VIRUS ANALYSIS 2

IT'S ZELL(D)OME THE ONE YOU EXPECT

Peter Ferrie and Heather Shannon
Symantec Security Response, USA

It was a Tuesday and it was sunny outside, but I was inside waiting for my email client to finish retrieving messages. It was stuck on one mail that had a huge attachment: a sample of W32/Zellome.

W32/Zellome arrives as an email attachment. It seems to exist only to demonstrate its polymorphic engine, since the worm component is messy and platform-dependent.

EXTREME PROGRAMMING

The polymorphic engine takes an idea that was first used by W32/Apparition, but takes it much further. W32/Apparition carried its own Pascal source code, which it dropped on machines on which a Pascal compiler was found. Apparition would insert garbage instructions into that source code, before directing the compiler to compile it. Zellome, by contrast, carries the compiler as a component of its polymorphic engine. Additionally, Zellome's polymorphism is implemented in an unusual way: by using a genetic algorithm.

NATURAL SELECTION

Evolutionary algorithms, also known as genetic algorithms, are based on the idea of biological evolution. By combining characteristics from a predefined set (genes), and altering parts of them randomly (mutation), new offspring is produced with new characteristics. The less fit of these tend not to pass their genes on to succeeding generations. At least, that's the idea. There have already been viruses that have used this technique, including W32/Simile (see *VB*, May 2002, p.4).

Zellome uses a genetic algorithm for a different purpose. Traditionally in this analogy, the virus is treated as a species, replications of the virus represent individuals, and 'fitness' is the ability to survive in an environment populated by hostile anti-virus software. For Zellome, the genetic algorithm is not a model of virus replication; rather, it is just a computational technique used to produce a polymorphic decryptor. The species is a set of functions, and 'fitness' is how close the function comes to producing the required output.

Ultimately, we come to the question: why use a genetic algorithm in the first place? This is difficult to answer, because the results are, essentially, no different from the output of a standard polymorphic engine. It is really no

more difficult to detect than normal polymorphic code. It is highly obfuscated, but it has constant characteristics. It is effective against emulators, with its many iterations and heavy floating-point usage, yet its extremely ugly compiled code is a giveaway.

INCUBATOR

The virus author calls the polymorphic engine an 'incubator'. Whenever the incubator is run, it begins by randomly displaying a message box identifying the virus author's name and his choice of the virus name (however, the engine is simply a modified version of a free tool written by someone else). Next, it will check if it was run from the %windows% directory. If it was not run from there, it will copy itself to the %windows% directory as 'incubator.scr' and create 'incubator.txt' that contains the name of the original file. Then the incubator executes itself from the %windows% directory and then deletes the file named in the 'incubator.txt' file.

KEPT AFLOAT

Before generating a new worm, the incubator encrypts the worm code that is stored in its .data section. The basic encoding scheme substitutes an 8-bit value, x, with a 32-bit float value, E(x), where E is a random quadratic function. The .C, .D and .E variants of the worm also preprocess the data with another polymorphic engine (the same one used in W32/Zelly.B, created by the same virus author), before applying the substitution encoding.

With the worm encoded as an array of floats, the incubator now needs a decryption routine to decode the encoding. There are several ways to do this: one could use hash tables, construct an interpolating polynomial, or use some algebraic manipulation to solve the quadratic equation. Instead, however, Zellome does it the hard way: it uses a genetic algorithm to 'grow' a decryptor. It generates random arithmetic expressions, then mutates and combines them until it finds one that happens to undo the encryption function for the 256 possible input values. This is a time-consuming task that can take anywhere from five minutes to half a day.

Once it finds a decryptor, Zellome generates C source code containing the encoded worm and a short decryption loop, interspersed with about a megabyte of garbage code.

WHAT'S COOKING?

Zellome starts the incubation process by generating an initial population of 16,384 expressions. The basic elements of the expressions are:

- binary operators: *, +, -, /
- unary functions: exp, sin, sqrt
- constant float or integer values
- pi (just another constant)
- a variable, 'X', representing the input to the function

An expression is represented as a list of 256 tokens, thus the size of the generated decryptor is limited.

No filtering is done on these expressions: duplicates, synonymous expressions, and obviously unsuitable candidates such as constant functions, are all allowed.

