MALWARE ANALYSIS 1

NOT 'MIFEVE'-OURITE THING

Peter Ferrie
Microsoft, USA

MATLAB is probably not the first platform that comes to mind when talking about viruses (despite a proof of concept having appeared in 2006[1]). However, with its vast collection of mathematical functions it lends itself to all kinds of problem-solving mischief, as we can see in the MLS/Mifeve virus.

ARTISTIC DIFFERENCES

The virus is extremely complex, but amazingly stable despite its size. All of the major bugs that I thought I had found (there were a few) turned out to be misunderstandings on my part (there were many). So, it has no great faults in terms of its logic. It does, however, have many faults in terms of its 'style'. The code has some non-optimal sections, but this contributes only a little to the size. For example, some blocks have been copied to other areas of the code, and then modified, which results in dead code due to the different context. Then there are little things like the fact that a line containing a minor bug has been reproduced multiple times – resulting in the bug appearing in multiple places. In one case, a string replacement function is used to search for a string that has already been replaced entirely in order to reach that line. In another case, a constant result is evaluated repeatedly due to its misplacement inside a while loop. In yet another case, a loop runs to completion without effect if a condition inside it evaluates to false. There is also heavy use of the fix(rand()) function, despite the existence of a single randi() function which combines the effects of both. Perhaps the author of the virus became as tired of writing it as I did of reading it.

LIFE STAGES

There are two versions of the code. One is a 'demonstration' version that shows the transformation of a simple message. The other is a 'release' version, which is fully self-replicating. The two versions have essentially the same engine functionality.

The virus goes through several steps to produce a new version of itself: (1) it splits its own code into parts; (2) it defines the parts in a random order; (3) it reconstructs the parts in the proper order using 'if/else' statements. The components of the 'if/else' statements are complex mathematical statements in the form of inequalities.

[1] Bontchev, V. Math baloney: yet another first. Virus Bulletin, June 2006, p.4. http://www.virusbtn.com/pdf/magazine/2006/200606.pdf.

STAGE 1

The virus begins by generating replacement names for each variable that is used in the code. The demonstration version has two key variables which are not in the list of variables, but the message to transform contains none of the names, so nothing is replaced. A funny bug exists here in both versions, which is that the random number generator has not been seeded yet. As a result, at least in earlier versions of *MATLAB*, the sequence of random numbers will be identical whenever *MATLAB* is restarted, until the generator is seeded (which might happen in the host code of an infected file). The virus avoids producing names that match existing variable names or keywords. The replacement names are between five and 19 lower case characters long, in the range of 'a' to 'y'. The letter 'z' cannot be generated due to a bug in the virus code.

At this point, the virus seeds its random number generator using the current time (there is a slightly different algorithm between demonstration and release versions, but the difference is not relevant) and the Mersenne Twister algorithm. The virus creates an array of offsets at which to split its own code. There can be as few as three offsets in both versions. For the demonstration version, there can be as many offsets as there are bytes in the code. For the release version, the maximum number of offsets is equivalent to about one fifth of the size of the code. The virus splits the code into parts whose size is determined by pairs of offsets, and creates a random name corresponding to each of the parts. The part names are between four and 18 lower case characters long, in the range of 'a' to 'y'. As before, the letter 'z' cannot be generated due to a bug in the virus code.

STAGE 2

The virus drops an ODE function file, which will be called if an ordinary differential equation is used. The name of the file is treated like the other variables in the virus code, and is therefore not constant, however its contents are. The virus creates a threshold for encrypting the individual parts. In the release version, there is an approximately 33% chance of encryption in all cases, an approximately 33% chance of no encryption at all, and a 'threshold' that is chosen randomly in all other cases (though it's really an anti-threshold, since it behaves as the upper limit – not the lower limit – for action). In the demonstration version, there is a 50% chance of encryption in all cases.

