VIRUS ANALYSIS

GATT GOT YOUR TONGUE?

Peter Ferrie
Symantec Security Response, USA

As operating systems have become more secure (or at least less insecure), virus writers have started to attack applications instead. One of the most popular tools for an anti-virus researcher is the *Interactive Disassembler* (*IDA*), and its IDC scripting language has become the latest target, thanks to W32/Gatt.

THE IDC LANGUAGE

.IDC files are script files that can control *IDA* by using the IDC scripting language. The IDC language is very C-like in appearance, and supports functions, variables, etc. – all of the things that one would expect from a good scripting language. However, as with *Microsoft*'s VBScript and JScript scripting languages, IDC files are compiled at the moment they are requested to run, and the resulting binary form is executed directly in memory. There is even a built-in Compile function, to perform dynamic compilation of IDC files.

GATTMAN AND BOBBIN

W32/Gatt is a polymorphic entry-point obscuring infector of these IDC files. It begins by allocating a one-megabyte(!) buffer for the new decoder. That might sound like overkill, but in fact the generated decoders often require more than half of that buffer. However, there is nothing in the generator to prevent a decoder from exceeding the buffer. If that were to happen, the virus would simply crash, since it contains no exception handling code.

After the allocation, the virus attempts to create a file mapping of itself, and here is the first bug: even if the mapping operation fails, the virus still attempts to infect files. Another, similar bug follows immediately: even if the attempt to map a view of the file fails, the virus still attempts to infect files. Additionally, if any handle cannot be closed for any reason, the allocated block is never freed explicitly.

WARP FACTOR NINE

Assuming that all goes well, the virus will generate a new decoder. Despite appearances, the decoder is only lightly polymorphic. The polymorphic engine is capable of producing random comments of both the '/**/' and '//' style, including comments that span multiple lines. For the first comment style, which is designed to support multiple lines already, no special handling is required. For the second comment style, which is intended to be only a single line, the virus ends the line with a backslash line-continuation character.

Each of the tokens can also be split randomly across lines, by using the backslash line-continuation character. In an extreme case, it would be possible for the virus to produce files where only a single character appears on each line, but this is unlikely to occur. The '/**/'-style comments can also appear between the tokens. Finally, non-token elements – variables, and string elements – have their case mapped randomly.

This is essentially all that the polymorphic engine does. The only other variation is in the way in which the virus chooses to rebuild itself.

TESTING, ONE, TWO... OOPS

Not surprisingly, the polymorphic engine is full of bugs. The decoder begins with a conditional expression, which tests whether a variable that the virus declares has a value of 0. The virus carries seven variations of this expression: two 'if' forms, two 'while' forms, and three 'for' forms. The bug occurs when selecting the form to use: the engine uses the 'test' instruction instead of the 'cmp' instruction.

This bit-wise comparison results in two variations of the expression that cannot be selected. One of those unselectable blocks contains a bug anyway: a missing semicolon character means that the line would generate a syntax error during compilation, and the execution will not occur. As if that wasn't bad enough, one of the remaining selectable conditional expressions also contains a bug. That bug is also related to a semicolon character. However, this time the bug is not that the semicolon character is missing, but that it is in the wrong place. Again, the line would generate a syntax error during compilation, and the execution will not occur.

THE WRITE WAY

The virus works by converting IDC files into droppers of a *Windows* executable file. This executable file is what performs the infection of other files – IDC files infected with W32/Gatt do directly infect other IDC files. To try to hide the executable file within the IDC file, the virus encodes the executable file into randomly sized blocks, and writes them out individually. This is as opposed to some viruses for other file formats, which declare an array of some kind to hold the entire file as a single block.

In the case of W32/Gatt, eight-bit values can be written by using 'fputc', followed by the literal character, or encoded in ‘0x’ form. 16-bit values can be written using the ‘writeshort’ function, and 32-bit values can be encoded using the ‘writelong’ function. These last two functions only accept the value in ‘0x’ form. These functions also accept a parameter that describes the endianness of the value. The virus selects the endianness randomly, and encodes the value in the appropriate order.

Otherwise, values can be written using the ‘writestr’ function. Another bug exists here: if the ‘writestr’ function is used to write the final character in the file, the engine will crash.

THE SEARCH BEGINS

Once the new decoder has been generated, the virus begins the search for files to infect. The file enumeration is done by using the usual recursive subdirectory searcher. The virus wants to find any file whose suffix is ‘.IDC’. The difference here is that the suffix is not compared directly. Instead, the virus uses the SHA-1 algorithm to create a hash of the suffix, and compares that hash to one that the virus carries. This might have slowed down analysis a little bit, to determine the file type of interest, if the virus author hadn’t made it quite clear what kind of file the virus wanted to infect.

The virus has no infection marker. The nearest thing to an infection marker is a check of the size of file that has been found. Any file larger than 419,430 bytes (0x66666 in hex) is considered to be infected. If a file is not infected already, then the virus searches within it for the string ‘static’, which the virus assumes is the start of a subroutine. If that string is found, then the virus examines the text between the first left and last right brace characters that it sees in that subroutine, counting all of the semicolon characters that it sees.

The virus also watches for the ‘for’ token, since it also contains semicolon characters, but they must not be counted. The virus recognizes the last right brace by incrementing a brace count for each left brace that is seen after the first one, and decrementing the count for each right brace that is seen. Once the count reaches zero, the virus stops looking.

Once the last right brace is seen, the virus chooses randomly from the count of semicolon characters, and inserts the virus code after the nth semicolon, which makes the virus entry-point obscuring. A critical bug occurs here: if any file is infected, the stack is unbalanced because of some leftover code. Specifically, the parameters for a particular API have been pushed onto the stack, but presumably during ‘optimization’ of the code, the API call was moved into a subroutine. This subroutine pushes the parameters locally, so the old parameters remain on the stack. Because of this bug, the virus crashes immediately after a single infection. Perhaps the virus author tested only on a single file at a time, and so never noticed the problem.

MAKING A HASH OF THINGS

If the current file is not an .IDC file, then the virus hashes the full filename and compares it to a list of five hashes that the virus carries. The reason for this check was clear even before the hashes were decoded: recognizable packer switches are present in the virus body, and though they are never used, it gave me a clue about the probable filenames. Three of the hashes were easy to guess, and they correspond to three runtime compressors (EXE32PACK.EXE, PEPACK.EXE, UPX.EXE). The other two yielded very quickly after a brute-force attack. One is a file manipulation tool called VGALIGN.EXE, but the other is an unknown tool called SPEC.EXE.

If one of these files is found, then the virus attempts to copy itself into the directory that contains that file. A bug exists here: the copy will fail if both files are present in the same directory, and in that case, the virus will keep searching for files.

If the copy succeeds, then the virus executes the file that corresponds to the hash, passing the virus filename as a parameter. The idea here is to use one of these tools to change the appearance of the file, and then to regenerate the decoder using the new file. However, yet another bug exists here: process execution is asynchronous, but the virus does not wait for the new process to complete before attempting to access the virus file again. Thus, the original virus file is used to generate the decoder, resulting in all encoded files having the same appearance.

CONCLUSION

On the day that the virus author released the virus, he posted a message on his website that said the virus ‘will be very hard for AVers to detect’. Later that same day, we started detecting it. The following day, the virus author changed the message to one that said the virus ‘will not be released’. I’d like to think that it’s not a coincidence – it might look polymorphic to him, but it doesn’t to me.

W32.Gatt

Type:	Polymorphic entry-point obscuring file infector.
Size:	16,384 bytes (EXE), varies (IDC).