MALWARE ANALYSIS

TWINKLE, TWINKLE LITTLE STAR

Peter Ferrie
Microsoft, USA

Sometimes a virus gets it completely wrong (see *VB*, October 2008, p.4). Sometimes a virus gets it mostly 'right', but sometimes that's only because the virus in question is a collection of routines taken from other viruses which got it mostly right. That is exactly what we have here, in W32/Satevis.

The virus begins by determining its location in memory. This makes it compatible with Address Space Layout Randomization (ASLR), though the technique has existed for far longer than ASLR. However, instead of using the common call-pop technique to determine the location, the virus uses a call, but then uses an indirect read from the stack via a string instruction. In the past, this kind of alternative method would have avoided some heuristic detections, but these days the call-pop method is so common in non-malicious code that this obfuscated method might be considered suspicious. In any case, there are few anti-malware engines now that would rely on such a weak detection method.

KERNELIFEROUS

The virus sets up a Structured Exception Handler (SEH), and does so correctly (unlike Zekneol, see *VB*, November 2009, p.4). Then the virus walks the host import table, looking for a DLL whose name begins with 'kernel32'. This leads into what we might consider to be the first bug in the code, though it does not come into play until later. The bug is that since nothing further is checked, the name of the DLL that the virus finds could be 'kernel32<any string>'. For example, 'kernel32foo.bar', and it will be accepted. While this is very unlikely to occur, it is still a bug.

Once a kernel32-style DLL has been found, the virus retrieves the address of the first API that is imported from it and uses that address as a starting point for a search for the MZ and PE headers. Assuming that the headers are found, the virus parses the export table directly to retrieve the addresses of the APIs that it needs in order to infect files. For each API that kernel32 exports, the virus determines the length of the API name and calculates the CRC32 value using a routine that was written for 16-bit CPUs and which has been copied blindly for years by virus writers around the world. The virus searches its entire list of checksums for a match, which is a very inefficient method. Someone clearly didn't pay attention in computer science class. This action is repeated until all of the needed APIs have been located.

The virus also carries a little anti-debugging routine. One trick is an intentional divide-by-zero, which should cause an exception that the virus will intercept. In the past, some CPU emulators in anti-malware engines did not support such tricks. That might have been a problem in 1998, but these days support is widespread. The virus then attempts to open two *SoftICE* driver devices by name. However, this routine has also been copied blindly for years by virus writers, despite the fact that it hasn't worked since 2004. This is described more fully in *VB*, February 2009, p.4.

ONWARD AND FORWARD

The virus retrieves the address of the SfcIsFileProtected() API from sfc.dll, if that DLL is available (it was introduced in *Windows 2000*), using the GetProcAddress() API instead of parsing the export table directly. The use of the GetProcAddress() API avoids a common problem regarding import forwarding. The problem 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 this case, support for import forwarding (which the GetProcAddress() API provides) is necessary to retrieve the SfcIsFileProtected() API from sfc.dll, since it is forwarded to sfc_os.dll in *Windows XP* and later.

The virus searches for files within the current directory and is interested in files whose suffix is 'EXE', 'CPL' or 'SCR'. For each such file that is found, the virus checks if the SfcIsFileProtected() API is available. If so, then the virus gets the full pathname of the file, and 'converts' it from ASCII to Unicode. However, the conversion is done using a routine in the virus that simply takes an eight-bit value and stores a zero-extended 16-bit value. This obviously doesn't correctly convert any character that is not part of the seven-bit US-ASCII set, but the virus author probably doesn't care about such things anyway. After the conversion, the virus checks if the file is protected, and will not infect the file if it is.

REDUNDANT SYSTEMS

If the file is not protected, then the virus removes any read-only attributes, opens the file and queries its size. This is despite the fact that the file size was included as part of the information that was returned when the virus found the file in the first place. The virus avoids infecting files that are smaller than 16KB or larger than about 64MB, along with files whose size is a multiple of either 113 or 117 (see below). This style of infection marker was introduced years ago by members of the 29A virus-writing group, whose works appear to have influenced this virus writer.

If the file still appears to be infectable, then the virus queries its time stamps. This is despite the fact that the times (creation, last access and last write) are also available as part of the information that was returned when the virus found the file.

The virus opens the file and checks for the 'MZ' and 'PE' headers, along with several other fields. A minor bug exists here, too, which is that the virus checks only the first two bytes of the 'PE' signature. Thus, it would be possible to create a DOS file which happened to have the 'PE' characters in the right location, followed by something other than zeroes, and the virus would try to infect it. One of the other things that the virus checks is that the size of the 'MZ' header is 64 bytes. This was an old recommendation from *Microsoft* for quickly identifying potential *Windows* files. At the time, it applied to the 'New Executable' file format, as part of *Windows 3.0*, but it would be equally applicable to the current Portable Executable format. However, *Windows* itself has never checked the field.

