# MALWARE ANALYSIS 1

## READ THE TRANSCRIPT

*Peter Ferrie*
Microsoft, USA

Metamorphism seems to be the holy grail for virus writers in general. It is a step above polymorphism, which is, in turn, a step above oligomorphism. The assumption is that it is more difficult for an anti-virus engine to detect a metamorphic virus than it is to detect a 'lesser' virus. As a result, there have been attempts to implement metamorphism on multiple platforms, the latest one being JavaScript, in the form of JS/Transcript.

### OVERVIEW

There are essentially two ways in which a virus can implement metamorphism – the first is for the virus to disassemble itself, gather information about each of the instructions, discard any garbage instructions, 'optimize' itself to the simplest form (which might be impossible, given certain combinations of modifications), and then perform an alteration.

The second is for the virus to carry a copy of its own source code (for scripts or binaries), or the simplest form of its compiled code (for binaries). However, even in the simplest case, the requirement remains for the virus to gather information about each line of code (for scripts) or instructions (for binaries).

JS/Transcript uses a variation of the second method.

The virus carries its own source code described in a meta-level language, which includes the critical information about each line of code – specifically, the variable dependencies between lines. The virus 'compiles' this code and then derives the next generation JavaScript code from there. The derivation is performed in three steps: pre-processing, code generation and post-processing. The pre-processing phase includes renaming variables, permutating line order, and random function creation. The post-processing phase includes variable placement within the virus body, which includes the possibility of creating arrays of variables.

Every meta-level line has the form:

```
(ident|restr)code
```

where *ident* is the identifier, *restr* is the set of 'restrictions' (that is, a prerequisite or dependency list), and then the code follows.

The identifier is a locally unique name which is used as part of the dependency list for subsequent lines within the same scope (that is, within the block of code declared by *if*, *while*, or *function*, or within the main body). The dependency list specifies the names of all identifier lines that must have executed before this line can execute. It is used primarily to ensure that required variables have been assigned meaningful values before they are used.

### MEET YOUR REPLACEMENT

The virus begins by creating an array containing the numbers 0-255. These are used by the value generation code later. The virus searches for all references to a variable named *$CreateObject$*, the intention being to replace each one randomly with either *WScript.CreateObject* or *new ActiveXObject*. (These two statements are equivalent ways of creating an instance of an object, and are essential for the virus to replicate.) However, there aren't any variables with such a name, so this code never executes. This is not a bug, it's more like an unimplemented feature. As a result, the virus uses only *WScript.CreateObject*. It is possible that the functionality was present in an earlier version but somehow missed being included in the final version. That leaves us with only one true bug in the virus – which, most surprisingly, is not in the metamorphic engine itself.

The virus makes a copy of the meta-level language version of the code. It replaces the first reference to *)var* with *)def*, then replaces the first *)while(var* with *)while(*, and then the first *)while(* with *)def* in the copy. Then it searches for an instance of *)def* in the copy. This replacement allows the variable name declarations to be located easily. The *)var* form is a variable declaration after the dependency list; the *)while(var* form is a variable declaration for a *while* loop after the dependency list (the virus does not support *do{}while()* loops, only *while(){}* loops). However, it is not known why the virus uses the double-replacement for the *while* form, since all variables are either using the *)while(var* form to declare themselves, or using the virus meta-level *$* form for an existing variable, and the virus is interested only in the *)while(var* form.

The virus isolates the variable name, and then chooses a new random string for the replacement name. The string consists of between six and 15 random-case alphabetic characters. The virus replaces all occurrences of the old name in the original code with the new name. The virus performs the same *)def* replacement on the next instance, and then searches for its location. It repeats this action until all variables have been replaced. The code is not optimized for speed – which becomes particularly apparent when the permutation begins (see below).

The virus makes another copy of the meta-level language version of the code. It searches for a reference to *)function*

in the copy. It isolates the function name, and then chooses a new random string for the replacement name. The virus replaces all occurrences of the old name in the original code with the new name. The names of each of the function parameters are also replaced with random strings, along with all references to the parameters within the function.

## FUNCTION CREATION

The virus creates up to 35 functions (75 in the first generation) that will perform an essential operation. The virus begins with the function template '(SOS)O(SOS)'. For each 'S' in the template, with a 25% chance, the 'S' is replaced with '(SOS)'. Otherwise, the virus replaces the 'S' with 'X'. There is no limit to how many times the '(SOS)' might be inserted into the template. While this could, in theory, lead to heap exhaustion, it is unlikely to occur in practice.

