ located in the middle of the virus code. It is just a series of commented lines that contain only the changeable variable names preceded by a single ‘ (REM) instruction. The virus processes the code line-by-line and if a line starting with a ‘ is found, the rest of the line after the ‘ will be added to the variable name array. As a consequence, no full comment lines can be present in the virus code.

After collecting all the variable names, the virus mutates them. The new names will have variable length (randomly selected from 2 to 22) and will be built of randomly selected characters with ASCII codes between 130 and 204. The new names are stored in the variable name array too.

Unlike most polymorphic macro viruses, Flitnic makes some checks for variable name conflicts. It searches the variable name array with to find two basic types of errors (both could lead to errors):

1. Some of the newly generated names coincide – this way two different variables would get the same name.
2. Some of the newly generated names coincide with an older name – this could also lead to problems when the variable names are changed to their new values.

If any of these mismatches is found, the virus solves the problem with generating a new name for the variable with higher index in the variable name array. This new name uses the same character range but it is only one character long. As all variable names created in the first round of the polymorph generation are at least two characters long, this could not lead to any new problems.

After the consistency of the new variable names has been checked, the virus searches through the buffer containing the virus code in the memory. It processes the code line-by-line. Each line is broken into tokens. The delimiters of the tokens are characters with ASCII code below 65 or the end of the line. If any token matches one of the old variable names, the virus replaces it with the new name of the same variable and continues the processing.

The polymorphic nature of this virus depends a great deal on whether the virus scanner uses source code or p-code representation for detection. If the latter is used, then the replicas are not much different; consequently the virus is not much of a polymorph. Nevertheless, it has to be dealt with for the sake of the ones who use source code representation. Which is clearly obvious from the MIRA output: all replicas are similar to the extent of at least 0.999 – which means that at the opcode level this virus is not polymorphic.

```
Top 10 matches for [c:\test\goat5.doc]:
0.99976 with [virus://Word97Macro/IIS.I] [500]
0.85305 with [virus://Word97Macro/IIS.H] [273]
0.83007 with [trojan://Word97Macro/Skeleton.A] [12]
0.79098 with [virus://Word97Macro/Class.FD] [10]
0.78864 with [garbage://Word97Macro/TestMacro] [29]
0.78277 with [intended://Word97Macro/Lys.H] [66]
0.76501 with [trojan://Word97Macro/Minus.A] [13]
0.75603 with [virus://Word97Macro/Loud.A] [47]
0.75518 with [virus://Word97Macro/Lys.G] [42]
0.75013 with [virus://Word97Macro/Lys.A] [46]
0.74608 with [virus://Word97Macro/Replog.A] [48]
0.73704 with [virus://Word97Macro/Ping.A] [54]
0.73108 with [virus://Word97Macro/Smac.B] [80]
0.72977 with [virus://Word97Macro/Example.B] [65]
0.72959 with [virus://Word97Macro/Replog.D] [45]
0.72758 with [intended://Word97Macro/Sant.A] [15]
0.71784 with [virus://Word97Macro/IIS.M] [40]
0.71507 with [intended://Word97Macro/Hana.A] [32]
0.71399 with [virus://Word97Macro/Coke.22231.A] [44]
```

```
0.71112 with [virus://Word97Macro/PassBox.S] [173]
0.71062 with [trojan://Word97Macro/Quitter.A] [53]
0.71055 with [virus://Word97Macro/Teocatl.A] [47]
0.71008 with [garbage://Excel97Macro/TestMacro] [28]
0.70962 with [virus://Word97Macro/Class.BV] [64]
0.70954 with [virus://Word97Macro/Aliv.A] [74]
```

However, at the source code level it is polymorphic enough to cause headache for the virus scanners that base their detection on the macro source code.