For each part of the virus code, the virus displays the index of the part and the number of parts as a kind of progress indicator. If the current part is not the first one, then with an approximately 60% chance, and if the previous part

has already been marked as processed, the virus chooses whether or not to encrypt it. If a randomly chosen value is below the 'threshold', then the part is encrypted. Otherwise, the part is stored as plain text. After processing, the part is appended to the previous part and marked as processed. This is followed by the generation of garbage code. The result is the line 'varPrev=[varPrev code]', where 'varPrev' is a random variable name.

If the current part is not already marked as processed, and if it is not the last part, then with an approximately 60% chance, and if the next part has already been marked as processed, the virus chooses whether or not to encrypt it. If a randomly chosen value is below the 'threshold', then the part is encrypted. Otherwise, the part is stored as plain text. After processing, the part is prepended to the previous part and marked as processed. This is followed by the generation of garbage code. The result is the line 'varPrev=[code varPrev]', where 'varPrev' is a random variable name.

If the current part is not already marked as processed, then the virus chooses whether or not to encrypt it. If a randomly chosen value is below the 'threshold', then the part is encrypted. Otherwise, the part is stored as plain text. After processing, the part is marked as processed, and the variable name is added to the list of defined variables. This is followed by the generation of garbage code. The result is the line 'varCurr=[code]', where 'varCurr' is a random variable name.

For each part, beginning with the second one, if the current part is already marked as processed, and if the previous part is also marked as processed, then with an approximately 60% chance, the virus will choose how to combine the current part. With an approximately 50% chance, the virus will combine the previous part and the current part into the previous variable and discard the current variable. Otherwise, it will combine the previous part and the current part into the current variable, and discard the previous variable. This is followed by the generation of garbage code. The result is the line 'varPrev=[varPrev varCurr]' or 'varCurr=[varPrev varCurr]', where 'varPrev' and 'varCurr' are random variable names.

After all parts have been processed, if any remain that have not been assigned, the code will execute the final routine repeatedly until all of them are assigned, and use a second method of garbage code generation. The end result is that all of the parts are combined into a single variable which will be the whole virus body. Once that operation is complete, there will be one garbage line for each real line.

ENCRYPTION

In order to encrypt the parts, the virus chooses randomly from several algorithms that are derived from a formal grammar: 'F(NOS)Q', 'F(SON)Q', 'F(S)Q'. Initially, each of these algorithms appears twice, thus there are two chances to select any of them. However, after the initial choice is made, one 'NOS' and one 'SON' algorithm is removed. The virus parses the algorithm while any of the 'S', 'O' or 'F' elements remain. Only these three need to be checked, because the 'N' and 'Q' elements will also be replaced while any of the others remain.

'S' is a start symbol. It is replaced either by 'if' or by another algorithm chosen randomly from a set. 'F' is a function symbol. It is replaced by a function chosen randomly from a set. 'O' is an operation symbol. It is replaced by an operator chosen randomly from a set. 'N' is a number symbol. It is replaced by a random floating-point number. This number is multiplied by 10 to produce a value which has a potentially non-zero digit to the left of the decimal point. 'Q' is a power symbol. It is either removed, or there is a 20% chance that it will be replaced. Given those rules, the grammar looks like this:

```
S -> F(NOS)Q | F(SON)Q | F(S)Q | if

if -> set of floating-point numbers

F -> sin | cos | exp | atan | sinh | cosh | log |
asin | acos | tan

N -> floating-point number

O -> "+" | "-" | ".*"

Q -> "" | ".^" Z

Z -> 2 | 3 | 4 | 5 | 6
```

If the number of parenthesis pairs exceeds 25, then the line is considered to be complex 'enough', and all algorithms are disabled to force the transformation to complete sooner. The 'if' that appears as part of the 'S' replacement is a placeholder for a set of random floating-point numbers ranging from zero to the length of the original string. Each of the values is multiplied by 100 to produce a value which has up to two digits to the left of the decimal point. The result of this transformation is a line such as 'tan(cos(1.396 2.*[75.6759 80.4688 ...]).^5.*5.7168)', which comes from 'F(F(NOS)QON)Q'.