Once all 'S's have been replaced with 'X's, the virus searches within the string for the 'O's. For each 'O' in the template, the virus replaces it with a randomly chosen operator from the set: '+', '-', '*' and '%'. Both '+' and '-' have an approximately 33% chance of being chosen, and '*' and '%' each have an approximately 16% chance of being chosen. The result is a template such as:

```
(X%X)-(X-X)
```

or

```
((X+X)*X)+(X%X)
```

The virus creates an array of between two and five random strings. For each 'X' in the template, with a 50% chance, the virus replaces it with a randomly chosen string from the array of strings. Otherwise, the virus replaces it with a random number in the range of 0-255. Once all 'X's have been replaced, the virus removes from the array of random strings any entry that has not been used. If any entries remain, the virus generates a new function. The name of the function is a new random string. The parameters of the function are all of the remaining entries. For each of the parameters, the virus assigns a random number in the range 0-255, and then executes the function. If the result is a value in the range 0-255, then the virus assigns the function to the corresponding entry in the array of increasing numbers that it created at start-up. The virus performs this action 100 times, thus creating a collection of equation functions that return particular values. These equation functions can be used during the code construction whenever a particular value is needed (see below). If at least one set of parameters returns a valid value, then the virus saves the function for use later.

The virus chooses another random string. This one is assigned to the variable that holds the meta-level language version of the virus code. All references to the original variable name are replaced with the new name. The meta-level language version is a single line with internal lines delimited by '__'. The virus splits the code into an array of lines, and then passes the array to the permutator function. Interestingly, the virus makes improper use of the 'slice()' method in several places, by passing it no parameters. This simply returns the original array, so it is not really a bug. It is not known what was intended here.

## PERMUTATOR

The virus splits each line of code further into its component parts of identifier, dependency list and actual code. The permutator function is interested in logic blocks that are declared by *if*, *while* and *function*. It extracts the contents of the blocks recursively until the keywords are no longer found. What remains is the functional body of the block which is to be permutated. The permutator function chooses a random line order while preserving the semantics of the dependency list. As a result, lines can be swapped or separated if they do not depend on each other, which gives enormous potential freedom for alteration, but actually there are relatively few lines in the code that do not depend on those preceding them almost immediately.

## CREATEBLOCKOFCODE (#")

The virus examines each line of code. It checks whether the line contains the *#"* sequence. This is used to declare a literal string. The virus generates a replacement string for the content between the *#"* and *"#* characters. The algorithm for the replacement follows:

The virus chooses a random number of up to approximately one sixth of the length of the string. In the unlikely event that the chosen number is larger than 1,000 (which would require an initial string of at least 6,007 characters in length and it could be that 'short' only with a vanishingly small chance – more realistically, the string would need to be much longer), the virus chooses a random number of up to approximately one fiftieth of the length of the string. The chosen number is the number of pieces into which the virus intends to split the string. In preparation for splitting, the virus inserts two '@' characters at each place where the string will be split later. The locations are

chosen randomly, and as more '@' characters are inserted, the chance increases that at least one of the split locations will match the location of an '@' character. The result of this will be that when the string is finally split, some of the substrings will be empty. The virus splits the string once all of the locations have been chosen, and removes the '@@' characters at the same time.

For each of the substrings, with an approximately 38% chance (100% chance in the first generation), the virus replaces all of the apostrophes with a *String.fromCharCode(39)* sequence, but makes no further alterations.

For the approximately 62% chance remaining, with approximately 22% chance, the string is passed back to the function to be split further. The result is assigned to a variable with a randomly chosen name. With a 4% chance, the variable is assigned to another variable with a randomly chosen name. If the reassignment occurs, then the chance increases to approximately 7% that it will occur again, and the chance remains constant at that point.

## NUMERIC REPRESENTATION

For the approximately 49% chance remaining if the string has not been altered yet, the virus converts each character of the substring to a numeric representation and wraps them in a *String.fromCharCode()* statement, separated by commas. With a 75% chance for each of the characters, the virus uses one of the equation functions that it generated earlier, if any exist for the corresponding value. If no equation function exists for the value, or in all cases for the first generation, the bare value is used.

For the 25% chance remaining, with approximately 13% chance each, the virus chooses one operator from the set: '+', '-' and '/', and applies it to a randomly chosen value acting on the original value. The result is passed back to the function for potential further modification. The randomly chosen value is then passed to the same function for the same reason.

In all cases, there is a 1% chance that the number is assigned to a variable with a randomly chosen name. If that occurs, then the chance increases to approximately 7% that the variable is assigned to another variable with a randomly chosen name. The chance remains constant at that point that it will occur again.