The appropriate solution for the detection of these type of viruses would be a pre-compilation of the code, which could determine which tokens are variables, which are VBA commands and which are object references. The variables would be replaced by variable tokens. The above code would look like:

```
If Left(var_1, 1) = "'" Then
var_2 = var_2 + 1
var_3(var_2, 1) = Mid(var_1, 2, Len(var_1))
```

Is that enough? Are we ready now? Not exactly. It is not guaranteed that the virus will add itself to an empty code module. In fact, there are parasitic viruses that attach themselves to the end of the existing module, and add only a call to the virus code in the macro – much like Win32 EPO viruses. In that case the variable token var_1 may well be var_234. Therefore the scanner should use adaptive scan strings: whenever it reaches a variable token in the scanned macro code, it should replace its value into the scan string used in the database. Alternatively, the scanner may lose all information about variables and use a single variable token. The latter solution would result in a huge loss of information, therefore it is not acceptable for virus identification.

THE FIRST POLYMORPHS

The first serious polymorphic macro virus was ***WM95/Slow*** [2]. It consists of a single *AutoClose* macro. The main virus code is stored in a string array, where the characters are shifted by a constant value selected randomly between 4 and 14.

```
FHUMCEAANPGT$(167) = "L{tizout&IKJQQQGXUGQLYXGN*"
FHUMCEAANPGT$(168) = "IUOIYOTPO[OUO*&C&FFFF"
FHUMCEAANPGT$(169) =
"Lux&HPNQHPSQW\[RLVT&C&QX\ZRWLOUW&Zu&WIYWVNKKVLZO[TMI&1&Xtj./
&0&WIYWVNKKVLZO[TMI&@&IUOIYOTPO[OUO*&C&IUOIYOTPO[OUO*&1&Inx*.Xtj./
&0&KILH[RNUYTV\NLWN[&1&XIKMNL\RH\T[NVP/&@&Tk~z&HPNQHPSQW\[RLVT"
FHUMCEAANPGT$(170) = "IKJQQQGXUGQLYXGN*&C&IUOIYOTPO[OUO*"
FHUMCEAANPGT$(171) = "Ktj&L{tizout"
```

After decoding it looks something like this:

```
LBJTAOCKNKC$(167) = "Function AADEEPGIVKAHI$"
LBJTAOCKNKC$(168) = "ONRFPIBQBFKETUJOFA$ = @@@@"
LBJTAOCKNKC$(169) = "For NAOMHIRBEOPKPVFFHUK = ACCDPJLOQGBMSFBSHTC
To BVSNPORQFGIUCSBPK + Rnd() * BVSNPORQFGIUCSBPK :
ONRFPIBQBFKETUJOFA$ = ONRFPIBQBFKETUJOFA$ + Chr$(Rnd() *
GFIMJETPTUOTSCSUV + SBQEERVPCTO) : Next NAOMHIRBEOPKPVFFHUK"
LBJTAOCKNKC$(170) = "AADEEPGIVKAHI$ = ONRFPIBQBFKETUJOFA$"
LBJTAOCKNKC$(171) = "End Function"
```

The variable names inside the encoded code are also randomly selected, 10- to 20-character long names. A more descriptive representation of the same code would read:

```
source$(167) = “Function Get_random_name$”
source$(168) = “templine$ = @@@@”
source$(169) = “For sourceline = number_1 To name_length + Rnd() *
name_length : templine$ = templine$ + Chr$(Rnd() * last_letter +
char_A) : Next sourceline”
source$(170) = “Get_random_name$ = templine$”
source$(171) = “End Function”
```

Only a short decryptor is in visible format, which creates a temporary macro and decodes the main code there. It then executes this code, which, in turn, mutates the virus body. First it picks a new encryption key, then it changes the variable names, finally it writes it all into the AutoClose macro of the target document.

The ultimate specimen of *Word 6* polymorphs was ***Nasty***, utilizing the complete arsenal of tricks [3]. The main virus code was stored as an Autotext entry, while the AutoOpen macro contained only a short stub.

