MALWARE ANALYSIS

PRESCRIPTION MEDICINE

Peter Ferrie
Microsoft, USA

People often ask how we choose the names for viruses. It might seem as if it's in the same way as pharmaceutical companies choose their product names. Zekneol – chemical or virus? In this case, it's a *Windows* virus: W32/Zekneol.

EXCEPTIONAL BEHAVIOUR

After decryption, the virus begins by discarding a number of bytes from the stack. The number of bytes to be discarded is specified in a variable in the virus body. However, the value in this variable is always zero because the polymorphic engine in the virus does not support the generation of fake push instructions.

After 'emptying' the stack, the virus retrieves the return address from it, which points into kernel32.dll. The virus intends to use this as a starting point for a search for the PE header of kernel32.dll. As a precaution, the virus registers a Structured Exception Handler (SEH), which is supposed to intercept any exception that occurs. The virus will search up to 256 pages for the PE header. If the header is not found, then the virus enters an infinite loop. This loop is intentional, it's not a bug. However, if an exception occurs during the search, the handler is reached, along with the first two bugs in the code. After restoring the stack pointer, we see a write to the ExceptionList field in the Thread Environment Block (TEB). Presumably the virus author wanted to unhook the handler, but he forgot to initialize the pointer register first. Thus, the code attempts to write to an essentially 'random' address. This causes a secondary exception, which destroys the handler pointer that is on the stack. What happens next depends on the platform.

On *Windows 2000* and earlier, the damaged handler pointer is assumed to be valid, and so it is used. This of course causes another exception to occur, and the damaged handler pointer is used again, causing yet another exception, and ultimately resulting in an infinite loop. On *Windows XP SP2* and later, the handler pointer is recognized as being invalid, and the application is terminated.

That's the first bug. The second bug occurs on the same instruction. Even if the pointer register were initialized, the wrong value would be written. When registering or unregistering a handler via SEH, the value to write to the ExceptionList field in the TEB is a pointer to a structure. The structure contains a pointer to the handler. The problem is that the virus tries to store the pointer to the handler itself. The reason this happens is that, despite the two values being next to each other on the stack, the virus picked the wrong one.

In fact, there is a third bug in the same region of code. Even if the write succeeds (if the virus initializes the register and chooses the correct pointer to use), the virus attempts to continue the search. The problem is that the search uses several other registers, all of which have been modified as a result of the exception, and none of which are now initialized.

THE PURSUIT OF H-API-NESS

If all goes well, and the virus finds the PE header for kernel32.dll, then the virus resolves some APIs including two which are never used (GetCurrentDirectoryA() and GetWindowsDirectoryA()). The virus uses hashes instead of names, but the hashes are sorted according to the alphabetical order of the string that they represent. This means that the export table needs to be parsed only once for all of the APIs, instead of once for each API, as is common in some other viruses.

After retrieving the API addresses, the virus registers another Structured Exception Handler. The same two bugs exist here regarding the handler behaviour of an uninitialized register and writing the wrong value. The virus uses the same hashing method to resolve an API from user32.dll and several from advapi32.dll (including CryptDecrypt(), which is never used). However, the virus uses the GetProcAddress() API to retrieve the address of the ChecksumMappedFile() API from imagehlp.dll and the SfcIsFileProtected() API from sfc.dll, if those DLLs are available. 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 IsFileProtected() API from sfc.dll, since it is forwarded to sfc_os.dll in *Windows XP* and later.