Note that the major percentages (38%, 22%, and 49%) are valid only when building the outer layer. Once the magic 'victory' symbol is seen (see below), the 38% chance block is avoided entirely, the 22% chance is increased to an approximately 43% chance, and the 49% chance is increased to an 86% chance. The reason for avoiding the 38% chance block is to prevent any part of the meta-level language version of the code from appearing in a plain-text form.

## CREATEBLOCKOFCODE (#n)

The virus checks whether the line contains the *#n* sequence. This is used to declare a number. The virus generates a replacement value for the content between the *#n* and *n#* characters. The number replacement algorithm is the same as the 'numeric representation' algorithm described in the *#"* section above, with two exceptions. The first is that if the original number is larger than 10,000, then the number is not altered further. The second exception is that if the original number is negative, then there is a 13% chance that the number is not altered further.

## CREATEBLOCKOFCODE (#O)

The virus checks whether the line contains the *#O* sequence. This is used to declare an object. The virus extracts the name of the object and the name of any method that is being called. If a method is being called, then with an approximately 67% chance, the object is not altered. Otherwise, the virus extracts the name of the object. It searches for all *#x* sequences, and erases them and any corresponding *x#* sequences. The virus creates a function with a random name which returns the original object, so *object.method(args)* becomes *function()->method(args)* where *function()* returns the object. If the original object was assigned to a variable, then the variable is passed to the function. The function parameter is a random string. The parameter is returned.

## CREATEBLOCKOFCODE (#x)

The virus checks whether the line contains the *#x* sequence. This is used to declare an execution sequence. The virus extracts the string from the sequence. With an approximately 85% chance (89% chance in the first generation), or approximately 95% if the string contains *eval(* (96% chance in the first generation), the string is not altered. Otherwise, if the string contains *eval(* already, or with a 40% chance, the virus splits the variable name into pieces separated by '@', using the algorithm described above. It uses *eval()* to reconstruct the name, and appends the parameters to the result. For the 60% chance remaining, the virus creates a function with a random name, which executes the sequence and then returns the result. The virus

calls the *createexecution* function recursively, and passes it the names of the local variables in the execution sequence, to transform them, too.

## CREATEBLOCKOFCODE (if)

The virus checks whether the line begins with *if*. This is used to declare a conditional execution block. If the operator is '==', then with a 50% chance (a 75% chance in the first generation), the virus converts the *if* block to a *switch()*, with a case element devoted to the *true* clause of the *if* block, and a default element devoted to the *false* clause of the *if* block, if it exists. Otherwise, the virus uses *if*, and *else* if applicable.

For the *if* case, the virus separates the components of the condition. For the left side of the condition, with an approximately 54% chance, or an approximately 83% chance if the string contains *eval(*, the string is not altered.

If the string is chosen to be altered, then with a 20% chance, or a 40% chance if the string contains *eval(*, the entire left side of the condition is assigned to a variable with a randomly chosen name.

For the 80% chance – 60% chance if the string contains *eval(* – remaining, then with a 20% chance, or always if the string contains *eval(*, the virus splits the variable name into pieces separated by '@', using the algorithm described above. It uses *eval()* to reconstruct the name, and appends the parameters to the result. If the string does not contain *eval(*, then the virus creates a function with a random name, which executes the sequence and then returns the result.

If the right side of the condition is a number, then with a 4% chance (approximately 8% chance in the first generation), the value is assigned to a variable with a randomly chosen name. If the reassignment occurs, then the chance increases to 6% (16% in the first generation) that it will occur again, and the chance remains constant at that point.

If the right side of the condition is not a number, then with a 78% chance, or a 92% chance if the string contains *eval(* (approximately 54% and 83% chance respectively in the first generation), the string is not altered.

If the string is chosen to be altered, then with a 20% chance, or a 40% chance if the string contains *eval(*, the entire right side of the condition is assigned to a variable with a randomly chosen name.

For the 80% chance – 60% if the string contains *eval(* – remaining, then with a 20% chance, or always if the string contains *eval(*, the virus once again splits the variable name into pieces separated by '@', using the algorithm described above. It uses *eval()* to reconstruct the name, and appends the parameters to the result. If the string does not contain *eval(*, then the virus creates a function with a random name, which executes the sequence and then returns the result.

With a 50% chance, the virus emits the left and right sides in that order. Otherwise, it reverses the order and 'inverts' the operator (for example, 'a<b' becomes 'b>a'). The virus does not have the ability to swap the order of the true and false clauses. Finally, it uses the *createblockofcode* algorithm to further transform the lines in the respective clauses of the *if*, or the case and default blocks in the switch.