The virus body is extracted into the ToolsMacro and FileSaveAs macros, with a junk line inserted after each virus line. The junk line is either a random comment, or a random value assigned to a random, never more used variable, or the date assigned to a random variable. Finally the virus shuffles the parameter list of the *Word* commands. For example, the command that edits the new macro may look like any of the following lines:

1. `ToolsMacro .Edit, .Name=”XXXX”, .Show=0`
2. `ToolsMacro .Edit, .Show=0, .Name=”XXXX”`
3. `ToolsMacro .Name=”XXXX”, .Edit, .Show=0`
4. `ToolsMacro .Name=”XXXX”, .Show=0, .Edit`
5. `ToolsMacro .Show=0, .Name=”XXXX”, .Edit`
6. `ToolsMacro .Show=0, .Edit, .Name=”XXXX”`

THE WAY TO REAL POLYMORPHISM

The recent development continued with ***Zevota*** [4].

Zevota uses random variable names and most of the constants (even the numeric ones) are encrypted with a key, which is not fixed either, changes with each mutation and code line. The procedures of the virus are also shuffled upon each mutation.

When an infected document is opened, the Document_Open procedure is activated, which is the only fix function name the virus uses. First it creates a mutated image of itself, saves it in the default document store directory (often the same as the location of the infected file). The name of this document will be the user name, with DOC extension. Then the virus infects NORMAL.DOT, without changing its shape.

After that Zevota mails itself out to at most 50 recipients picked from the first *Outlook* address list. The outgoing mail messages have the subject line:

```
{Username} "- Curriculum Vitae"
```

Where {Username} is the user name as registered in *MS Office*. Furthermore, the virus sends a reply to the last nine messages from the incoming mailbox, with an empty mail and the mutated virus copy attached.

Then the virus infects all documents on opening, without mutating the code.

Zevota utilizes the old variable-name mutating trick, combined with a little twist. Not only the internally used variable and procedure names (except for the Document_Open procedure) are changing, the numeric and string constants used by the virus are also encoded, moreover this encryption key is not the same for the entire code module, rather varies with each line, and changes with each mutation.

The virus stores the name of the variables to mutate as a comment line. It is recognized by starting with the ' comment sign and ending with *. The variables names are separated by – characters. The line is inserted in a random location in the code module of the infected document.

Once the old variable name pool is collected, the new names are generated with seven- to eight-character length using only capital letters from the English alphabet.

As the new variable names are generated, the entire code module is processed and the new variable names replace the old ones. One of the variables is the name of the variable encryption function.

Most of the numeric and character constants are stored in encrypted form, e.g. *OISOJQVM("Y{r•j}n", -9)* stands for the string *"Private"* and *Val(OISOJQVM("HL", -22))* for the numeric value *65*. The first mutation step processes these encrypted constants. All the above function references to the variable decryptor are parsed from the code module text, the encrypted string and the encryption key are extracted, and then a new key is generated. After that the variable is decrypted with the old key, then encrypted with the new key and the new expression is collected and inserted in place of the old one in the code module.

The virus code is processed line-by-line, with the encryption key being generated only once for a line. Different lines use different encryption keys.

While generating the new encryption key the virus checks if the encrypted text would contain the characters " or (. These separator characters would cause serious problems when inserted into the middle of a variable name; therefore the virus has to avoid this undesirable situation.