RANDAMN

Due to an improperly seeded random number generator, the initial expressions are not actually as random as they should be. It is possible for the incubator to generate the same list of initial expressions in subsequent runs, depending on the value of an uninitialized variable that is passed to srand(). It still produces a different decryptor each time, because the encryption function is generated before this call to srand().

DARWINIAN EVOLUTION

The incubator then begins the process of creating new generations from this initial population.

First, the current generation is checked for a suitable decryptor function. It estimates the fitness of each expression, and saves the value for later use. The 'fitness' of a candidate is the sum of absolute distances from the target values, multiplied by -0.01. Expressions that produce any extraordinary values (such as infinity or 'not a number') are assigned a fitness of -FLT_MAX, effectively eliminating them from further consideration. Particularly promising expressions are checked against each possible input. If the resulting values all fall within 0.5 of the target output, a decryptor function has been found, so control passes to the source generation routine.

If a decryptor is not found, Zellome proceeds to select the next generation.

The population is kept constant at 16,384 individuals.

- 15 per cent are unmodified members of the current generation, including the three fittest specimens.
- 50 per cent are mutated individuals from the current generation.
- 35 per cent are new 'children', constructed by combining existing expressions.

When Zellome selects an expression for propagation, mutation, or breeding, it chooses four at random, then picks

the fittest of the four. Duplicates are allowed, so the same expression may be used more than once.

The following kinds of mutation can occur:

- Replace one constant with another
- Replace a subexpression with a new random expression
- Change the order of arguments (for example, change 1/X to X/1)
- Simplify constant expressions (for example, replace sqrt(4.0) with 2.0)
- Replace an operator or function with another (for example, change sqrt(X) to sin(X))
- Switch subexpressions (for example, change (X + 1.0) * 2.0 to (X + 2.0) * 1.0)

To keep the expression size from running over the 256-token limit, large expressions (those with more than 64 tokens) are not subject to mutation, except for constant-substitution.

To 'breed' two expressions together, Zellome selects a subexpression from one parent, and replaces it with a subexpression from the other parent. There is a five per cent chance that the offspring will be mutated.

For example:

1st parent: (sqrt ((X) / (5)))

Subexpression: (X) / (5)

2nd parent: ((sqrt (X)) / (2.649156))

Subexpression removed: ((sqrt (X)) / (...))

Offspring: ((sqrt (X)) / ((X) / (5)))

This process continues up to 10,000 generations. In practice, about 50 to 150 usually suffice to find a decryptor, though it can take much longer.

In testing, it was observed that all of the decryptors found by this method contained a sqrt() subexpression. This may be explained by the quadratic encryption function: intuitively, if you want to invert Ax^2 + Bx + C, it makes sense to start by taking the square root of the input. By contrast, only 25 per cent of the decryptors contained sin(): periodic functions are unlikely to be useful when inverting polynomials.

GETTING RESULTS

The incubator creates a file, 'result.c', in the current directory. It writes a constant preamble declaring some functions and global variables, emits the decryption code to a buffer (to be written to the file later), and writes a series of random functions that contain array assignments and garbage code.

The array assignments initialize a buffer with the encoded worm. (These values are treated as floats during the decryption computation, but they are initialized as an array of integers, probably to prevent rounding error.) Zellome does the assignments in random order. After it writes the decryptor function, the remaining array assignments access the buffer through a pointer to a random location in the middle of the array; the purpose of this obfuscation is not clear, but one possible explanation is that it is to make it more difficult to see that one assignment belongs to the same region of memory as another assignment.

The garbage code consists of function calls, arithmetic expressions, assignments to local variables, 'if' statements with random conditions, and 'for' loops that execute up to 1,000 times. A function call may optionally be enclosed in an 'if' or 'for' block, or both, but not if the function contains or calls any necessary code, such as an assignment function. This ensures that all of the non-garbage code is called exactly once.

Rather than assigning random names to functions and variables, Zellome observes the following naming convention:

l#### – local variable

p#### – parameter

d#### – array assignment or decryption function

f#### – other function

if#### – inline function

where the ####s are numbers assigned in increasing order.

(This systematic naming convention, together with the constant appearance of parts of the code, suggest that the author's design goals did not include concealing the source code from detection.)