## CREATEBLOCKOFCODE (while)

The virus checks whether the line begins with *while*. This is used to declare a loop (the virus does not accept *for* loops, but it can produce them as part of its transformation process). The implementation is the same as for the right side of an *if* block (the equivalent of the left side is unaltered in all cases because it might contain a variable declaration, which the virus handles in a different way). With a 50% chance, the virus emits a *while* statement. The virus calls the *createblockofcode* function recursively, and passes it the body of the *while* loop, to further transform the lines in the block. If there is an 'action' to perform (for example, updating a value in a variable that controls when to exit the loop), then with a 20% chance (a 60% chance in the first generation), the virus splits the variable name into pieces separated by '@', using the algorithm described above. It uses *eval()* to reconstruct the name, and appends the parameters to the result.

If the virus does not emit a *while* statement, then it emits a *for* statement instead. It calls the *createblockofcode* function recursively, and passes it the body of the *for* loop, to further transform the lines in the block. The *for* statement generation makes use of a special variable that controls the variable declaration. Its presence here has no purpose, and is probably a left-over from when the logic creation function was made to be shared between the *while* and *for* loop handling.

## CREATEBLOCKOFCODE (c)

The virus checks whether the line begins with *c*. This is a special instruction that can perform multiple operations, such as adding or subtracting numbers, or concatenating strings. If the right operand is a one (presumably to either increment or decrement – there are other possibilities, but what happens next means that the virus does not support them), then with an approximately 33% chance, the virus uses the 'double-operator' form (that is, '++' for increment, or '--' for decrement). If it does not use the double-operator

form, then with a 50% chance, the virus uses the 'operator=' form (that is, '+=' or '-='), regardless of the value of the right operand. If it does not use the 'operator=' form either, and if the right operand is an 'n', to represent an arbitrary number, then with a 50% chance, the virus reverses the order of the parameters – for example, 'a+b' becomes 'b+a'. (Note that for an operator such as subtract, divide or modulus, this returns the wrong value. This behaviour is only a potential bug, however, since the virus does not use the divide or modulus operators, nor the subtraction of an arbitrary number.) Otherwise, the virus emits the operands in the original order.

If the operation is the 'operator=' form, then with a 45% chance, or a 90% chance if the string contains *eval(*, the string is not altered.

If the string is chosen to be altered, then with a 20% chance, or a 40% chance if the string contains *eval(*, the second variable is assigned to a variable with a randomly chosen name.

For the 80% chance – 60% chance if the string contains *eval(* – remaining, then with a 20% chance, or always if the string contains *eval(*, the virus splits the variable name into pieces separated by '@', using the algorithm described above. It uses *eval()* to reconstruct the name, and appends the parameters to the result. If the string does not contain *eval(*, then the virus creates a function with a random name, which executes the sequence and then returns the result.

If the operation is in neither the 'double-operator' nor the 'operator=' form, then with an approximately 63% chance, or an approximately 87% chance if the string contains *eval(*, the first variable is assigned to a variable with a randomly chosen name.

For the 80% chance – 60% chance if the string contains *eval(* – remaining, then with a 20% chance, or always if the string contains *eval(*, the virus once again splits the variable name into pieces separated by '@', using the algorithm described above. It uses *eval()* to reconstruct the name, and appends the parameters to the result. If the string does not contain *eval(*, then the virus creates a function with a random name, which executes the sequence and then returns the result.

This algorithm is applied to the second variable with identical percentages.

## CREATEBLOCKOFCODE (x)

The virus checks whether the line begins with *x*. This is used to declare an execution block. With an approximately 38% chance (25% chance in the first generation), the string is not altered.

With a 25% chance (approximately 38% chance in the first generation), and if the string does not begin with *return(*, the virus splits the variable name into pieces separated by '@', using the algorithm described above. It uses *eval()* to reconstruct the name, and appends the parameters to the result. If the string contains *eval(*, then it is not altered any further. Otherwise, the virus creates a function with a random name, which executes the sequence and then returns the result. The virus calls the *x* handler function recursively, and passes it the name of the function for possible further transformation. The transformation can include creating another function with a random name that calls the original function. This function chaining can happen repeatedly. While this could in theory lead to stack exhaustion, it is unlikely to occur in practice.

## CREATEBLOCKOFCODE (y)

The virus checks whether the line begins with *y*. This is used to assign a value to a variable. With a 20% chance (a 60% chance in the first generation), the virus splits the variable name into pieces separated by '@', using the algorithm described above. It uses *eval()* to reconstruct the name, and appends the parameters to the result.

