MALWARE ANALYSIS 1

ANOTHER TUSSLE WITH TUSSIE

Peter Ferrie
Microsoft, USA

When one has a nice idea – such as a tricky method for encoding data – it is common to take that idea and improve on it. It is rare to see someone take such an idea and degenerate it, but that is essentially what we have with the W32/Tussie.B virus.

SECOND PLACE GOES TO...

The virus begins by pushing the host ImageBase value from the Process Environment Block onto the stack. It adds the RVA of the host original entry point to that value, 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). The virus registers a Structured Exception Handler and 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 for *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). We have seen the effect of this problem elsewhere recently [1].

IMPORT/EXPORT BUSINESS

The virus resolves the addresses of the API functions that it requires. It uses hashes instead of names, but the hashes are 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 Structured Exception Handler to deal with any problems that occur. 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. The assumption is that each hash is unique and thus that 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 could 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 allocates a memory block and writes a random byte to the buffer. We do not know why the write is performed. The virus sets the error mode to prevent warning messages for all of the supported error types, even though none of them will trigger anyway. It registers another Structured Exception Handler and then decompresses an MZ/PE header combination 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 an RLE format could compress the data further, that gain would be more than lost by the size of the decompression code.

JOIN THE DOTS

The virus drops the resulting file – which contains only the headers and a page full of zeroes. 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, which is unnamed, to reduce the number of bytes that need to be written during the decompression phase. The section has only the executable flag 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.

After dropping the file, the virus 'runs' it (despite it containing no code, in a way that is entirely different from the 'virtual code' technique [1]) and specifies that it is the target of a debugger. This action allows the virus to intercept debug events as they occur, which becomes important later. It registers yet another Structured Exception Handler, and then waits for debug events to occur within the child process. The virus is interested only in the breakpoint debug event, and ignores all others. When a breakpoint debug event occurs, the virus checks if it is the first such event. If it is, the event corresponds to the debug breakpoint that is triggered by the thread that *Windows* creates when a process is being debugged. Since this thread executes before the main thread does, it is the perfect place to perform certain actions before the main thread begins to execute. The virus takes advantage of this to insert some content into the process – in this case a series of call instructions into an array of 'int 3' instructions. The virus then resumes execution of the child process.

BREAKDANCING

With the new content in place, a series of intentional breakpoint events occurs in the child process, which the virus intercepts. For each of these breakpoint events, the virus registers another Structured Exception Handler, and then queries the execution context for the child process. The virus reads a value from the stack of the child process and uses this to set the pointer to the next instruction to execute within the child process. Note that the stack pointer is not adjusted at this point, leading to stack leakage corresponding to the number of instructions that are executed by the child. Thus, if the child executed a sufficient number of instructions, a stack fault would occur which would trigger an event (technically, two events – a stack overflow exception, followed by an access violation exception) that the virus would ignore, causing the fault to occur again, and leading to an infinite loop.

The virus uses the exception address to decode one byte of code for each iteration, and place it in the virus body. Once the child has finished executing, the virus verifies that the entire body has been decoded correctly. If the decoding is correct then the virus runs the decoded body.

The body begins by searching the current directory (only) for all objects. It is really only interested in files, but it will examine everything that it finds. For each object that is found, the virus will attempt to remove the read-only attribute, open it, and map a view of it. For directories, the open will fail and the map will be empty. For files, the entire file is mapped into memory, if the file can be opened. However, as with many previous viruses by the same author, this one uses only 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 is interested in Portable Executable files for the 32-bit *Intel* x86 platform, which are executable but not DLLs, system files, or ROM images, and which target the GUI subsystem. The virus checks that the optional header size is the standard value (which is a good idea to exclude unconventional modifications such as those made by many runtime compressors). The virus also excludes files that appear to have a Load Configuration Table, because this contains protections such as SafeSEH, which are difficult to modify in a way that would allow the virus to raise exceptions at arbitrary addresses. The virus requires that the file has a base relocation table that begins at exactly the start of the last section, and which is at least as large as the virus body.

TOUCH AND GO

If the file passes those checks, the virus marks the last section as writable and executable, and then copies itself over the relocation table. The use of the executable characteristic allows the virus to run in case the file opts into Data Execution Prevention, or if the system enforces it for the process. The virus clears only two flags in the DLLCharacteristics field: IMAGE_DLLCHARACTERISTICS_FORCE_INTEGRITY and IMAGE_DLLCHARACTERISTICS_NO_SEH. This allows signed files to be altered without triggering an error, and enables Structured Exception Handling. Interestingly, the use of Structured Exception Handling, and thus the need to clear the flag, is essentially optional. The virus could use Vectored Exception Handling instead, which is entirely independent of the value of the flag. The reason for the NO_SEH flag is to increase the security of a process by disallowing the use of attacker-defined exception addresses when a stack buffer vulnerability is exploited. Since Vectored Exception Handlers reside on the heap instead of the stack, they are less vulnerable to buffer overflow exploits.

The virus also zeroes the Base Relocation Table data directory entry. This is probably intended to disable ASLR for the host, but it also serves as the infection marker. Unfortunately for the virus writer, this has no effect at all against ASLR. The 'problem' is that ASLR does not require relocation data for a process to be 'relocated'. If the file specifies that it supports ASLR, by having the IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE flag set in the DLLCharacteristics field, and by not having the IMAGE_FILE_RELOCS_STRIPPED flag set in the COFF Characteristics field, then it will always be loaded to a random address. The only difference between the presence and absence of relocation data is that without the relocation data, no content in the process will be altered. *Windows* assumes that if the process specifies that it supports ASLR, then it really does support ASLR, no matter what the structure of the file looks like. The result is that a process that had a relocation table overwritten by the virus will crash when it attempts to access its variables using the original unrelocated addresses. Alternatively, if the platform does not support ASLR (i.e. *Windows XP* and earlier), and if something else is already present at the host load address (or if the load address is intentionally invalid to force the use of the relocation table), then the file will no longer load.

Finally, the virus sets the host entry point to point directly to the virus code 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.

EXCEPTIONAL BEHAVIOUR

The exception handler unmaps the view and closes the file handles, restores the file attributes, and then continues the search for more objects. After all the objects have been examined, the virus raises another exception to unwind further. That exception handler closes the thread and process handles of the debugged process, and then raises yet another exception. At this point, the virus wants to restore the error mode and free the allocated memory, before raising the final exception to run the host code, but there is a bug in this code (technically, two bugs of the same kind): the wrong offset is used when indexing into the structure that holds the API addresses. Instead of calling the SetErrorMode() function, the virus calls the ReadProcessMemory() function, and instead of calling the GlobalFree() function, the virus would call the GlobalAlloc() function if the ReadProcessMemory() function returned successfully. However, the improper parameters on the stack for the ReadProcessMemory() function cause the API to raise its own exception, which the virus intercepts.

The final exception handler restores the registers and transfers control to the host. Of course, since the GlobalFree() function was never called, the virus leaks a small amount of memory as a result. The bug that causes the early exception is the kind of problem that can only be found by single-stepping through the code, since an exception of some kind is the expected behaviour, it just happens too early in this case.

CONCLUSION

The Tussie.A virus [2] showed us a way to hide encoded data by using something approaching the smallest possible implementation of individual opcode decoding. The Tussie.B virus takes a step back by introducing a layer of indirection for no obvious purpose other than to make debugging a bit more difficult. Fortunately, the simplicity of this implementation still results in simplicity of detection.

REFERENCES

[1] http://www.virusbtn.com/pdf/magazine/2013/201310.pdf.

[2] http://www.virusbtn.com/pdf/magazine/2012/201208.pdf.