MALWARE ANALYSIS 1

DOIN' THE EAGLE ROCK

Peter Ferrie
Microsoft, USA

If a file contains no code, can it be executed? Can arithmetic operations be malicious? Here we have a file that contains no code, and no data in any meaningful sense. All it contains is a block of relocation items, and all relocation items do is cause a value to be added to locations in the image. So, nothing but relocation items – and yet it also contains W32/Lerock.

Lerock is written by the same virus author as W32/Fooper (see *VB*, January 2010, p.4), and behaves in the same way at a high level, but at a lower level it differs in an interesting way.

EXCEPTIONAL BEHAVIOUR

Like Fooper, the virus begins by walking the Structured Exception Handler chain to find the topmost handler, and at the same time registers a new exception handler which points to the host entrypoint. Once it has found the topmost handler, the virus uses the resulting pointer as the starting location in memory for a search for the MZ and PE headers of kernel32.dll. Once it has found the headers, the virus parses the export table to find the APIs that it needs for infection.

The first problem in Lerock's code is identical to the first bug in Fooper's code: in *Windows Vista* and later, the topmost handler points into into ntdll.dll rather than kernel32.dll. As a result, the virus crashes on these platforms, because it assumes that the APIs it needs for infection will be found, and falls off the end of a buffer because they do not exist.

HAPI HAPI, JOY JOY

If the virus finds the PE header for kernel32.dll, then it resolves the required APIs. It uses hashes instead of names, but the hashes are sorted alphabetically according to the strings they represent. This means that the export table only needs to be parsed once for all of the APIs, rather than parsing once for each API (as is common in some other viruses). Each API address is placed on the stack for easy access, but because stacks move downwards in memory, the API addresses end up in reverse order in memory.

LET'S DO THE TWIST

After retrieving the API addresses from kernel32.dll, the virus initializes its Random Number Generator (RNG). Like Fooper, Lerock uses a complex RNG known as the 'Mersenne Twister'. In fact, the virus author has used this RNG in every virus for which he requires a source of random numbers.

The virus then allocates two blocks of memory: one to hold the intermediate encoding of the virus body, and the other to hold the fully encoded virus body. The virus decompresses a file header into the second block. The file header is compressed using a simple Run-Length Encoder algorithm. The header is for a *Windows* Portable Executable file, and it seems as though the intention was to produce the smallest possible header that can still be executed on *Windows*. There are overlapping sections, and 'unnecessary' fields have been removed. The virus then allocates a third block of memory, which will hold a copy of the unencoded virus body.

The virus searches for zeroes within the unencoded memory block and keeps a count of them. The zeroes will be skipped during the encoding process, which is the next step.

RELOCATION ALLOWANCE

The virus chooses randomly among the bytes in its body until it finds one whose value is not zero. For each such byte that is found, the virus stores the RVA of the byte within the encoding memory block, along with a relocation item whose type specifies that the top 16 bits of the delta should be applied to the value. The result of this is to add one to the value. The reason why this occurs is as follows:

The virus uses a file whose ImageBase field is zero in the PE header. This is not a valid loading address in *Windows*, so when *Windows* encounters such a file, it will relocate the image (with the exception of *Windows NT*, which does not support the relocation of .exe files at all). However, the location to which the relocation occurs is different for the two major *Windows* code-bases. *Windows NT*-based versions of *Windows* (specifically, *Windows 2000* and later) relocate images to 0x10000. *Windows 95*-based versions (*Windows 9x/Me*) relocate images to 0x400000. It is the *Windows NT*-based style of behaviour that the virus requires.

When relocation occurs, *Windows* calculates the delta value to apply. This value is calculated by subtracting the old loading from the new loading address (this can be a negative value if the image loads to a lower address than it requested). In this case, the new loading address is 0x10000, and the old loading address is 0, so the delta is also 0x10000, or to be more explicit, 0x00010000. Thus, the top 16 bits of the delta are 0x0001. It is this trick that allows the virus to adjust the value by one.

The virus decreases the value of the byte within the unencoded memory block. If the value reaches zero, then the virus decreases the number of bytes left to process. The virus also increases the corresponding value in the intermediate encoding memory block.

At this point, the virus decides randomly if it should apply special relocation items to the surrounding values, and, if so, what type of items to apply. The virus can produce a relocation item that adds 0x40 to any byte that is in the location one byte after the current position, but it has a side effect (not all of the bits are maintained) on three of the four bytes beginning at the current position, so the virus selects this type only if the next three bytes are still zero. The virus subtracts 0x40 from the value of the byte within the unencoded memory block. If the value reaches zero, then the virus decreases the number of bytes left to process.

The virus can also produce a relocation item that adds 0x20 to any byte that is in the location 13 bytes after the current position, but it has the same side effect as above, on a much larger scale (10 out of 16 bytes are affected), so the virus selects this type only if those 10 bytes are still zero. The virus subtracts 0x20 from the value of the byte within the unencoded memory block. If the value reaches zero, then it decreases the number of bytes left to process.