The other things the virus checks for are that the file contains at least three sections, has non-zero values for the SizeOfOptionalHeader, SizeOfCode and BaseOfCode fields, and that the file targets the GUI subsystem (as opposed to being a console-mode or driver file). The virus does not exclude DLLs – presumably assuming that no DLL will have one of the suffixes of interest. As with the MZ header size, *Windows* has never checked either the SizeOfCode or the BaseOfCode field values, and it is possible to create a file whose SizeOfOptionalHeader is zero. Of course, such a file will not be infected by this virus.

INFECTIOUS GROOVES

If a file is found to be infectable, then the virus adjusts the file size to include the size of the virus, then rounds up the result to a multiple of the SectionAlignment value from the PE header. It then rounds this number up to a multiple of 117. The resulting value is used as the size in memory for the temporary copy of the host. Unfortunately, this value might be insufficient (see below), which will result in file corruption.

The virus requires that the file to infect has an import table. The virus attempts to find the section that contains the import table, and then walks the table, looking for a DLL whose name begins with 'kernel32'. However, the virus uses a faulty method to determine the location of the section table. The problem is that the virus relies on the value in the NumberOfRvaAndSizes field to determine the size of the optional header, instead of using the value in the SizeOfOptionalHeader field (see also *VB*, February 2009, p.7). As a result, it is possible to create a file with two section tables: one that this virus sees, and one that *Windows* sees.

This bug is repeated when the virus attempts to find the section with the largest file offset (in fact, the bug appears in the code a total of five times). This is used to determine where the data ends in the file, in order to check for overlay data. However, there is another bug in this code, which is that the physical size for the section is not checked. If the physical size for a section is zero, then the file offset can be set to any value, and that would cause problems for the virus.

The virus determines that a file has overlays if the amount of data is at least twice the size of the SectionAlignment field value. As a result, the virus misses the presence of small overlays, such as debug data. Such data will be destroyed when the virus infects the file.

If a file is found not to be infectable at this point, then the virus rounds up the original file size to a multiple of 113. This allows the virus to skip files that have been examined already, thus improving the efficiency of any future searches.

A NEW EPOCH

The virus uses an entrypoint-obscuring (EPO) technique. The EPO routine begins by attempting to find the section that contains the entrypoint. Within that section, the virus searches for FF15- and E8-style calls. This kind of EPO is similar in style to the W95/MTX virus from 2000. If an FF15-style call is seen, then the virus checks whether the address that follows points into the import table. Specifically, the virus checks whether the import table entry corresponds to an import from kernel32.dll. If the import comes from kernel32.dll, then the original call will be considered a candidate for replacement.

If an E8-style call is seen, then the virus checks if the destination remains within the current section. If it does, then the virus checks if it points to an FF25-style jump. If it does, then the original call will be considered a candidate for replacement.

For either call style, there is a 50% chance that the virus will replace the call immediately. However, if the search reaches the end of the section without making any change, and if a candidate has been located, then the virus will replace that candidate without exception. The replacement uses an E8-style call to point to the virus code.

After deciding on the entrypoint, the virus generates a new polymorphic decryptor. The engine was written by another virus writer in 1999, and used in the W32/Aldebaran virus. It is quite a simple engine. It uses very few instructions but it contains some characteristics that are always present, which make it easy to identify. One potential problem with the engine is that it has no concept of maximum size. Thus, the decryptor may be so large that an exception occurs while appending the virus body. In fact, the decryptor may be so large that an exception occurs while producing the decryptor itself!

THIS SECTION RESERVED

If the polymorphic decryptor is generated successfully, then the virus appends its body to the decryptor, and places the whole thing at the end of the section whose data appeared last in the file, adjusting the virtual size appropriately. There is a significant problem with this approach. If that section was not the last in the file, then the new virtual size might result in that section overlapping the next one. Such a file cannot be loaded in *Windows NT* and later.

The virus marks the section as readable, writable and executable. This allows the virus to run in environments in which Data Execution Protection (DEP) is enabled. Then the virus does a most peculiar thing. It scans the data directories for a reference to the section that holds the virus body. If a reference is found, then the virus sets the size of the entry to the size of the section. This can cause some peculiar behaviour, particularly regarding the export table. In the event that an export address table entry originally pointed within the same section, but outside of the export table (and was therefore truly exported by the file), the entry will now appear to point into the export table and will therefore appear to be forwarded to another DLL, whose name will look ... quite foreign.

After infecting the file, the virus will check if it had a checksum. If it did, then the virus will recalculate it. The virus carries its own routine for this calculation, which combines code that is taken from imagehlp.dll along with some 16-bit code to perform a further adjustment to account for the existing checksum. This suggests that the virus author did not understand the algorithm at all.

