

# MALWARE ANALYSIS

## DOIN' THE EAGLE ROCK... AGAIN!

Peter Ferrie  
Microsoft, USA

In 2010, *Virus Bulletin* published a description of W32/Lerock [1]. It described a technique which was called 'virtual code' by the virus author. However, at the time the virus was written (2007), it was already incompatible with what was then the current version of *Windows* (*Windows Vista*). The release of *Windows 7* in 2009 introduced another incompatibility. In 2012, the virus author updated Lerock – purportedly to support *Windows 7* (and, presumably, *Windows Vista*), but apparently insufficient testing led to a critical bug being overlooked. The release of *Windows 8* in 2012 introduced a fundamental incompatibility. Despite that, it is interesting to take another look at the virus, this time W32/Lerock.B.

### EXCEPTIONAL BEHAVIOUR

The virus begins by retrieving the base address of kernel32.dll. It does this by walking the InLoadOrderModuleList from the PEB\_LDR\_DATA structure in the Process Environment Block. The address of kernel32.dll is always the second entry in the list – at least it is in *Windows XP* and later. Previously, the virus walked the Structured Exception Handler chain to find the topmost handler, which used to point to kernel32.dll until the release of *Windows Vista*. This change in behaviour solves the major compatibility problem with *Windows Vista*, and one of the problems with *Windows 7*, but it introduces another for *Windows 2000* and earlier.

The virus assumes that the InLoadOrderModuleList entry is valid and that a PE header is present there. This assumption is unfortunate in the case of *Windows 2000*, because there is no longer any registered Structured Exception Handler to deal with the issue that arises on that platform.

### 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). Lerock uses a complex RNG known as the 'Mersenne Twister'. In fact, the virus author has used this RNG in nearly 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 inclusion of an import table containing a reference to a real 'es.dll' DLL (and specifically, one that contains a reference to 'kernel32.dll') means that the file is intended to work on *Windows 2000*. However, the act of loading that DLL instead of loading 'kernel32.dll' directly, means that kernel32.dll is not the second entry in the InLoadOrderModuleList list – 'es.dll' is. As a result, the virus can no longer run on *Windows 2000*. Furthermore, the use of that particular DLL (which does not exist on any version of *Windows* prior to *Windows 2000*) instead of the 'gdi32.dll', which was used by other viruses created by the same author, and which exists in all versions of *Windows*, means that the virus can no longer run on *Windows NT* or earlier, either.

An interesting but quite unrelated observation can be made at this point about the es.dll file. It is the Event Services service, and it contains two exports with amusing names: 'RegisterTheFrigginEventServiceDuringSetup' and 'RegisterTheFrigginEventServiceAfterSetup'. Researchers who have analysed the 'miniFlame' malware should recognise these two names. One of the components of miniFlame is a DLL that also exports two functions with these names, along with names that match the other exports from es.dll. It appears that the authors of miniFlame based that component on the *Windows 2000* version of the file, because the names were removed in the version that runs on *Windows XP*.

### RELOCATION ALLOWANCE

The virus allocates a third block of memory, which will hold a copy of the unencoded virus body. The virus progresses linearly along the bytes in its body until it finds one whose value is not zero. This is in contrast to the

previous version, which performed the search randomly. 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 three to the value. The reason why this occurs is as follows:

The virus uses a file whose ImageBase field is 0xffff0000 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 image is relocated 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 address 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 0xffff0000, so the delta is 0x30000, or to be more explicit, 0x00030000. Thus, the top 16 bits of the delta are 0x0003. It is this trick that allows the virus to adjust the value by three.

The reason why the virus chose that value for the old loading address is twofold. Firstly, the value of zero, which was used by the previous version of the virus to produce a delta of 0x0001, is not supported by *Windows 7*. However, any value which corresponds to non-user-space (that is, any value in the range of 0x7ffe000 to 0xffff0000) is accepted. Secondly, the delta must be an odd number in order for the virus to be able to construct all other values from it. Three is the smallest odd number that can be produced with the new load address restriction.

If the byte-value within the unencoded memory block is zero, then the virus moves to the next byte, until there are none left to process. Otherwise, it subtracts three from the value of the byte (relocation type 1), and applies any carry to the following bytes until no carry remains. The virus also decreases 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 ( $\text{delta} * 0x40 = 0xc0$  for the delta of 0x0003) 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.

Therefore, the virus selects this type only if the next three bytes are within the range of the virus body, if the second byte of the four has an unencoded value of at least 0xc0, and if all four encoded bytes are currently zero. The check for the four zero bytes is unusual. The code zeroes the lowest byte of the register that holds the values, then increments it. It is not known why the virus author did not simply assign the value of one to the byte. This code appears in the previous version, too. If the checks pass, then the virus subtracts 0xc0 from the value of the byte (relocation type 5), and applies any carry to the following bytes until no carry remains.

The virus can also produce a relocation item that is intended to add ( $\text{delta} * 0x20 = 0x60$  for the delta of 0x0003) 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, and this is the subject of the bug mentioned above and described below). The virus selects this type only if the next 15 bytes are within the range of the virus body, if the 13th byte of the 15 has an unencoded value of at least 0x60, and if those 10 encoded bytes are still zero. If the checks pass, then the virus subtracts 0x60 from the value of the byte (relocation type 9), and applies any carry to the following bytes until no carry remains.