MISDEEDS AND MISDIRECTION

The virus selects a random number from one to five, which it uses as the number of 'diversion' API calls to make. Then the virus counts the number of 'safe' APIs that it found in the host (this will be described in detail below). The table contains a number of structures, each of which is two DWORDs large. The first DWORD is the RVA of the 'safe' API, and the second one is the number of parameters that the API accepts. However, there is a bug in the counting method. Instead of examining every second DWORD, the virus examines every DWORD for a value of zero. Thus, if an API accepts no parameters, then the parameter count slot will be considered the end of the list. The result is a count that is both incorrect and invalid, since the end of the list is now misaligned. This bug is essentially harmless, though. The virus chooses randomly from among the APIs, using the wrong number of entries, as calculated previously. However, a more serious bug does exist. The virus multiplies by eight the index of the chosen API. The assumption is that the original count was simply the number of APIs, and therefore multiplying by eight would be the correct behaviour. However, since the count is already too large, and if the table is very full, then as a result of the multiplication the access will be beyond the end of the table. If, for example, it should hit one of the variables that exists after the table, then that variable will be considered the number of parameters to place on the stack. This number might be very large and cause a stack-overflow exception, and a possible hang as described above. Even if the parameter count appeared to be zero, the assumed API itself is still called, which might cause some unexpected behaviour.

If the corresponding slot in the table is empty, then no attempt is made to call the API. If an API does exist, and if that API accepts parameters, then the virus places onto the stack a corresponding number of random values before calling the API. An API is considered 'safe' if it will return an error when passed invalid parameters.

The buggy selection is repeated according to the number of 'diversion' API calls to make, which was chosen previously. The virus then encrypts its memory image, with the exception of a small window, prior to calling the true API. Upon return from the true API, the virus decrypts its memory image, and then performs another set of 'diversion' API calls, as described above. The encryption key for the memory image is changed each time this routine is called. The intention of the routine is to defeat memory scanners that perform their scanning whenever certain APIs are called.

The whole routine, beginning with the first set of 'diversion' API calls, is called whenever the virus wishes to call an API (with two exceptions: CreateThread and GetTickCount are called directly – this is probably an oversight, since nearby APIs within the same routine are called indirectly).

KAMIKAZE CODE

The virus searches within the current directory for '.exe' files whose name begins with 'kaze'. This appears to be a bug, since the first generation version of the virus uses one string, but replicants of the virus use another. However, the string has been duplicated, so the result is always the same.

For each such file, the virus begins by checking if the SfcIsFileProtected API exists. If the API exists, then the virus retrieves the full path of the file, converts the pathname from ASCII to Unicode, and then checks if the file is protected. This is the correct method to determine the protection state. Most viruses that perform the check forget that the API requires the full path to the file. However, if the file is protected, the virus attempts to unmap a view of the file and close some handles. The problem is that the file has not yet been either opened or mapped. Fortunately, the attempt simply returns an error, unless a debugger is present. If a debugger is present, then closing the handle will cause an exception, and a possible hang as described above.

If the file is not protected, then the virus attempts to open it. If the attempt fails, then the virus skips the file, without attempting to unmap or close it. If the open succeeds, the virus maps a view of the file.

The virus contains only one bounds check when parsing the file. That check is simply that the PE header starts within the file. There is no check that the header ends within the file, and the existence of an Import Table is assumed. This means that certain valid but unusual files will cause an exception and a possible hang, as described above. The virus is interested in PE files that are not already infected, and which contain an Import Table that is less than 4,066 bytes large. The virus does not care if the file is really a DLL or a native executable. The infection marker is that the second byte in the time/date stamp in the PE header has a value of 0x36.

The virus places the infection marker immediately, and resizes the file enough to hold the virus code. The virus does not care about any data that has been appended to the file outside of the image. Any such data will be destroyed when the file is resized.

The virus searches for the section with the largest virtual address. For files that run on *Windows NT* and later, this will always be the last section. However, *Windows 9x*/*Me* files do not have such a requirement. If the virtual size of that section is larger than the physical size, then the virus will not infect the file. However, the infection marker and increased file size remain.

RELOCATION REQUIRED

With a 20% chance, and if the file contains relocations, the virus will relocate the image. The virus parses the relocation table, and applies the relocations to the image using a new image base of 0x10000. After the relocation has been completed, the relocation table is no longer required. There is a bug in the parsing, which is that the virus assumes that the relocation table ends when the page RVA is zero. The assumption is incorrect. The size field in the data directory contains the true size. Further, the virus assumes that any non-zero value is valid, but if the virus is reading data from beyond the end of the relocation table, then it might cause an exception and a possible hang, as described above.