Once all of the files have been infected in the current directory, the virus performs the same actions within the *Windows* directory and the system directory, before moving on to an entirely new target.

LINK IN THE CHAIN

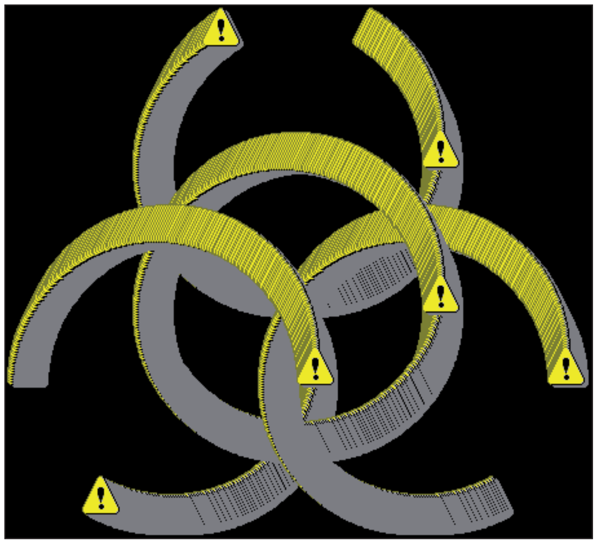
The virus searches within the current directory for files whose suffix is ‘LNK’. For each such file that is found, the virus checks that the file is in LNK format, and that it contains a shell item ID list and points to a file system object. If that is the case, then the virus skips the shell item ID list and examines the file system object entry. The virus ignores the file attributes field, which could be used to determine if the object is a file or a directory, and simply assumes that the object is a file. This is essentially harmless, though, because even if a directory had one of the suffixes of interest, the virus would not be able to open it as a file. However, if the link points to a file whose suffix is of interest, and if all other tests pass as described above, then the virus will infect the file as usual. After infecting the link files in the current directory, the virus searches for link files in the %desktop% directory.

To minimize its memory consumption, the virus attempts to free any DLL that it has been using, even if the virus did not load the DLL itself. This might appear to be a bug, but actually it isn’t one, because statically loaded DLLs cannot be unloaded. Thus, the attempt to free the DLL will be ignored by *Windows*, when appropriate.

BAITING THE HOOK

Once the infection routine has completed, the virus walks the host import table, looking for a DLL whose name begins with ‘kernel32’. Then the virus searches for imports of any of the following functions: CreateFileA(), MoveFileA(), CopyFileA(), CreateProcessA(), SetFileAttributesA(), GetFileAttributesA(), SearchPathA(). Perhaps coincidently, this list is very similar to that of the W32/Cabanas virus from more than a decade ago. The virus hooks as many functions as are imported from that list. Interestingly, the virus uses the WriteProcessMemory() API to install the hooks, even though the memory is addressable directly. This does not bypass any memory protection that might be present. As a result, since the virus does not call VirtualProtect() first, and if the import table is in a read-only memory region, then no hook will be installed. However, the use of the API does avoid the need for an exception handler. In the event that the import table is in a read-only memory region, then any attempt to write directly to the memory would cause an exception, but the WriteProcessMemory() API will simply fail the write.

Each of the hook routines calls a single common routine, then unhooks itself, before calling the original API. The common routine retrieves the directory from the API’s parameter list, changes to there, and searches within that directory for files to infect. The fact that the original API is not called until after the search has completed means that the process could appear to be unresponsive and obviously infected. Of course, the virus could use a thread



W32/ Satevis payload.

to perform the scan instead, but that introduces a different problem for the virus. The problem in that case would be that any thread that called the ExitProcess() API would cause all other threads to be terminated, essentially no matter what they were doing. While there are ways to deal with that, and some of them have been demonstrated by other viruses, the solutions are complex, and this virus is simple.

PAYLOAD

The virus has a graphical payload, which activates if an infected file is executed on the 31st day of any month. The payload is to draw a biohazard symbol in the centre of the screen, covering half of the screen in both dimensions.

The final step in the virus code is to allow the host to continue executing. The virus replaces the start of its code with some redirection code that points to the original API. Thus, the virus cannot be reached a second time, no matter how many times the hooked call is executed. For the FF15-style call, the replacement code begins with an ‘inc [esp]’ instruction to skip an additional byte, since the E8-style call is one byte shorter than the FF15-style call that it replaced. Then the virus stores an FF25-style jump to the original address, before jumping directly to that location.

CONCLUSION

There should be a term for a virus that is nothing but a collection of old routines. This is like the viral version of a mix tape. It’s so very retro, I feel like disco-dancing now.