To make the situation more delicate, the virus shuffles its procedures upon each mutation. It collects the procedures into a string array (the procedures are recognized by starting with 'Private' keyword. The public declarations on top of code module (that, according to the VBA syntax, cannot be placed to a different location) are inserted first, and then generating a random number picks one of the procedures. This procedure is inserted first into the new code module, and then all of the remaining procedures are inserted in order. Therefore after the mutation, only one procedure changes its place, being moved to the beginning of the code module.

```
Top 10 matches for [c:\test\zevota\GEN3\VIRUSB~2.DOC]:
0.99996 with [c:\test\zevota\GOAT1.DOC] [466]
0.99333 with [c:\test\zevota\GEN2\VIRUSB~2.DOC] [465]
```

```
0.98915 with [virus://Word97Macro/Cham.A] [464]
0.98470 with [c:\test\zevota\VIRUSB~2.DOC] [468]
0.69993 with [virus://Word97Macro/Minimal.BD] [27]
0.68194 with [intended://Excel97Macro/Sud.A] [38]
0.67591 with [virus://Word97Macro/Minimal.AY] [28]
0.67307 with [virus://Word97Macro/AntiSocial.K] [68]
0.67046 with [virus://Word97Macro/Muck.BF2] [31]
0.66391 with [virus://Word97Macro/Minimal.AU] [27]
0.65844 with [virus://Word97Macro/AntiSocial.I] [71]
0.65499 with [virus://Word97Macro/Minimal.B] [44]
0.65068 with [virus://Word97Macro/Minimal.J] [26]
0.64487 with [virus://Word97Macro/Minimal.F] [34]
0.64338 with [virus://Word97Macro/Replog.A] [48]
0.64164 with [virus://Word97Macro/AntiSocial.J] [76]
0.64006 with [virus://Word97Macro/Azrael.A] [42]
0.63825 with [virus://Word97Macro/Thus.EM] [81]
0.63812 with [virus://Word97Macro/Example.C] [25]
0.63718 with [virus://Word97Macro/Minimal.D] [25]
0.63636 with [trojan://Word97Macro/Skeleton.B] [11]
0.63458 with [virus://Excel97Macro/Toraja.B] [11]
0.63339 with [virus://Word97Macro/AntiSocial.L] [66]
0.63298 with [virus://Word97Macro/Minimal.Z] [22]
0.63097 with [virus://Word97Macro/Minimal.AM] [49]
```

It is obvious from the MIRA output that the polymorph engine in the virus is only complicated enough to obscure the code and make the analysis more difficult, the compiled code does not change much. Despite the high level of similarity, it is not possible to transform the code into an invariant form, because of the encryption of the numeric and string variables with variable encryption keys. The procedure shuffling does not change the instruction frequency matrix much. The most appropriate solution for the detection of this virus is the selection of proper scan strings.

At this moment the most complicated polymorph macro virus is ***Jug.A*** [5]. Most of the virus code is encrypted with a variable length encryption table that is changed with every infection. The encrypted body is itself variable, random junk strings are appended to the end of the code line and inserted into random location in the code. The polymorphic engine randomly gathers even the decryptor.

The most complex part of the virus is definitely the polymorphic engine. It consists of two consecutive procedures; the first inserts random comments into the main virus body, and then encodes it, while the second part generates the decryptor.

The engine first attempts to find the encryption table stored in the decryptor, recognized as any string coming after Asc(Mid(“ in the target code module. If such a string is found, the virus attempts to decode the first line of the target code module with this key. If the decoded string contains the string ‘Nephalim v0’, then the document is considered to be already infected with some version of Jug, therefore left alone.

If it is not infected yet, the virus creates a new encryption key (consisting of 6 to 35 characters from the ASCII range 65…122), then after each virus code line a random postfix is added with a 1 in 3 chance in order to change the line length. The postfix consists of :NPR and 6 to 35 random characters. Note that the postfix is not separated from the virus code with a comment, as such, the codeline in this form would not be executable. However the decryptor recognizes these postfixes by the :NPR string, and removes them from the decrypted virus code.

After each code line the virus will insert a random junk line with a one in three chance. The random line will start with 'NP and 6 to 115 random characters.

These random comments serve the purpose that the encrypted body would not have a fixed number of lines or the encrypted lines would not have fixed length; these parameters vary with each infection.

Once the virus body is ready with all comments inserted, the virus encodes it with the newly generated encryption table. The coding is simple: each character is shifted with the ASCII value of the encryption table character (for each character at a position over the encryption table length, the first character of the table will be used), with special care taken that the coded character's ASCII value would be between 32 and 130.

The last step is the creation of the random decoder. The decryptor utilizes variable-name changing also. Each variable is one byte long and randomly selected from the 'A' .. 'Z' range. Special care is taken so that variable names would be different.