When parsing the relocation data, the virus supports only two types of relocation item. They are the IMAGE_REL_BASED_ABSOLUTE and IMAGE_REL_BASED_HIGHLOW. There are several other documented relocation types, and if one of them is seen, then the virus will hit a breakpoint and possibly hang as described above. However, it is rare for files to use relocation types other than the supported two.

After relocating the image, the virus chooses a new image base randomly. The new image base always points into the upper 2Gb of memory, and is 64kb-aligned. The alignment is a requirement for *Windows NT* and later. It is interesting that the virus appears to have been written to support older versions of *Windows*, since it considers the presence of both imagehlp.dll and sfc.dll to be optional. In fact, imagehlp.dll was introduced in *Windows 98*, and sfc.dll was introduced in *Windows 2000*, so the support goes back a long way. However, the use of 0x10000 as the relocated image base ties the virus to *Windows 2000* and later. The reason for this is that *Windows NT* does not relocate .exe files, and *Windows 9x* and *Me* use 0x400000 as the default image base for relocated files.

DEP-RECATED CODE

The virus increases the size of the last section by 139,264 bytes, and changes the section attributes to read/write/initialized. Unfortunately for the virus author, the executable attribute is not set explicitly. As a result, if the attribute was not already set in the original file, then the virus will fail to execute on systems which have Data Execution Protection enabled.

The virus saves information about the address and size of resources and imports. The virus pays special attention to the imports, parsing the table and saving pointers and ranges. However, as before, there is no bounds checking while saving the values, so a very large Import Table could cause corruption of other entries in the list.

The virus will now choose a decryptor method to use. The virus uses a crypto-based method 80% of the time. For the other 20% of the time, it uses a simple 32-bit add-based decryptor.

CRYPTONITE

If the crypto-based decryptor is chosen, the virus copies the host's Import Table to the original end of the last section and updates its RVA in the data directory. The size of the Import Table is increased by the size of one Import Table record, and the Bound Import Table data directory entry is erased. The virus appends to the Import Table an entry that refers to the advapi32.dll file. The 'advapi32.dll' string is appended to the section, at a random location beyond the end of the Import Table. The five crypto-related APIs that the virus uses (CryptAcquireContextA, CryptCreateHash, CryptHashData, CryptDeriveKey and CryptDecrypt) are appended to the Import Table, interspersed with one to four imports chosen randomly from a set of 75 'safe' APIs from the advapi32.dll file. The name of each API is placed at a random location beyond the end of the Import Table. The virus also contains code to replace the unused bytes with random data, but this routine is never called.

SAFETY IN NUMBERS

The virus examines the host Import Table for references to DLLs that it knows contain 'safe' APIs. Those DLLs are kernel32.dll, ws2_32.dll, user32.dll and gdi32.dll. If one of the 'safe' DLLs is imported, then the virus searches for references to the 'safe' APIs. If any 'safe' API is imported, then the virus adds the reference to a table within the virus body. There is what might be considered a bug in the search routine. The virus searches the entire Import Table for a reference to the first 'safe' DLL, then searches for references to the 'safe' APIs of that DLL, then searches for a reference to the second 'safe' DLL, and so on. However, if the host does not import anything from one of the 'safe' DLLs, then the virus stops searching completely. None of the following DLLs will be checked, and no more 'safe' APIs will be added to the table. Thus, in the extreme case, if the host does not import anything from kernel32.dll, then no 'safe' APIs will be added at all.

The virus then copies the decryptor, and optionally inserts calls to the 'safe' API if the crypto-based method was chosen. As before, the method to choose from the 'safe' API table uses a count that is too large, resulting in empty slots being seen, and thus no API call being inserted in that instance. However, there are multiple places within the decryptor where APIs can be inserted, which increases the chance that at least one of them will succeed.

BAIT AND SWITCH

If the crypto-based method was chosen, the virus changes the attributes of each section to read/write/initialized, until the section containing the entrypoint is seen. However, the virus chooses random locations only from within the section that contains the entrypoint. The virus saves the contents from each of the locations that were chosen, since they will be replaced later by parts of the decryptor.

