

Polymorphic Macro Viruses, Part Two

Gabor Szappanos 2002-11-05

This article is the second of a two-part series that will offer a brief overview of polymorphic strategies in macro viruses. The [first installment](#) of this series looked at some early examples of polymorphism, along with some of the early polymorphic techniques. This installment will look at the first serious polymorphic macro viruses, as well as the evolution of viruses into true polymorphic and, ultimately, metamorphic viruses.

The First Polymorphs

WM95.Slow was the first serious polymorphic macro virus. 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$(167) = "L{tizout&IKJQQGXUGQLYXGN*"
FHUMCEAANPGT$(168) = "IUOIYOTPO[OUO*&C&FFFF"
FHUMCEAANPGT$(169)
=
  "Lux&HPNQHPSQW\[RLVT&C&QX\ZRWLOUW&Zu&WIYWVNKKVLZ0TMI&1&Xtj./&0&WIYWVNKKVLZO
[TMI&@&IU
  OIYOTPO[OUO*&C&IUOIYOTPO[OUO*&1&Inx*.Xtj./&0&KILH[RNUYTV\NLWN&1&XIKMNL\RH\T[NVP/
&@&T
  k~z&HPNQHPSQW\[RLVT"
FHUMCEAANPGT$(170) = "IKJQQGXUGQLYXGN*&C&IUOIYOTPO[OUO*"
FHUMCEAANPGT$(171) = "Ktj&L{tizout"
```

After decoding, it looks something like this:

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

The variable names inside the encrypted 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$(168) = "templines$ = @@@@"
source$(169) = "For sourceline = number_1 To name_length + Rnd() * name_length :
    templines$ = templines$ + Chr$(Rnd() * last_letter + char_A) : Next sourceline"
source$(170) = "Get_random_name$ = templines$"
source$(171) = "End Function"
```

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

Nasty was the ultimate example of a Word 6 polymorph that utilized the complete arsenal of tricks. The main virus code is stored as an Autotext entry, while the AutoOpen macro only contains 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 consists of a random comment, a random value assigned to a random obsolete 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

Zevota continued the development of true polymorphs. It uses random variable names, and most of the constants (even the numeric ones) are encrypted with a key that is not fixed but 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 fixed function name the virus uses. First it creates a mutated image of itself and saves it in the default document store directory, which is often the same as the location of the infected file. The name of this document will be

the user name, with the ".doc" extension. The virus then infects NORMAL.DOT, without changing its shape. After that Zevota mails itself out to a maximum of 50 recipients picked from the first Outlook address list.

The outgoing mail messages have the subject line: {Username} "- Curriculum Vitae" in which {Username} is the user name that is registered in MS Office. Furthermore, the virus also sends a reply to the last 9 messages from the incoming mailbox, with an empty mail and the mutated virus copy attached. The virus then infects all documents on opening without mutating the code.

Zevota utilizes the old variable name-mutating trick with a little twist. Not only do the internally used variable and procedure names (except for the Document_Open procedure) change, 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, it varies with each line and changes with each mutation.

The virus stores the name of the variables to mutate as a comment line. This comment line is signified by the character - and ends with the character *. The variable names are separated by a dash (-). 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 using 7-8 characters consisting of only upper case 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. For example, the string **OISOJQVM** ("**Y{rj}n**", -9) stands for the string "Private" and the string Val(OISOJQVM("HL", -22)) stands for the numeric value 65. The first mutation step processes these encrypted constants. All of the 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 contains 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 starting with the keyword "Private"). The public declarations on top of the code module (that, according to the VBA syntax, cannot be placed to a different location) are inserted first, and then one of the procedures is selected by generating a random number. 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.

The polymorphic 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 the time of publication of this article, **Jug.A** was the most complicated polymorphic macro virus. Most of the virus code is encrypted with a variable-length encryption table that changes 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 locations in the code. The polymorph engine also randomly gathers the decryptor.

The polymorphic engine is definitely the most complex part of the virus. 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, which is signaled by any string coming after the string **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 vO**, then the document is already infected with some version of Jug. It is therefore left alone.

If it is not infected yet, the virus creates a new encryption key consisting of 6-35 characters from the ASCII range 65-122. Then, after each virus code line, a random postfix is added with a 1 in 3 chance of changing the line length. The postfix consists of the string **:NPR** and 6-35 random characters. Note that the postfix is not separated from the virus code with a comment. As such, the code line 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 1 in 3 chance that the line length will be changed. The random line will start with the characters **?NP** and 6 - 115 random characters. These random comments ensure that the encrypted body will not have a fixed number of lines or the encrypted lines would not have fixed length. These parameters vary with each infection.

After 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 A-Z. Special care is taken so that variable names will 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 "." references, 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. (For example, R will be `Z.VBProject.VBComponents`, further references will appear as `R.Item(Y).CodeModule`). As a result of this, 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. The code lines are then generated one by one. For those lines that can be merged together, a line connector is selected randomly, which can be either the character : (which means that the line is joined with the next one), a line feed, two line feeds, or a random comment and a line feed.

For those lines that cannot be joined with the next line (such as the "Else" in an "If" statement), a random line terminator is picked. This can be 1-3 consecutive line feeds.

After the decryptor, 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".

Most of the virus code is stored as a comment. Therefore, the similarity level refers to the decryptor procedure. The polymorphic engine of the virus is quite successful in altering the decoder. It is not possible 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 an 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 (for instance, instead of R.Item(Y).CodeModule it would be Z.VBProject.VBComponents.Item(Y).CodeModule) would be a better approach, although still not perfect.