This is where the intermediate encoding memory block comes into play. It is a representation of the relocation items that have been applied at the current moment in time. The buffer begins by containing all zeroes, and the values are increased as the relocation items are applied. The ultimate aim is to reduce all of the original non-zero bytes to zero, thus avoiding the need to have any code in the file. All that is left is an empty section.

The encoding process repeats until all of the non-zero bytes have been encoded. The random ordering and type selection of the relocation items produces an essentially polymorphic representation of the virus body. Once the encoding process has completed, the virus creates a file called 'rel.exe', places the size information into the section header, writes the encoded body, then runs the resulting file. Finally, it transfers control to the host.

DROPPING YOUR BUNDLE

The dropped file begins by walking the Structured Exception Handler chain to find the topmost handler, and at the same time registers a new exception handler, which points to the host entrypoint. As above, the code locates kernel32.dll in order to resolve the APIs that it needs for replication. Unlike the W32/Fooper, this virus uses only Unicode-based APIs, since the *Windows* code base that it requires is also Unicode-based.

After retrieving the API addresses from kernel32.dll, the virus attempts to load 'sfc_os.dll'. If that attempt fails, then it attempts to load 'sfc.dll'. If either of these attempts succeed, then the virus resolves the SfcIsFileProtected() API. The reason the virus attempts to load both DLLs is that the API resolver in the virus code does not support import forwarding. The problem with import forwarding is that, while the API name exists in the DLL, the corresponding API address does not. If a resolver is not aware of import forwarding, then it will retrieve the address of a string instead of the address of the code. In the case of the SfcIsFileProtected() API, the API is forwarded in *Windows XP* and later, from sfc.dll to sfc_os.dll. Interestingly, the virus supports the case where neither DLL is present on the system, even though that can occur only on older platforms – which it does not support.

The virus then searches for files in the current directory and all subdirectories, using a linked list instead of a recursive function. This is simply because the code is based on existing viruses by the same author – this virus does not infect DLLs, so the stack size is not an issue. The virus avoids any directory that begins with a '.'. This is intended to skip the '.' and '..' directories, but in *Windows NT* and later, directories can legitimately begin with this character if other characters follow. As a result, such directories will also be skipped.

FILTRATION SYSTEM

Files are examined for their potential to be infected, regardless of their suffix, and will be infected if they pass a strict set of filters. The first of these is the support for the System File Checker that exists in *Windows 2000* and later.

The remaining filters include the condition that the file being examined must be a *Windows* Portable Executable file, a character mode or GUI application for the *Intel* 386+ CPU, not a DLL, that the file must have no digital certificates, and that it must not have any bytes outside of the image.

TOUCH AND GO

When a file is found that meets the infection criteria, it will be infected. The virus resizes the file by a random amount in the range of 4–6KB in addition to the size of the virus. This data will exist outside of the image, and serves as the infection marker.

If relocation data is present at the end of the file, the virus will move the data to a larger offset in the file, and place its code in the gap that has been created. If no relocation data is present at the end of the file, the virus code will be placed here. The virus checks for the presence of relocation data by checking a flag in the PE header. However, this method is unreliable because *Windows* ignores this flag, and relies instead on the base relocation table data directory entry.

The virus increases the physical size of the last section by the size of the virus code, then aligns the result. If the virtual size of the last section is less than its new physical size, then the virus sets the virtual size to be equal to the physical size, and increases and aligns the size of the image to compensate for the change. It also changes the attributes of the last section to include the executable and writable bits. The executable bit is set in order to allow the program to run if DEP is enabled, and the writable bit is set because the RNG writes some data into variables within the virus body.

The virus alters the host entrypoint to point to the last section, and changes the original entrypoint to a virtual address prior to storing the value within the virus body. This will prevent the host from executing later, if it is built to take advantage of Address Space Layout Randomization (ASLR). However, it does not prevent the virus from infecting files first. The lack of ASLR support might be considered a bug but for the fact that ASLR was only introduced in *Windows Vista*, which the virus does not support. What is strange, though, is that changing the entrypoint affects DLLs in the same way. Thus, if an infected DLL is relocated because of an address conflict, then it, too, will fail to run.

APPENDICITIS

After setting the entrypoint, the virus appends the dropper code. Once the infection is complete, the virus will calculate a new file checksum, if one existed previously, before continuing to search for more files.

Once the file searching has finished, the virus will allow the host code to execute by forcing an exception to occur. This technique appears a number of times in the virus code, and is an elegant way to reduce the code size, as well as functioning as an effective anti-debugging method.

Since the virus has protected itself against errors by installing a Structured Exception Handler, the simulation of an error condition results in the execution of a common block of code to exit a routine. This avoids the need for separate handlers for successful and unsuccessful code completion.

CONCLUSION

The virus author called this technique ‘virtual code’, which is quite an accurate description. However, the technique lends itself to simple detection by anti-virus software, given the randomly ordered relocation items that are applied multiple times to bytes in an empty section. Of course, future virus writers might try to bypass detection by ordering the relocation items sequentially (which would make it less suspicious, but reduce the polymorphism at the same time). Alternatively, they might fill the section with legitimate-looking code and transform that instead (which would make it less suspicious, but potentially require even more relocation items). However, what remains is still a set of relocation items that are applied multiple times to bytes in a section, and there’s no getting around that one.