This logic runs until several conditions have been satisfied. The conditions are: that the number of random numbers in the line is equal to the number of elements in the original string, that the line does not contain any INF (infinity) or NaN (Not a Number) or imaginary values, and that the sum of the values does not exceed 10,000. Then, with an approximately 33% chance, the sets are added together and a decryptor is produced which subtracts them. Otherwise, the sets are subtracted, and a decryptor is produced which adds them together. In the latter case, the order of the two sets is chosen randomly (that is, 'a+b' or 'b+a').

GARBAGE IN, GARBAGE OUT

The same routine is used both for encryption of parts and for generating the garbage code. In the case of garbage code, a random section of the code might be encrypted. In some cases, a randomly generated name will be used, but the minimum length of the name is reduced to a single character. In other cases, one of the defined names will be used. The garbage code is fully functional, and will construct decrypted code, but it would likely concatenate the parts in the wrong order. However, the garbage code is never executed by the virus. It exists simply to camouflage the real code. There are two methods of garbage code generation, but they differ only in the chance of generating particular sequences. If a randomly chosen value is below the 'threshold', then the garbage code is encrypted. Otherwise, it is stored as plain text. There is a 'bug' in this behaviour, which causes the garbage code to be distinguishable in some cases from the real code. It doesn't help much for detection purposes, but it does allow those lines to be skipped.

If at least half of the code parts have been marked, then a selection of real names will be used as garbage names. This is safe because the code is never executed. If the first method of garbage code generation is in use, then there are several conditions which are checked. With an approximately 10% chance, the garbage string is assigned to a real name. Otherwise, with an approximately 20% chance, and if at least one garbage name exists, the garbage string is prepended to a random name. Otherwise, with an approximately 40% chance, and if at least three garbage names exist, then two variables are concatenated in a random order. Otherwise, with an approximately 20% chance, the garbage string is appended to a random name. Otherwise, the garbage string is compiled from between three and nine random lower case characters, in the range of 'a' to 'y'. The string might be encrypted in the same way as for the real code. The result is then assigned to a real name.

If the second method of garbage code generation is in use, then with an approximately 30% chance, the garbage string is prepended to a random name. Otherwise, it is appended to a random name.

STAGE 3

Once the code has been processed completely, the next stage of obfuscation begins. There is a 50% chance that the appearance of 'if' statements will be changed. There is an approximately 70% chance that a random number of spaces between zero and seven will be used per true clause. Otherwise, four spaces will be used. There is an approximately 70% chance that the spacing in false clauses will be the same as the spacing in true clauses. Otherwise, a random number of spaces between zero and seven will be used. The virus generates 11 unique random names for use in producing the conditional statements. It avoids producing names that match existing variable names or keywords. The random names are between five and 19 lower case characters long, in the range of 'a' to 'y'.

FUNCTION CONSTRUCTION METHODS 2 AND 4

For each line of real code, the virus chooses one of five possible methods. For the second and fourth method, the virus generates a random name which we will call 'R1'. The random name is between two and five lower case characters long, in the range of 'a' to 'y'. The virus avoids producing a name that matches any of the functions 'sin', 'cos', 'exp', or 'atan', or keywords. The virus makes a copy of this name for later use. We will call the copy 'R2'.

The virus starts with the algorithm 'SOS'. The virus parses the algorithm while any of the 'S', 'O' and 'F' elements remain. 'S' is replaced by either R1 or R2 or another algorithm chosen randomly from a set. With a 50% chance, and if R1 still matches R2, the virus generates a replacement random name for R2. The random name is between two and five lower case characters long, in the range of 'a' to 'y'. The virus avoids producing a name for R2 for which either R1 or R2 is a substring of the other, or that matches any of the functions 'sin', 'cos', 'exp', or 'atan', or keywords. 'O' is replaced by an operator chosen randomly from a set. 'F' is replaced by a function chosen randomly from a set. If the number of parenthesis pairs exceeds 25, then the line is considered to be complex 'enough', and the two algorithms are disabled to force the transformation to complete sooner. This forms a partial transformation of the line. Further processing occurs later. Given those rules, the grammar looks like this:

```
S -> (SOS) | F(S) | R1 | R2
O -> ".*" | "+"
F -> sin | cos | exp | atan
```