Two crucial object variables, *Z.VBProject.VBComponents.Item(Y).CodeModule* and *VBProject.VBComponents.Item(1).CodeModule* (where Z and Y are one of the random variable names generated) are both cut into two pieces among one of the "." dereferences, and stored in two variables each. The first part is stored in a variable, and all further references appear as relative to that variable object (e.g. R will be *Z.VBProject.VBComponents*, further references will appear as R. *Item(Y).CodeModule*).

Thus, it is not easily possible to lay a scan string on these references for detection.

Furthermore, the numerical constants in the code (virus code line count, character range borders) are all generated as a sum of two numbers. The virus line count is 130, therefore a random number is generated between 0 and 129 (say 35), and the line count is referenced later in the generated decoder as 35+95.

Then the code lines are generated one by one. For those lines, that can be merged together, a line connector is selected randomly, which can be either ':' (which means that the line is joined with the next one), or a line feed, or two line feeds, or a random comment and a line feed.

For those lines, that cannot be joined with the next line (e.g. the Else in an If statement), a random line terminator is picked. This can be one to three consecutive line feeds.

After the decryptor, an almost empty Document_Open and Workbook_Open procedures are generated. These procedures contain only a call to the virus decryptor. The declaration of these procedures is also randomly generated; it can be '*Private Sub*', '*Public Sub*' or plain '*Sub*'.

```
Top 10 matches for [c:\test\jug-a\GOAT5.DOC]:
0.99999 with [c:\test\jug-a\GOAT5.DOC] [207]
0.82223 with [virus://Word97Macro/Jugular.A] [175]
0.81800 with [c:\test\jug-a\GOAT1.DOC] [171]
0.79944 with [c:\test\jug-a\GOAT4.DOC] [175]
0.79574 with [c:\test\jug-a\GOAT2.DOC] [172]
0.78111 with [c:\test\jug-a\GOAT3.DOC] [165]
0.75522 with [virus://Word97Macro/Jugular.A] [172]
0.51765 with [virus://Word97Macro/Minimal.AU] [27]
```

```
0.49964 with [virus://Word97Macro/Minimal.Z] [22]
0.47906 with [virus://Word97Macro/Minimal.D] [25]
0.47638 with [virus://Word97Macro/Oldguy.C] [63]
0.47460 with [virus://Word97Macro/Azrael.A] [42]
0.46703 with [virus://Word97Macro/Azrael.B] [32]
0.45582 with [virus://Word97Macro/Juntin.A] [81]
0.45239 with [trojan://Word97Macro/Quitter.A] [53]
0.45137 with [virus://Word97Macro/AntiSocial.I] [71]
0.44746 with [virus://Word97Macro/Horn.A] [78]
0.44400 with [virus://Word97Macro/AntiSocial.K] [68]
0.44344 with [virus://Word97Macro/AntiSocial.G] [102]
0.42278 with [virus://Word97Macro/Minimal.J] [26]
0.42260 with [virus://Word97Macro/AntiSocial.J] [76]
0.41976 with [virus://Word97Macro/Hope.B] [26]
0.41904 with [virus://Word97Macro/AntiSocial.G] [96]
0.41820 with [virus://Word97Macro/AntiSocial.N] [67]
0.41773 with [virus://Word97Macro/AntiSocial.P] [67]
```

Most of the virus code is stored as a comment, therefore the similarity level refers to the decryptor procedure. It is obvious from the values of about 0.8 that the polymorphic engine of the virus is quite successful in altering the decoder. There is no hope to transform the code into an invariant form, because the numeric and the crucial object variables are stored in a variable form.

It is beyond the scope of code normalization to transform this virus into invariant form. A simple precompilation algorithm, that would replace numeric expressions with their result (thus instead of 35+95 it would be 130), and relative object references with full object references (thus instead of R. *Item(Y).CodeModule* it would be *Z.VBProject.VBComponents.Item(Y).CodeModule*), would be a better approach, yet not perfect.

METAMORPHIC MACRO VIRUSES

Some viruses take a different approach to polymorphism. Instead of making the changes at the code line level, these viruses modify the order of the procedures.