MALFUNCTION

Functions take up to seven arguments with random types. They always return a value: there are no void functions. The return values are either saved to dummy variables, or discarded; they are not relevant to the decryption process.

Function calls can be any of the following:

- other random functions in the source file
- sqrt, exp, sin, abs, acos, asin, atan, atof, cos
- rand, srand
- fopen (but not fclose)
- malloc (allocating up to 65535 bytes – but not free)
- strcmp, strlen

- SetCurrentDirectoryA, CreateDirectoryA, CopyFileA, DeleteFileA, MoveFileA

The code ignores any errors returned by any of these functions. Since the parameters are well-formed, none of the functions would cause an exception to occur, so there is no need for critical error detection. However, the use of CreateDirectoryA() does create random directories, and the use of DeleteFileA() and MoveFileA() could, in exceptional circumstances, result in the deletion or renaming of real files.

Nestled in the middle of this random code, there is a surprisingly readable, un-obfuscated decryption loop which applies the decryption expression, saves the decoded worm to a stack buffer, and transfers control to the worm.

GET BUFF

Curiously, the code generator allocates a 10,000,000-byte buffer to contain this decryptor code, though the decryptor part itself is only a few hundred bytes long, and the entire source file is seldom longer than 1,500,000 bytes. It is possible that the goal was to generate more of the source 'genetically', but the design was scaled back to a smaller sub-problem: use a genetic algorithm to generate a numeric expression, but produce the rest of the code through normal means.

To give structure to the code, Zellome creates a series of call trees, each containing an arbitrary number of garbage functions, and one non-garbage function as a leaf node. It emits each tree to the file separately, first declaring each function that appears in the tree, then writing the functions, in breadth-first order, starting from the top of the tree.

Finally, it generates a top-level tree that calls all of the others – first the assignment parts, then the decryptor part. The root node for this tree is supposed to be WinMain, but due to a bug, this is not always the case. When it generates a new node for the call tree, it first decides whether the node will be a garbage function, and only later checks whether it should be WinMain. If it creates the non-garbage function as the root node of the last tree, the tree-generation routine thinks it has finished, so it exits without producing WinMain. In this case, the source will not compile.

DROP YOUR BUNDLE

The incubator drops the compiler files at this point, in order to compile the produced source code. The .A and .B variants are missing one key file, though (mspdb60.dll), so unless the file is present already on the system, all compilations will fail. When compiling, the incubator uses compiler switches chosen at random from a set that it carries.

The compiler switches that are used cover different areas. There are switches for code optimization (optimize for size, or for speed, or disable optimization entirely); for the expansion of inline functions (all, some, or none); for the emitting or omitting of frame pointers; for the presence or absence of exception handling; and the type of exception handling, if present.

Finally, the incubator will either compress the file with the copy of UPX that it carries, or append the incubator to the created file, but not both, then run the result. Since there is no detection of multiple instances, new replicants will continue to be generated for as long as the incubator is part of any replicant.

THE WORM HAS TURNED

The worm component begins by retrieving the list of APIs that it will use, some of which are not used, including two which are critical to prevent multiple copies of the worm running at the same time.

It copies itself to the %system% directory as 'bigfish.scr', then hooks the execution of Task Manager. This is done by creating the registry key 'HKLM\Software\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\taskmgr.exe', and the value 'Debugger', whose data are set to point to the copied file.

This technique was first described by the virus writer GriYo as 'Execution redirection', and published in the eighth issue of the *29A* virus-writing magazine. The idea is that *Windows NT*-based systems run the process named in the 'Debugger' value, expecting it to control the application that is named in the key. The worm does not run the original file afterwards. This change continues to work in Safe Mode, so it is necessary to rename the file instead, in order to run it manually.

To improve the chance of being executed, the worm also attempts to create a value in the 'HKLM\Software\Microsoft\Windows\CurrentVersion\Run' key, which it names 'bigfish', and whose data also point to the copied file. However, the use of the seemingly incorrect API (RegSetValue() instead of RegSetValueEx()) causes *Windows* to create a subkey instead of a value. The result is that there is no execution via that method on any platform apart from *Windows 2000*. Perhaps this is the platform that the virus author uses, which is why he didn't notice the problem.