METHOD 1

The first method is all about matrices. The virus generates two vectors that contain a randomly chosen number of entries between three and seven, and one vector that contains the square of the number of entries in the first vector. Each of the entries will contain a randomly chosen floating-point number in the range of 0 to 1. The virus chooses a matrix algorithm randomly from the set: dyadic product, direct matrix, 'toeplitz', 'vander', 'pascal', 'magic', 'hilb', 'invhilb', 'wilkinson' or 'rosser'. In the case of the 'toeplitz'

or 'vander' matrix algorithms, the algorithm will be applied to the first vector. In the case of the 'pascal', 'magic', 'hilb', 'invhilb' or 'wilkinson' matrix algorithms, the size of the first vector will be used as an immediate value, but the actual vector will not be used any further. The 'rosser' algorithm returns a constant matrix, and no parameters are needed.

With an approximately 34% chance per round, the virus prepends a function chosen randomly from the set: 'sin', 'cos', 'sinh', 'cosh', 'exp', 'tan', 'sqrt', 'real' and 'imag'. This check is performed randomly between one and three times.

The virus chooses a name from the variable list and prepends a function chosen randomly from the set: 'sum', 'max' and 'min'. With an approximately 66% chance, the virus will contract the matrix to a vector, and then the vector to a scalar. The virus uses a random floating-point number for the scalar, which might be a negative number.

The virus repeats the logic above, beginning with the 34% chance per round of prepending a function from the first set, and finishing by prepending a function from the second set. Then, the virus repeats the logic but with only a 15% chance per round. The logic is executed one more time, using the 15% chance per round again.

The virus determines which is the larger of the first vector and the variable or value which was chosen second. The virus constructs an 'if' statement consisting of an inequality that contains a combination of the first vector and the variable or value, followed by 'true else false' clauses. The virus chooses randomly which clause will hold the real code and which will hold the garbage code. The 'if' statement will be constructed appropriately to always reach the real code. The result is a pair of lines such as:

```
pcuwsd=[28.7828 22.4722 17.9312 13.5236 1.5371 1.17
18.6505];
if((tan(max(pcuwsd)))<cosh(sum(exp(sum(hilb(7))))))
```

METHOD 2

The second method is numerical integration. There is an approximately 40% chance that the virus will replace R2 with R1. The virus will choose two elements randomly from the set: 'pi', 'log(2)', 'sqrt(2)', 'sqrt(3)', 'float1', 'float2', 'float3', 'float4' and 'float5' (where float1–5 are floating-point numbers). The sign of the elements is chosen randomly. With an approximately 30% chance for each element, the sign will be negative. The virus places the smaller of the two values first, and the inequality will use 'quad' as its operator, for a one-dimensional integral. Otherwise, the virus will choose four elements randomly from the same set as described above. As above, the sign of the elements is chosen randomly. With an approximately 30% chance for each element, the sign will be negative. The virus will separate the four elements into two pairs, and place the smaller of the two values in each pair first. The inequality will use 'dblquad' as the operator, along with the second pair of elements, and R2, for a two-dimensional integral.

After constructing the expression, the virus evaluates it. The virus checks that the integration takes *at least* 100ms to complete, and that it succeeds. The accuracy of the result is improved until either the expression takes 'long enough', or the tolerance is too small for a solution to be found. If the tolerance is too small, then the expression is abandoned. The result is a line such as:

```
quad(@(kwam)kwam+sin((kwam+(((((kwam+kwam)+kwam)+k
wam).*cos(atan((kwam+kwam))))+kwam))),0.099816,0.2
3802,1e-20)
```

or

```
dblquad(@(jbxf,ckt)sin(jbxf)+sin(jbxf),log(2),0.8952,
0.92058,sqrt(2),1e-17)
```

but the variables 'kwam', 'jbxf' and 'ckt' in these examples are components of an anonymous function, and are not defined anywhere else.