The virus then constructs a new decryptor. The decryptor is described using a p-code language, which gives it great flexibility. The p-code contains only 57 instructions, but they are quite capable of producing a seemingly wide variety of code. However, the characteristics of that code are instantly recognizable, and the instruction set used is very small. Some of the instructions are also called recursively, so that, for example, a simple register assignment first becomes a series of assignments and adjustments through other registers. The two types of decryptor together use fewer than half of the possible instructions, but internally those instructions use all but one of the remaining instructions (the missing one is a 'test' instruction involving a memory address and a constant). While interpreting the p-code, the virus resolves the API calls, both real and fake, and inserts random numbers for the parameters to the fake APIs, and real parameters for the real APIs.

If the virus has relocated the image, then it will also encrypt some of the blocks by using relocation items (for a description of the process, see *VB*, April 2001, p.8). The virus creates a new relocation table that contains only the items for the decryptor, by overwriting the original relocation table in the host. However, in contrast to ordinary files, the virus places the relocation items in decreasing order in the file, and calculates some page addresses using values that are not page-aligned. These two characteristics immediately make those files suspicious. The virus stops applying relocations when fewer than 328 bytes of space remain in the original table. There is a bug here, though, which is that if the original table was less than 328 bytes long, then the virus sets the table size to zero bytes. The resulting file will no longer load, because when an image must be relocated, *Windows* requires that a relocation table contains at least the page address and the number of relocation items (even if the number of items is zero).

ROCK CITY FUNK

At this point, the virus copies itself to the file in unencrypted form. The encryption is performed next, on the copy of the virus body, using the chosen method. The crypto-based method uses a 128-bit RC4 cipher, with an MD5 hash as the key. The key is derived from the four-byte Import Lookup Table RVA in the first entry of the new Import Table.

The virus increases the size of the image by 139,264 bytes, and if the ChecksumMappedFile API is available, then the virus uses it to calculate a checksum for the file. This results in a file having a checksum that might not have existed before. Finally, the file is unmapped and closed. The virus then searches for the next file to infect.

Once all files have been examined, the virus displays the message 'Infecté', if not running a first generation of the code. If the executing image is not a first generation of the code, then the virus changes the section attributes to read/write/executable for the section that contains each block. Of course, it's too late to save the virus on DEP-enabled systems. Since all of the blocks are chosen from the same section that contains the entrypoint, changing the attributes multiple times is ultimately pointless. It appears that the virus author wanted to support blocks in multiple sections, but this virus does not support it. After changing the attributes of the blocks, the virus restores their contents.

GOSSAMER THREADS

After restoring the host to an executable state, the virus creates a thread to search drives for other files, then run the host. The thread registers another Structured Exception Handler. However, this time only the second bug is present. The virus initializes the pointer correctly, but the value to write is still wrong. Further, if an exception occurs, then the virus wants to exit the thread, but the problem is that the code uses another register which has been modified as a result of the exception, and is now not initialized.

If no exception occurred, then the virus begins with drive 'B:' and proceeds though drive letters until the first drive is found which is either fixed or removable. Only that drive will be examined. This might also be considered a bug, but the loop contains no other exit condition, so perhaps it was intentional to stop after one drive. The idea of starting with drive 'B:' rather than 'A:' could also introduce a bug, in the (admittedly rather unlikely) event that the only drive on the system is 'A:'. In that case, all possible values would be tested, but even so, eventually the value would wrap around and the 'A:' drive would be found. When an appropriate drive is found, the virus sleeps for one second before beginning the search for files. The search is for 'kaze' files, as described above. Upon completing the search for files, the virus will search for directories. If a directory is found, then the virus will enter the directory and begin the search again, after sleeping for one second, as before. If no other directories are found, then the virus will step out of the current directory and resume the search. After all directories have been examined, the thread will exit.

In the event that the host process finishes before the virus thread exits, the virus thread will be forcibly terminated. This could result in a corrupted file if the virus was in the act of infecting it at the time.

CONCLUSION

Zekneol certainly appears to be a complicated virus, but looks can be deceiving. The crypto-based decryptor has so many tell-tale signs that detection is straightforward; the simple decryptor is really very simple; and the new relocation table looks like no other.

As for how we choose the names for viruses, that's a question for another day.