REGISTER NOW

After hooking the registry, the worm queries the registry value 'HKCU\Software\Microsoft\Internet Account

Manager\Default Mail Account', then uses that value to query the 'Accounts' key for the 'SMTP Server' value. This server will be used to send mail, if possible.

The email attachment can arrive in one of two forms. One form is simply the worm file, the other form includes the polymorphic engine as appended data. If the polymorphic engine is present, the worm will detach it into the current directory and execute it at this time.

The worm always uses the name 'incubator.scr' for the detached file. The detached file is an independent component and executes without reference to the worm file.

In any case, the worm will encode itself into base64 format – perhaps surprisingly, using the standard dictionary algorithm. It might be considered surprising because an alternative algorithm was published in the seventh issue of the *29A* magazine, which, at only 59 bytes in length, is smaller than the base64 dictionary itself. In fact, the worm code is taken from another virus by another author, with only a few modifications (single subject, etc.), but the same bugs.

The worm collects email addresses from two sources, and keeps a list of every email address that it finds. The worm does not avoid duplicated addresses. The first source of addresses is the file referenced by the registry key 'HKCU\Software\Microsoft\WAB\WAB4\Wab File Name'.

JUST BROWSING

The second source is the browser cache directory, within which all subdirectories will be searched for files whose extension is one of 'htm', 'asp', or 'xml'. For any such file found, if its size is between 512 bytes and 80 kilobytes, the worm searches within the file for the 'mailto:' string.

A number of bugs exist in this code – the most important of which is that, while parsing the file, the buffer pointer is updated to skip any address that was found, but the variable that holds the number of remaining bytes is not adjusted correspondingly. This can cause the routine to crash if at least one address exists, because the buffer pointer will fall off the end of the buffer. The crash is intercepted by the worm code, though, so the worm will continue to execute, but exit the address collection routine.

If the routine does not crash, potential addresses are examined for the presence of disallowed characters. If any such characters are found, then the worm will adjust the pointer in its collected address list to allow the next address to overwrite the invalid one. However, if no other addresses exist in the same file, then a bug causes the next address to be appended to the invalid address, instead of overwriting it.

HELO WORLD

After looking for email addresses, the worm attempts to resolve the address of the SMTP server. There is a critical bug here, which is the result of an incorrect assumption about the layout of certain networking structures. The worm assumes that the hostent structure, returned by the gethostbyname() API, is followed immediately by the address list. In fact, this is true for all *Windows* versions prior to *Windows XP*. In *Windows XP/2003*, there is a null pointer at that location. Thus, in all variants prior to .E, if run on *Windows XP/2003*, the code crashes at this point and never sends mail. However, on any earlier version of *Windows*, the code does work correctly. Additionally, the 'incubator.scr' file will still be running, if it was present.

In the event of a successful resolution, the worm will connect to the SMTP server. It was intended that the worm would check the return values from the server, however some of the branch instructions were removed, leaving compare instructions whose results are ignored. These compare instructions relate to the client initiation. The most likely reason for their removal is that the worm's domain string is malformed, and the worm author might not have worked out why a server would not return the expected response.

The worm chooses a random number prior to sending the message. This random number would be used to select between different sender addresses, subjects, message bodies and attachment names, however all of the conditions point to the same respective texts. This results in an email that always appears to come from 'Don Quijote y Sancho Panza', with subject 'juas juas cuidadin con el attachhhhrrrr!!!!!' (which translates roughly as 'heh heh watch out for the attachment!!!!!'), a message body of

```
juas juas juas peaso de bicho que lleva el attach!!!
juas juas!!! ;D

Vallez\29a
```

(which translates roughly as 'heh heh heh what a tiny bug is carrying the attachment!!! heh heh!!!'), and an attachment name of 'soyunpeasodebichooooooo.scr' (roughly 'I am a tiny buuuuuuug.scr'). The attachment will be the worm file. The worm will send a single email, but to multiple recipients. The recipients are all addresses found in the address book, and no more than 40 of the addresses found in files. After sending the email, the worm will exit.

CONCLUSION

As an unusual example of self-compiling malware and a novel misapplication of artificial intelligence, Zellome is an interesting specimen, but its many bugs and painfully slow execution time prevent it from working as a practical worm. In evolutionary terms, this species is heading for extinction.