MALWARE ANALYSIS 3

TUSSLING WITH TUSSIE

Peter Ferrie
Microsoft, USA

When we think of decoding, we think of a block of encoded data, and a decoder. There are multiple ways to hide the decoder, such as by forcing *Windows* to apply a relocation delta [1], or by using obscure instruction side effects [2]. Now, W32/Tussie shows us a way to hide the encoded data.

CALLING ALL CARS

The virus begins by caching the address of the Process Environment Block. There is no good reason for this (and in fact it can result in unexpected behaviour, see below), because by simply swapping the caching register in three places, the register that originally held the value would not be altered. The virus retrieves the value from the ImageBaseAddress field in the Process Environment Block, and applies the appropriate relative offset to point to a writable buffer. This buffer receives the decoded code. The virus registers a Structured Exception Handler, and then begins the decoding process.

The way that the data is encoded is simple but interesting, because the data is hidden in executable instructions. A series of 'call' instructions are made into an array of 256 'int 3' instructions. When each 'call' instruction is executed, the return address is saved on the stack. When the 'int 3' instruction is reached, an exception occurs. The exception handler in the virus code intercepts the exception and checks whether the 'int 3' instruction was the cause of the exception. If it was, the exception handler retrieves the address of the exception, and subtracts the process image base plus a delta to recover the original opcode. The opcodes are stored one at a time in the writable buffer. The exception handler retrieves the return address from the stack and uses that as the address from which to resume execution. Upon returning from the exception handler, the virus executes the next 'call' instruction, which will execute another 'int 3' instruction, and cause another exception. This cycle is repeated until all of the original opcodes are decoded.

The use of the 'int 3' instructions serves to make debugging difficult, since the interrupt 3 instruction is used most commonly by debuggers to interrupt execution. Since it is also only a one-byte instruction, it is the most compact way to cause an exception to occur via code execution (ultimately, the most compact way to cause an exception to occur is to branch to a non-readable page, wherein no code is executed, and thus no space is used).

In the unlikely event that an exception occurred during the decoding process and it was not caused by an 'int 3' instruction, the virus simply transfers control to the host, but without restoring either the original stack pointer value or the register that should hold the address of the Process Environment Block. This second part is a bug, because there can be any number of programs that rely on that documented initial value. The first part might also be considered a bug, because a program might attempt to exit by returning directly to the kernel, but this aspect of the environment, though well known, is not documented officially.

DRAGNET

After the decoding is complete, the virus unregisters the Structured Exception Handler that handles the 'int 3' trick, retrieves the host entry point RVA from an unused field in the MZ header, and applies it to the ImageBaseAddress field value (which the virus should have known already, because it disables Address Space Layout Randomization for the file). The resulting value is saved on the stack to allow the host code to be executed later.

The virus continues by setting up a Structured Exception Handler in order to intercept any errors that occur during infection. The virus retrieves the base address of kernel32.dll by walking the InMemoryOrderModuleList from the PEB_LDR_DATA structure in the Process Environment Block. The address of kernel32.dll is always the second entry on the list for all existing versions of *Windows*. The virus resolves the addresses of the bare minimum set of API functions that it needs for replication: find first/next, open, map, unmap, and close. The virus uses hashes instead of names, encoded using the CRC32 method, to avoid the need to store the strings. However, the CRCs are not sorted according to the alphabetical order of the strings they represent, so multiple passes over the export table are required to resolve the imports.

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. The virus also checks that the exports exist by limiting the parsing to the number of exports in the table. The hash table is terminated with a single byte whose value is 0x2a (the '*' character). This is a convenience that allows the file mask to follow immediately in the form of '*.exe', however it also prevents the use of any API whose hash ends (despite the comment in the source code that says 'begin') with that value. As with previous viruses by the same author, this virus only uses ANSI APIs. The result is that some files cannot be opened because of the characters in their names, and thus cannot be infected.