## CREATEBLOCKOFCODE (def)

The virus checks whether the line begins with *def*. This is used to declare a global variable. The implementation is identical to that of *x*.

## CREATEBLOCKOFCODE (var)

The virus checks whether the line begins with *var*. This is used to declare a local variable. The implementation is identical to that of *y*.

## CREATEBLOCKOFCODE (function)

The virus checks whether the line begins with *function*. This is used to declare a function. The virus separates the components of the function into its name, its parameters, and its body. It calls the *createblockofcode* function recursively, and passes it the body of the function, to further transform the lines in the block.

## CREATEBLOCKOFCODE (victory)

The virus checks whether the line begins with *victory*. This is used to define the location where the meta-level language

version of the virus code is assigned to a global variable. With a 75% chance, the virus emits a *var* statement first. The virus splits the string into pieces separated by '@', using the algorithm described above.

## POST-PROCESSING

There is a block of code here that is reached in all cases, even though it is specific to only one case (perhaps something more was intended but not completed). If a *while* statement has been used, and if it contains a variable declaration, then the virus creates a list of candidate locations for inserting the variable declaration. This can appear after a semi-colon but not within braces, and it must appear prior to the first use of the variable. The variable declaration is placed in a randomly chosen location prior to the addition of the *while* statement.

## CREATEVARS

For each variable that was assigned a value, the virus finds the first use of the variable. The virus creates a list of candidate locations for inserting the variable assignment. This can appear after a semi-colon but not within braces, and it must appear prior to the first use of the variable. The variable assignment is placed in a randomly chosen location. In all generations after the first one, the virus chooses a random number of up to approximately one fifth of the number of variables. This becomes the number of arrays that the virus creates. A selection of variables is chosen randomly, and a subset of those are placed into arrays. A variable is a candidate for inclusion in an array if it is defined and then a value is assigned to it only once. With a 75% chance, the virus creates an anonymous function that simply returns the variable, inserts that into the array, and replaces the variable reference with a function call that is indexed in the array. Otherwise, the virus creates a function with a random name, which returns the variable.

For each of the functions, the virus chooses a random location in the code. This can appear after a semi-colon but not within braces, but there are no other restrictions, since all of the functions have global scope so they can even appear after the first reference to them.

## ...TO THOSE WHO WAIT (AND WAIT)

Everything up to this point could be considered a highly polymorphic decryptor for the virus source code. Of course, the true virus body is altered metamorphically, too. The virus produces one metamorphic representation of itself per run, and uses that representation to infect all files that it can find. This makes it a slow metamorph. Each run can take upwards of five minutes to produce a new copy – which makes it a *very* slow metamorph. More to the point, the host code has not been executed yet, which makes it an *extremely* slow metamorph. It is probably safe to assume that another iteration of the code will avoid this problem by spawning a copy of itself and passing a special string so that the host code can be executed first, using a technique similar to that used by the Lymer [1] virus.

The virus searches in the current directory (only) for files whose suffix is '.JS'. It opens and reads the entire file each time it finds one, no matter how large it is. The virus checks the length of the read data and skips the file if it is at least 150,000 bytes long. This serves as the infection marker, and is a very conservative value given that even the smallest infected file is probably over 1MB. If the file is small enough, the virus attempts to open it again in write mode, then prepend its code to the file. If the file has the read-only attribute set, then an exception will occur here and the virus will be terminated, because it does not use any exception handling to intercept the error. This could be considered the one true bug in the virus. After the enumeration has completed, the virus runs the host code.

## CONCLUSION

The assumption is that detection of a metamorphic virus is more difficult than detection of an ordinary virus for an anti-virus engine. While this is certainly true, the act of making a virus metamorphic introduces so much 'noise' that, in a sense, detection is not always as difficult as the virus writer intended. Since the resulting code contains so much obfuscation, it rarely resembles the code of a regular program. This allows us to find all kinds of artefacts which attract our attention, and that in turn allows us to spend more time scanning, with no impact on ordinary users who tend not to have such samples. We can even take this further – hundreds of hours of the virus writer's work can be undone in a matter of a few hours by an anti-virus researcher. The existing metamorphic viruses have been detected in a matter of days (once the time was devoted to writing a detection, of course), in contrast to the months of work put in by the virus writer. Given that, you have to wonder why the virus writers bother.

## REFERENCES

[1] Ferrie, P. Like a bat out of hell. Virus Bulletin, May 2012, p.9. http://www.virusbtn.com/pdf/magazine/2012/201205.pdf.