One example, ***NPR.A*** relocates only one procedure per infection, moving it to become the first procedure. In the first sample the sequence of procedure would look like this (this is just an illustration, the virus itself contains several other procedures):

```
Adapt()
SCP()
Sender_main()
AutoClose()
```

While in another it would look like this:

```
Sender_main()
Adapt()
SCP()
AutoClose()
```

As for the viruses, it is very easy to accomplish procedure swapping, a lot simpler than in the case of binary viruses. VBA provides the necessary functions (ProcBodyLine, ProcCountLines, ProcOfLine) required to extract the code of a single procedure into a string. Some viruses use the hard way to achieve the same, recognizing the border of procedure by the inserted marker comments (like ***NPR.A***) or using a list of function declarations (like ***Saray.A***). The code is not changing much, as it is clear from the output below. The change at source code level is only a procedure reorder, while on opcode level minimal modifications are possible, because the variable references are changed with the reorder of the variables.

```
Top 10 matches for [c:\test\saray\NORMAL.DOT]:
0.99553 with [c:\test\saray\GOAT1.DOC] [452]
0.88941 with [virus://Word97Macro/Class.FD] [10]
0.81533 with [virus://Excel97Macro/Toraja.B] [11]
0.79129 with [virus://Word97Macro/Kop.E] [59]
0.77201 with [virus://Word97Macro/NotHere.A:Ru] [58]
0.76290 with [virus://Word97Macro/Hope.B] [26]
0.74768 with [virus://Word97Macro/VMPCK1.CG] [105]
0.74563 with [virus://Word97Macro/Coke.22231.A] [44]
0.73681 with [virus://Word97Macro/Kop.F] [62]
0.73489 with [virus://Word97Macro/Keta.A] [111]
0.72805 with [virus://Word97Macro/Keta.A] [112]
0.72699 with [virus://Word97Macro/Nid.A] [97]
0.72489 with [intended://Word97Macro/Elbag.A] [85]
0.72484 with [virus://Word97Macro/MCK.B] [85]
0.72057 with [virus://Word97Macro/Tech.G] [102]
0.72034 with [virus://Excel97Macro/Teocatl.A] [51]
0.71819 with [virus://Word97Macro/Breeze.F] [36]
0.71749 with [virus://Word97Macro/Marker.HM] [25]
0.71360 with [intended://Word97Macro/Hana.A] [32]
0.71351 with [virus://Word97Macro/NightShade.B] [92]
0.71148 with [virus://Word97Macro/Titch.F] [91]
0.70918 with [virus://Word97Macro/Hope.F] [63]
0.69964 with [virus://Word97Macro/Lys.K] [27]
0.69942 with [virus://Excel97Macro/Tha.A:De] [105]
0.69720 with [virus://Word97Macro/Teocatl.A] [47]
```

The detection of viruses in this group is not that difficult. In fact, even CRC-based detection is possible without extra code transformation, if the range is carefully selected to cover only a singe procedure. This is only applicable if the scanner is flexible enough to define the CRC range relative to a scan string position found in the code, and not only relative to the macro code start. In the latter case, simple scan string detection is still possible.

FUTURE IMPROVEMENTS

The most complicated polymorphs, explained above, prove such a level of morphism that is quite difficult to detect with the traditional CRC techniques. A little step further and the simple signature-based techniques will fail. Virus scanners will have to employ at least pre-compiler methods, if not partial VBA code emulation.

REFERENCES

[1] C. Raiu: ‘M.I.R.A. - searching for the lost kids’, *AVAR 2001 conference.*

[2] Cai-Gong Quin, ‘SlovakDictator’, *Virus Bulletin*, October 1997.

[3] Beata Ladnai, ‘New Kid in Town’, *Virus Bulletin*, November 1997.

[4] G. Szappanos, ‘Zevota Confidence’, *Virus Bulletin*, June 2001.

[5] G. Szappanos, ‘Juggling the Code’, *Virus Bulletin*, April 2002.