# MALWARE ANALYSIS 1

## A SHORT VISIT WITH A VIRUS

*Peter Ferrie*
Microsoft, USA

Last month, I described a *Windows* virus that turns Java class files into droppers for the virus [1]. I concluded that it would be a simple matter for a virus writer to reverse that – in other words, to have a Java class file that turns *Windows* files into droppers for the virus. That is exactly what we have in {W32/Java}/Tarry.

### SECOND PLACE GOES TO...

The virus begins by pushing the host original entry point onto the stack. It then adds the host ImageBase value from the Process Environment Block, to construct the virtual address of the host entry point. This allows the virus to support applications that opt into Address Space Layout Randomization (ASLR), even though it does not support files that support ASLR.

The virus registers a Structured Exception Handler, then retrieves 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 virus assumes that kernel32.dll is the second entry in the list. This is true for *Windows XP* and later, but it is not guaranteed under *Windows 2000* and earlier because, as the name suggests, the list shows the order of *loaded* modules. If kernel32.dll is not the first DLL that is loaded explicitly, then it won't be the second entry in that list (ntdll.dll is guaranteed to be the first entry in all cases).

### IMPORT/EXPORT BUSINESS

The virus resolves the address of the only API function that it requires: LoadLibraryA. Despite this being only a single entry, the virus uses a hash instead of a name. It uses a reverse polynomial to calculate the hash. The virus does not check that the export exists, relying instead on the Structured Exception Handler to deal with any problems that occur. Of course, the required API should always be present in the kernel, so no errors should occur anyway.

The hash table is terminated explicitly using a single byte. The position of this byte corresponds to the next hash value in the list, and the search exits when a particular value is seen. This is intended to save three bytes of data, but actually introduces a risk. The assumption is that no hash will have that value in that position. While this is true in the case of this virus, it could result in unexpected behaviour if other APIs are added, for which the low byte happens to match the current sentinel value. The reason the hash technique is used to resolve a single API address is simply because this virus is derived from an existing code base that makes use of the technique for resolving multiple API addresses. It would be quicker simply to resolve GetProcAddress() by string name, and then use that API to resolve LoadLibraryA().

### JAVA VIRTUAL MISMATCH

The virus loads the jvm.dll with no path. This assumes that the DLL can be found in one of the locations in which *Windows* searches, but the host can affect this list by using DLL redirection or carrying a manifest. The virus resolves a single API from this DLL, again by using the hash method. The API is JNI_CreateJavaVM(). The virus uses this API to create a new instance of the Java VM, but in order for the API to succeed, the '_ALT_JAVA_HOME_DIR' environment variable must be defined and must point to the Java installation directory. This is not normally the case. The usual method for invoking Java is by adding the Java installation directory to the path environment variable, and then either passing the directory on the command line or allowing the Java executable to define the environment variable dynamically.

The virus creates a byte array that is large enough to hold the combined virus body, then defines a class that contains the Java-specific virus body. It retrieves a pointer to the infection method within the Java-specific virus body, then runs the method. This technique allows the virus to execute Java code directly from memory, instead of dropping a class file and executing it from disk. After the method call returns, the virus deletes the objects in memory, then raises an exception using the 'int 3' technique. The 'int 3' technique appears only once in the virus code, but is an elegant way of reducing the code size. 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.

The exception handler restores the registers, then transfers control to the host entry point.

### CAFEBABE

The Java-specific virus body begins by creating a list of the objects in the current directory. The virus enumerates the entries in the list, looking specifically for files. It uses no bounds checking during the enumeration, relying instead on a defined exception handler to receive control when the list is exhausted. The virus attempts to open any file it finds, in writable mode. It does not check whether the file is writable, nor does it remove the read-only attribute first.

The virus is interested in 32-bit Portable Executable files for the *Intel* x86 platform that have no relocation items, line numbers, symbols, or appended data, and that are not system or DLL files. The virus checks the COFF magic number to ensure that the file is 32-bit. It requires the file to be targeting the GUI subsystem and to have no flags set in the DllCharacteristics field. This reduces the pool of candidates to those which are primarily not No-eXecute compatible, and which do not support Address Space Layout Randomization. This effectively rules out all modern applications. The virus also requires that the file has no Load Configuration Table structure, which further rules out any file that uses SafeSEH, among other things.

The virus has some strange code in a couple of places, whereby a query is made for the current file pointer position, despite a seek() operation being issued immediately before it. In that case, the position would be known already. The virus also searches to the end of the file by adding the SizeOfRawData and PointerToRawData values from the last section, instead of simply using the file.length property.

## [NO] TEST, [NO] FIX

The technique that the virus uses to infect a file is to write a push instruction to the end of the file, followed by what should be the original AddressOfEntryPoint value, and then the array that contains the combined virus body. However, there is a bug in the code that performs writes to the file. The bug is that the third byte of each four-byte write references the wrong variable. Instead of accessing the third byte of the four-byte value, it accesses the first byte of the SizeOfRawData value from the last section. Such a bug would have been obvious immediately had any attempt at recursive infection been made. It seems likely that the first generation of the virus code was executed, it infected a file, and the result was examined statically.

After writing the combined virus body, the size of the file is increased by 4KB. This value is hard-coded, and is entirely independent of the size of the virus code. The virus introduces the infection marker for files whose file alignment is less than 4KB. However, since the increase in size does not take into account the actual file alignment value, a file whose file alignment value exceeds 4KB will be reinfectable because it lacks the infection marker. The virus increases the virtual and physical size of the last section by exactly 4KB each. In this case, since the increase in size does not take into account the actual file alignment value, a file whose file alignment value exceeds 4KB will appear to be truncated and will fail to load. Finally, the virus marks the section characteristics as writable and executable.

## BEST INTENTIONS

The virus intends to set the entry point to point to the original end of the last section, but the file-write bug applies here, resulting in a corruption of the value. This means that unless the host had an entry point whose third byte happened to match the value of the first byte of the size of the last section (most commonly, a value of zero), then the infected file will not execute any code.

The virus also intends to increase the size of the image by 4KB, but depending on the original value of the size, the file-writing bug is likely to corrupt the value to the point where the new image size is far too small to cover all of the sections, and thus the infected file will not even load. Specifically, if the size of the image (not the file) is any multiple of 61,440 bytes up to about 15MB, which is the case for calc.exe in *Windows XP* (126,976 bytes), then the new size of the image will be zero. Other values will simply be truncated, but with the same effect.

Once the file has been infected, the virus closes the handle, and then continues to search for more files.

## CONCLUSION

There is nothing inherently difficult about creating a Java virus that infects *Windows* executable files. The fact that the Java-specific portion of this virus code was written directly in byte code rather than the high-level code is merely a detail. However, the fact that something as simple as that could yield such a significant bug should give pause to the virus writer. This is clearly not where he/she should be spending his/her time. There are many other arts which are far more forgiving of mistakes in implementation. Why not go and paint something instead?

## REFERENCE

[1] Ferrie, P. Getting one's hands dirty. Virus Bulletin, February 2014, p.4. http://www.virusbtn.com/pdf/magazine/2014/201402.pdf.