METHOD 3

The third method is interpolation. The virus creates a vector between one and 54 values long, containing random floating-point numbers. The numbers are multiplied by 1,000 to produce values which have up to three digits to the left of the decimal point. The virus creates an expression that requires interpolation to solve. With an approximately 40% chance, the interpolation uses a cubic spline method. The interpolation will be performed on a random subset of the vector. The result is a line such as:

```
interp1(yxrwj,21.2865,'spline')
```

In this example, 'yxrwj' is a variable that was defined earlier.

METHOD 4

The fourth method is a differential equation. The virus creates an ordinary differential equation in three stages. The first stage constructs the right side of the equation. The second stage solves the differential equation and returns the solution array. The time span is chosen randomly, but with a very limited range. The lower bound is in the range of -3 to 3, and the upper bound is in the range of the lower bound plus 1 to 4. The initial condition is a random floating-point number between 0 and 3.9999. The ODE function file is used during this stage to check the size of an interval. If the interval is too

small then the equation will be abandoned. A small interval indicates the presence of a singularity. If the size of the interval is acceptable, then the third stage uses interpolation to check the value. If the maximum time is reached while attempting to solve the equation, then the solution probably contains a singularity and the equation will be abandoned. The result is a set of lines such as:

```
msbqybsbtjnpibjnbt=inline('cos(cdx).*udq','udq','cdx
');
[rjwqtivdaes,yniaaytxnfpswott]=ode45(msbqybsbtjnpibjn
bt,[1 4],3.4908);
interp1(rjwqtivdaes,yniaaytxnfpswott,1.8901)
```

where each line can be separated by garbage instructions and other inequalities using values that were constructed earlier.

METHOD 5

For the fifth method, the virus starts with the algorithm 'F(F(S))'. The virus parses the algorithm while any 'S' remains. 'S' is replaced either by 'R' or by another algorithm chosen randomly from a set which introduces the 'F' and 'D' symbols. If the number of 'F' and 'D' elements exceeds 10, then the line is considered to be complex 'enough', and all algorithms are disabled to force the transformation to complete sooner. The virus parses the resulting algorithm while any of the 'D', 'F' or 'R' elements remain. 'F' is replaced by a function that accepts one parameter, chosen randomly from a set of 46(!) standard *MATLAB* mathematical functions, covering many areas. 'D' is replaced by a function that accepts two parameters, chosen randomly from a set. 'R' is replaced by a random floating-point number between -5 and 4.9999. The grammar looks like this:

```
S -> F(S) | D(S,S) | R
F -> sin | ... | sec | ... | exp | ... | log | ...
| sqrt | ... | abs | angle | conj | imag | real |
unwrap | fix | floor | ceil | round | sign | airy |
expint
D -> hypot | dot | cart2pol | pol2cart | atan2
R -> floating-point number
```

If the number of parenthesis pairs is fewer than 25, then the line is considered to be acceptable, otherwise the expression is abandoned. The result is a line such as:

```
sinh(asech(angle(acosh(cart2pol(4.5123,cosh(acot(angl
e(cos(acsch(-3.4196)))))))))
```

