MALWARE ANALYSIS 1

GETTING ONE'S HANDS DIRTY

Peter Ferrie
Microsoft, USA

Cross-platform execution is one of the promises of Java. Cross-platform *infection* is probably not what the designers had in mind. However, it was clearly in the mind of the author of W32/Java.Grimy, a virus for the *Windows* platform, which infects Java class files.

SECOND PLACE GOES TO...

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 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* or earlier because, as the name implies, it is 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 addresses of the API functions that it requires. The list is very small, since the virus is very simple: set attributes, find first/next, alloc/free, open, seek, read, write, close, exit. The virus uses hashes instead of names, with the hashes sorted alphabetically according to the strings that they represent. The virus uses a reverse polynomial to calculate the hash. Since the hashes are sorted alphabetically, the export table needs to be parsed only once for all of the APIs. Each API address is placed on the stack for easy access, but because stacks move downwards in memory, the addresses end up in reverse order in memory.

The virus does not check that the exports exist, relying instead on the fact that if an exception occurs then the virus code will be terminated silently. This is acceptable because the virus file is a standalone component so there is no host code to run afterwards. Of course, the required APIs should always be present in the kernel, so no errors should occur anyway.

The hash table is not terminated explicitly. Instead, the virus checks the low byte of each hash that has been calculated, and exits when a particular value is seen. This is intended to save three bytes of data, but introduces a risk. The assumption is that each hash is unique and thus when a particular value (which corresponds to the last entry in the list) is seen, the list has ended. While this is true in the case of this virus, it might result in unexpected behaviour if other APIs are added, for which the low byte happens to match another entry in the list.

Once the virus has finished resolving the API addresses, it searches the current directory (only) for all objects. Unlike most other viruses written by this virus author, this one uses Unicode APIs for the 'find' and 'open' operations. This allows the virus to examine files that cannot be opened using ANSI APIs. The virus is really only interested in files, but it examines everything that it finds. For each object that is found, the virus will attempt to remove the read-only attribute, open it, and allocate a memory block equal to the size of the virus plus twice the size of the file. For directories, the open will fail and the file size will be zero. The virus intends to read the entire file into memory. It is not known why the author did not use a buffer of just the size of the virus plus the size of the file, and read the file into the buffer at the offset equivalent to the size of the virus. As it is, the virus is at risk of a heap overflow vulnerability for files of around 2GB in size, since the file is read entirely before it is validated – these days files of 2GB or more are not uncommon.

COFFEE, COFFEE, COFFEE

After reading the file into memory, the virus registers a Structured Exception Handler, and then checks for the Java signature (0xCAFEBABE) and the class version. The virus excludes files that are not Java class files, as well as any that are built with Java 6 or later. This seems to be a severe restriction, given that Java 6 was released in 2006 – the virus is left to target extremely old versions of Java.

When an acceptable file is found, the virus retrieves the count of entries in the constant pool table, and exits if there are not enough free entries left for the virus to insert its own. The virus parses the entries in the constant pool table, and watches for UTF-8 format strings that contain the text 'hh86' or 'Code'. The 'hh86' string is used as an infection marker, so the virus exits if this string is seen, regardless of the context in which the reference appears. This means that any reference to the infection marker string (via, for example, 'String foo=' or 'System.out.println()') will cause the file to appear to be infected. The 'limitation' is acceptable to the virus. In the case of the 'Code' string, this check is meaningful only during the infection phase.

While parsing the file, the virus also checks for three tag types that were only added to Java 7 in April 2013: MethodHandle, MethodType and InvokeDynamic. It is not known why the virus checks for these tags, since they cannot appear in class files built with Java 5 or earlier.

METHOD ACTING

The virus knows how to skip the interface and field tables in order to reach the methods table. For each of the methods

in the table, the virus retrieves the number of attributes. For each of the attributes for a method, the virus retrieves the name index, and then searches the constant pool for the 'Code' string with a matching index. If a match is found, the virus retrieves the size of code attribute, and skips the method if not enough free space is left for the virus to insert its own code. If the method is small enough, the virus checks whether it makes use of exceptions (the result of a 'try/catch' sequence in the source code). The virus is interested only in the first method that implements exceptions.

When a suitable method is found, the virus duplicates the contents of the file in memory, up to the point where the constant pool ends. The virus increases the number of entries in the constant pool by 31, and then appends the new entries to it. It updates the class index for each of the virus-specific entries in the constant pool by adding the index of the last host constant pool entry to each of them. Next, it appends the host data from the end of the host constant pool until the start of the methods table, to the new copy of the file in memory. The virus prepends its own method to the methods table, and updates the two method indexes by adding the index of the last host constant pool entry to each of them.

The virus carries a compressed MZ/PE header combination, which will be used for the standalone virus file which holds the replication code. The headers are very sparse – they contain almost the minimum number of non-zero bytes that must be set in order for the file to be acceptable. Specifically, the headers contain the minimum number of non-zero bytes for a file that contains a section. For a file that contains no sections, several more bytes could be removed. The dropped file has one section with no name, to reduce the number of bytes that have to be written during the decompression phase.

The section has only the writable and executable flags set. This is an interesting choice, since it does not affect the number of bytes to decompress but it does introduce the (infinitely small) risk that a future version of *Windows* will enforce the flag exactly as specified, and thus break the virus. Currently, the setting of the executable flag results in the readable flag being set, even if that is not explicitly the case. The reason for this is to support the mixing of code and read-only data in the same segment, for example in ROM code. However, the CPU does have the ability to mark a segment as only executable, which would result in read-access failures in the case of the virus.

The virus declares a 2KB array and decompresses the header into the array, using an offset/value algorithm. The implementation supports writing only to the first 256 bytes of a buffer, but this is sufficient to describe the PE file that the virus uses. This compression format is probably optimal for the purpose – while a Run-Length Encoding format could compress the data further, that gain is more than lost by the size of the decompression code. The result is a series of assignments to offsets within the array. The virus does the same thing for each byte of the virus body. While this technique works well enough, it results in a large amount of repetitive code. It is not known why the author chose the array method instead of, for example, a textual encoding method which would have reduced the code size enormously.

GOING ON A FIELD TRIP

The virus appends the remainder of its method code, and updates the constant pool references by adding the index of the last host constant pool entry to each of them. Next, it appends the host data from the start of the methods table until the start of the method that makes use of exceptions, which it identified earlier. The virus updates the attribute and code length fields in the method information structure, before copying the rest of the method information to the new copy of the file in memory. The virus appends its own exception handler code to the host method, and then alters the first entry in the exception table to point to the virus exception handler. The virus exception handler invokes the virus method that the virus added, and then transfers control to the original host exception handler. Thus, if an exception occurs during the execution of the block defined by the first exception handler, then the virus exception handler will gain control. If no exception occurs within that block, then the virus will never execute. Finally, the virus appends the remaining content from the host file to the new copy of the file in memory. Once the copy is complete, the virus replaces the file on disk with the copy in memory, and then raises an exception using the 'int 3' technique. The 'int 3' 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.

The exception handler frees the allocated memory, closes the file, and then continues the search for more objects. After all objects have been examined, the virus simply exits.

CONCLUSION

This virus demonstrates the simplicity of creating a *Windows* file that turns Java class files into droppers. What's next? It would be equally simple to reverse that – to have a Java class file that turns *Windows* files into droppers for the virus. From there, it would only be slightly more work to combine the two into a circular infection process. Cross-platform infection is a promise that we'd be happy to see broken.