Metamorphic Macro Viruses

The term metamorphic refers to viruses that change the order of the procedures or code blocks during replication. The functionality of the code remains the same, as does the execution order. Only the physical location in the code will differ. While this technique is very difficult to implement in binary malware, as the virus has to find out the hard way the procedure borders, it is very easy to do in macro viruses. First of all, it spreads in source code. As a result, string markers can be easily placed on the procedure borders (if the "Sub - End Sub" is not enough by itself), and the string search routines are accessible in VBA. Moreover, code manipulation functions that enable the virus to extract and manipulate procedure code are readily available in VBA.

Procedure Swappers

Some of the metamorphic viruses approach polymorphism differently. 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 so that it is the first procedure. In the first generation, the sequence of the procedure would resemble the following (this is just an illustration, the virus itself contains several other procedures):

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

While in another generation it might look like:

```
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 does not change much, as is clear from the output below. The change at source code level is only a procedure reorder, while on the opcode level minimal modifications are possible because the variable references are changed with the reordering of the variables.

The detection of viruses in this group is not very difficult. In fact, even CRC-based detection is possible without extra code transformation if the range is carefully selected to cover only a single 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.

Code Block Swappers

Code swapping cannot stop at the procedure level. Procedure blocks can be isolated and swapped as well. This is what **Walker.B** does. This virus merges swappable instruction blocks into single code lines. Special care has to be taken so that code structures are in the same line. For example, it would generate unexecutable code if the "If" and the "End If" in a conditional statement would be swapped.

Each code line has a number and ends with a "Goto" to the next command line. Code line numbering is an old BASIC attribute that was implemented in VBA as well, though not publicized at all, and missing from the tutorials. It still works for the virus. Upon replication the virus randomly picks lines, deletes them and inserts the deleted line content into a random location. In one sample, the beginning of the virus code would look like the following:

```
GoTo 10
50 Set ActCarrier = ActiveDocument.VBProject.VBComponents(1).CodeModule: GoTo 60
20 Options.VirusProtection = False: GoTo 30
```

or in another sample:

```
GoTo 10
20 Options.VirusProtection = False: GoTo 30
30 Options.SaveNormalPrompt = False: GoTo 40
10 Application.EnableCancelKey = wdCancelDisabled: GoTo 20
```

Due to a little sloppiness, the illustrated virus is not viable as the "End Sub" procedure terminator was judged to be swappable and, as a result, in most of the replicas, most of the "Goto" instructions point after the procedure end, which will generate a compile error without executing the virus code fragment.

Only the variable indexes and the "Goto" displacements changed in the opcode. However, it is not possible to transform the code into invariant form. Furthermore, the CRC detection is not applicable since the code is broken into small fragments (much smaller than in the case of procedure-swappers). The most adequate method for the detection of **Walker.B** samples is to use a combination of scan strings. The scanner engine should support the use of multiple scan strings and conditional statements between them (for instance, 3 scan strings collected from three lines of the virus should be detected in the macro at the same time for a virus hit).

I mentioned that the individual code lines are exactly the same in the source code representation. This does not hold true for the opcode representation. For example, the code line

```
90 If UCase(NI) = "ON ERROR RESUME NEXT" Then NormInstalled = True Else NormInstalled = False: GoTo
100
```

in opcode representation is represented as the following in one of the replicas (normal.do2):

```
0000:0040  A8 00 6E 02 20 00 2A 02 24 00 2E 02 01 00 B6 00
0000:0050  14 00 4F 4E 20 45 52 52 4F 52 20 52 45 53 55 4D
0000:0060  45 20 4E 45 58 54 05 00 9B 00 47 00 B7 04 27 00
0000:0070  30 02 63 00 47 00 B7 00 27 00 30 02 4C 02 46 00
0000:0080  00 00 9A 00 4E 02
```

and as this one on another (normal.do3):

```
0000:02FA  A8 00 6E 02 20 00 2A 02 24 00 2E 02 01 00 B6 00
0000:030A  14 00 4F 4E 20 45 52 52 4F 52 20 52 45 53 55 4D
0000:031A  45 20 4E 45 58 54 05 00 9B 00 47 00 B7 04 27 00
0000:032A  30 02 63 00 47 00 B7 00 27 00 30 02 38 02 46 00
0000:033A  00 00 9A 00 7A 02
```

Due to the variable displacement and the Goto range change, only minor changes are apparent. The similarity is still high.

So Walker.B is a good example of a virus that is not really polymorphic from the source code point of view, but that is polymorphic enough from the opcode point of view. It is still possible to normalize the opcode by replacing the variable references by the actual variables, thereby creating a mixed opcode-source variable representation.

Looking Into the Future

The most complicated polymorphs discussed in this article prove 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 in order to overcome this challenge. So what can we expect in the future?

I have an optimistic and a pessimistic vision and it is not clear yet which one of them will be realized. In the optimistic version, the constant decrease of new macro viruses will continue, up to the point when there will be no more, and the virus writers will lose interest in them. In the pessimistic version, we will see that more and more complicated macro viruses will be created, and their complexity will reach the point where the traditional techniques will become useless.

Relevant Links

[Polymorphic Macro Viruses, Part One](#)
Gabor Szappanos, SecurityFocus

[Privacy Statement](#)

Copyright 2006, SecurityFocus