METHODS 2–5

For all but the first method, the virus checks the result of the expression for two conditions. Specifically, the virus checks that the result of the expression is less than or equal to a random subtraction value, or greater than or equal to a random addition value. While both of those conditions remain true, the virus will adjust the subtraction and addition values by random increments with precision ranging from five to nine decimal places, until both conditions are false. With an approximately 60% chance, the addition or subtraction value will be placed in a random variable which will be used later. Otherwise, the value will be used directly. An 'if/else' statement will be constructed such that one clause will contain the real instruction, and the other will contain the garbage instruction. With a 50% chance, the 'if/else' statement will compare the result of the inequality with the subtraction value. Otherwise, the statement will compare the result of the inequality with the addition value. With a 50% chance, the comparison in the 'if/else' statement will be reversed so that the 'true' and 'false' clauses will be reversed. The result is a line such as:

```
if(aeeynvlqvsfivdjip>acot(atan(unwrap(tanh(acoth(a
tan(tanh(floor(nextpow2(hypot(2.3959,dot(1.3456,0.
72016)))))))))))
```

IF-THEN-WHAT ELSE?

If a block of code consists of an 'if/else' statement, then there is a 50% chance that any of the 'true' clause, the 'else' statement, the 'false' clause, and the 'end' statement will be concatenated to the following component part. This is applied to all of the component parts, such that the block might be collapsed into a single line. If the appearance of 'if' statements was chosen to be randomly changing, then there is an approximately 70% chance that a random number of spaces from zero to seven will be used per true clause. Otherwise, four spaces will be used. There is an approximately 60% chance that the spacing in false clauses will be the same as the spacing in true clauses. Otherwise, a random number of spaces from zero to seven will be used.

At this point, the virus walks backwards through the code and assigns the real code lines to the final code array. With an approximately 70% chance per line of real code, the virus will assign a variable definition line to the final code array. If all of the real code has been assigned but some variables have not, then with an approximately 50% chance per iteration, the virus will assign one of the remaining variables. This action is repeated until all variables have been assigned. The result of this is a randomly ordered set of variable definition lines.

SEEK AND DESTROY

Finally, the virus searches the current directory for *MATLAB* module files. For each file that is found that is less than 1,000 bytes long (this check filters out infected files, which cannot possibly be that small), and is not the ODE function

file that belongs to the virus, the virus opens and reads the entire file, line by line. The virus searches each line for '%' (comment) and '...' (line continuation), except if they appear inside quotation marks. If '%' is seen, then the virus discards the entire line. If '...' is seen, then the virus appends the next line to the current line at the point where the '...' began, and rescans the line repeatedly until no more '...'s are seen.

After the first pass has completed, the virus identifies potential insertion points. If the current line is not inside a logic block, then it is considered to be a potential insertion point. The virus searches each line for any one from the set: 'if', 'for', 'while', 'try', 'switch' and 'parfor'. If one is found, then it must either be at the exact start of a line, or immediately following spaces, semicolons, or tabs. It must also be either the only thing on the line (which seems to be illegal, at least for earlier versions of *MATLAB*), or followed immediately by spaces, a left parenthesis or tabs in order to be considered valid. This marks the beginning of a logic block. Once inside a logic block, the virus searches each line for 'end'. If it is found, then it must either be at the exact start of a line, or immediately following spaces, semicolons, or tabs. It must also either be the only thing on the line, or be followed immediately by spaces, semicolons or tabs in order to be considered valid.

After the second pass has completed, a subset of the potential insertion points is chosen randomly as actual insertion points. The virus inserts the code backwards (which is now forwards, because of the backwards assignment, as described previously) while there is code left to insert. Since there can be fewer insertion points than parts of the virus, multiple virus lines might be grouped at a single insertion point. Finally, the combination is written back to the file. There will always be at least one host line before one virus line. If there are more insertion points than parts of the virus, then the remaining host code is appended after the last virus line.

CONCLUSION

This virus appears to have been written in response to a possible detection method for a previous version, whereby the plain text virus body could be produced by concatenating the individual parts. That is not possible with this version because of the difficult expressions that would need to be solved in order to decrypt the parts. However, the very nature of the polymorphism in this version essentially substitutes one kind of plain text for another. There are plenty of interesting and constant characteristics that can be identified very quickly. This allows us to perform a deeper inspection of only the most likely candidates without the performance hit of spending a long time looking at random files. This is great for us, and obviously not the result that the virus writer was expecting.