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 decreased 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 fixed ordering reduces the polymorphism greatly compared to the previous version, but the type selection of the relocation items still produces an essentially polymorphic representation of the virus body.

## WINDOWS ATE MY RELOCS

The critical bug that exists in the code is exposed by the handling of relocation type 9. The change was actually introduced in *Windows Vista*, and relocation type 9 is the only one that demonstrates the effect because it is the only one that the virus uses which treats the delta as a 64-bit number (note that relocation type 10 also treats the delta as a 64-bit number).

The change is that in *Windows XP* and earlier, the delta is a sign-extended 32-bit value (0x10000-0xffff000=0x00030000), but in *Windows Vista* and later, it is a fully 64-bit value (0x10000-0xffffe000=0xffffffff00030000). As a result,

the new value from a relocation type 9 is no longer solely (delta\*0x20), but rather 0x0800??00007fffffff, where ‘??’ is (delta\*0x20). This has a significant effect on the decoding process.

One reason for fixing the order of the relocation items in this version of the virus is simple: size. Since the virus is performing a subtraction operation, this can affect the neighbouring bytes in a significant way. Specifically, if any given byte has a value which falls below zero, because it is not originally a multiple of three, then a carry is generated which must be applied to the following byte(s). If the following byte is a zero, then its value becomes -1. This change requires that 85 relocation items be generated for the byte to transform it back to a zero. However, the act of initially converting the zero to a -1 also generates a carry which must be applied to the following byte, which then requires another 85 relocation items for that byte, and so on. So a series of zeroes which should be skipped becomes a multiple of 85 relocation items each. The problem is made worse if the selection is random, since the first selected value might not fall below zero when reached linearly, if the previous byte generated a carry that caused the selected value to become zero.

Another reason for fixing the order of the relocation items in this version might well be time. It is a simple matter to allocate an initial region of memory to hold the relocation data, followed by a guard page in case the transformation size expands wildly. When the guard page is reached, it can be mapped in, and the region can be resized to include it as a commit page followed by another guard page. This act can be repeated until all of the relocation data has been processed, but it might take quite a noticeable amount of time to complete.

However, as noted above, the release of *Windows 8* in 2012 introduced a fundamental incompatibility: relocation types 1, 5 and 9, are no longer supported. Any file that contains any of these relocation items will fail to run on that platform. Perhaps it is a coincidence that they happen to be the three types that the virus uses, but perhaps not. It is interesting to note that relocation type 4 – which behaves exactly like type 1, though occupying twice the space of the standard relocation item – remains supported. Thus, the virus could have been composed entirely of these exotic relocation type 4 items – which, while no longer polymorphic, would still be likely to challenge most analysis tools.

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, and then runs the resulting file. Finally, it transfers control to the host.

## FACT VS FICTION

Another interesting note is that a previously published article [2] also examined the first version of the virus with respect to *Windows 7*, but made some quite dramatically incorrect conclusions. The authors made the claim that Address Space Layout Randomization (ASLR) makes the relocation technique of the virus unworkable, but in fact, ASLR has nothing to do with *Windows* relocating the image. While it is true that ASLR makes the virus unworkable, this is simply because the virus transfers control to the host entry point via its virtual address rather than via its relative virtual address or a relative branch. As a result, when an ASLR-supporting file is infected, it will crash if it is relocated.

The authors of [2] also made the claim that the relocation types 1, 5 and 9, were no longer supported. It seems more likely that they encountered the type 9 bug and extrapolated from there (and were unlikely to have known about the impending changes in *Windows 8*, since it had not been released at the time of writing). They produced their own *Windows 7*-compatible implementation, but it used a delta of 0x0002, which, as described above, cannot be used to produce all possible values. Thus, their version had a code section which contained actual values. They used relocation type 3 only, and so their polymorphism resulted from the random selection of values to encode to a random degree, rather than encoding all of the values.

## DROPPING YOUR BUNDLE

The dropped file begins by registering a Structured Exception Handler, and then walking the InLoadOrderModuleList from the PEB\_LDR\_DATA structure in the Process Environment Block. As above, the code locates kernel32.dll in order to resolve the APIs that it needs for replication. 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 only occur 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 ‘.’. The intention is 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 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 4KB to 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* essentially ignores this flag, and relies instead on the base relocation table data directory entry (more accurately, if the flag is set, then *Windows* will disable ASLR for the process, but will still relocate the image if the value of the ImageBase requires it).

The virus increases the physical size of the last section by the size of the virus code, and 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 entry point to point to the last section, and changes the original entry point 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 ASLR. However, it does not prevent the virus from infecting files first. The lack of ASLR support in this version is a bug, given the attempt at ‘*Windows 7* compatibility’.

## APPENDICITIS

After setting the entry point, 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 cause itself to be terminated 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, even this version of the technique lends itself to simple detection by anti-virus software, given the many relocation items that are applied multiple times to bytes in an empty section – and there’s still no getting around that one.

## REFERENCES

- [1] Ferrie, P. Doin’ the eagle rock. *Virus Bulletin*, March 2010, p.4. <http://www.virusbtn.com/pdf/magazine/2010/201003.pdf>.
- [2] Fortunato, A; Passuello, M; Giacobazzi, R. Relock-based vulnerability in *Windows 7*. *Virus Bulletin*, August 2011, p.16. <http://www.virusbtn.com/pdf/magazine/2011/201108.pdf>.