The virus searches in the current directory (only), for objects whose names end in '.exe'. There is a bug in the code in that it does not close the handle that is used to search the directory. As a result, a handle is leaked for as long as the process runs. The search is intended to be restricted to files, but can also include directories, and there is no filtering to distinguish between the two. For each such file that is found, the virus attempts to open it and map an enlarged view of the contents. There is no attempt to remove the read-only attribute, so files that have this attribute set cannot be infected. In the case of a directory, the open will fail, and the map will be empty. The map size is equal to the file size plus a little more than 4KB, to allow the file to be infected immediately if it is acceptable. The value of the size increase is hard-coded in the virus, which is strange, given that the size of the encoded form of the virus is only slightly more than half of that value. Using the post-infection size during the validation stage allows the virus to avoid having to close the file and re-open it with a larger map later. The virus assumes that the handle can be used, and then checks whether the file can be infected.

ALL POINTS BULLETIN

The virus is interested in Portable Executable files for the *Intel x86* platform with no appended data. Renamed DLL files are not excluded. The subsystem value is restricted to GUI mode applications. If the file passes all of these checks, then the virus increases the file size by 4KB+1 bytes. The extra byte serves as the infection marker, because the byte will appear to be appended data, and the virus will not attempt to infect the file. The virus increases the virtual and physical sizes of the last section, and the SizeOfImage, by 4KB. The section attributes are marked as writable, but not executable. This is possible because of a change that the virus makes later to the DLL Characteristics field (see below). It also takes advantage of an undocumented behaviour of Data Execution Prevention, in the name of compatibility. If execution begins within a section (not the file header) that is not marked as executable, and if the file is not marked as NX_COMPAT, then all sections (and the file header) are marked internally as executable, execution is still allowed to proceed, and no exception will occur. However, regardless of the NX_COMPAT setting, if execution begins in an executable section and a transfer of control is made to a non-executable section, then an exception will occur.

The virus saves the original entry point in the unused field in the MZ header, and then sets the host entry point to point directly to the virus code. The virus updates the delta that is used for the decoding, but nothing further is done to the virus body. The encoded bytes are not altered, so the virus body is essentially constant. Then the virus copies itself to the host file.

The virus zeroes the DLL Characteristics field in the PE header. This has the effect of disabling the 'No eXecute' behaviour, and allowing execution to begin in a non-executable section. The change disables Address Space Layout Randomization for the file, which would allow hard-coded address to work correctly if the virus author had decided to use them. The change also enables Structured Exception Handling in the file, which the virus requires. The virus zeroes the RVA of the Load Configuration Table in the data directory. This has the effect of disabling SafeSEH, but it affects the per-process GlobalFlags settings, among other things.

The virus code ends with an instruction to force an exception to occur, which is used as a common exit condition. However, it does not recalculate the file checksum, and does not restore the file's date and timestamps either, making it very easy to see which files have been infected.

CONCLUSION

We have seen hidden encoded data before, where each opcode is decoded individually, but normally the decoders are highly polymorphic and very large (see [3] for an extreme example). Tussie approaches the smallest possible implementation of the idea, and is quite elegant in its simplicity. Fortunately, the simplicity of the implemention also results in a simplicity of detection.

SUMMARY: W32/TUSSIE

Type: Current directory direct-action infector.

Infects: *Windows* Portable Executable files.

Payload: None.

Removal: Delete infected files and restore them from backup.

REFERENCES

[1] Ferrie, P. Doin' the eagle rock. Virus Bulletin, March 2010, p4. http://www.virusbtn.com/pdf/magazine/2010/201003.pdf.

[2] Ferrie, P. So, enter stage right. Virus Bulletin, June 2012, p4. http://www.virusbtn.com/pdf/magazine/2012/201206.pdf.

[3] Ferrie, P. Leaps and bounds. Virus Bulletin, December 2006, p4. http://www.virusbtn.com/pdf/magazine/2006/200612.pdf.