MALWARE ANALYSIS

IT'S A BIRD, IT'S A PLANE, IT'S FOOPERMAN!

Peter Ferrie
Microsoft, USA

It is sometimes said that one man's trash is another man's treasure. In this case, we might say 'one man's data is another man's code'. What we have here is a virus that uses the FPU to magically transform a block of data into executable code, but the secret is in the details of W32/Fooper.

EXCEPTIONAL BEHAVIOUR

The virus begins by walking the Structured Exception Handler chain to find the topmost handler. At the same time, it registers a new exception handler which points to the host entrypoint. The reason for this will be described below. Once the topmost handler has been found, the virus uses the resulting pointer as the starting location in memory for a search for the MZ and PE headers of kernel32.dll. Once the headers have been found, the virus parses the export table to find the APIs that it needs for infection.

This leads us to the first bug in the code. The problem with the SEH walking method is that in *Windows Vista* and later, the topmost handler no longer points into kernel32.dll but points into ntdll.dll instead. The result is a crash on these platforms, because the virus assumes that the APIs will be found, and falls off the end of a buffer because they do not exist.

HAPI HAPI, JOY JOY

If the virus finds the PE header for kernel32.dll, it resolves the required APIs. The virus uses hashes instead of names, but the hashes are sorted alphabetically according to the strings 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. 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. This becomes important later.

After retrieving the API addresses from kernel32.dll, the virus attempts to load 'sfc_os.dll'. If this attempt fails, then the virus attempts to load 'sfc.dll'. If either of these attempts succeeds, then the virus resolves the SfcIsFileProtected() API. The reason the virus attempts to load both DLLs is that the API resolver in the virus code does not support import forwarding. The problem with import forwarding 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 the case of the SfcIsFileProtected() API, the API is forwarded in *Windows XP* and later from sfc.dll to sfc_os.dll.

CULTURAL AWARENESS

The virus retrieves both the ASCII and Unicode versions of the required APIs. One minor detail exists here, which is that because of the way in which the virus uses the APIs, it must swap the address of the CreateFileW() API and the CreateFileMappingA() API on the stack, even though this goes against the alphabetical ordering. The reason for the swap is because the virus requires the ASCII and Unicode versions of any given API to be sequential on the stack. This allows for transparent use of the appropriate API.

Specifically, the virus calls the GetVersion() API to determine the current *Windows* platform, and uses the result to select the appropriate API set (ASCII for *Windows 9x/Me*, and Unicode for *Windows NT* and later). Yes, this virus still supports *Windows 95*! This is because the infection engine used here is the same as the one we first saw the virus author use in 2002. In fact, the only update to the code is the support for Data Execution Prevention (DEP), but setting the executable bit in the section characteristics when appropriate.

The GetVersion() API returns a bit that specifies whether the platform is *Windows 9x*-based (1) or *Windows NT*-based (0). The virus multiplies this value by four, adds the stack pointer value to it, and places the result in a register. Now, whenever the virus wishes to use an API which exists in the two forms, it simply calls the function relative to the register. As such, there is no need ever to check for the platform again. For example, the virus can call '[ebp+CreateFile]', where ebp contains the platform-specific value. If ebp is zero, then the CreateFileW() API is called, and if ebp is four, then the CreateFileA() API is called. This is why the reverse alphabetical order is important for the API addresses on the stack, and why the CreateFileW() and the CreateFileMappingA() API addresses had to be swapped.

LET'S DO THE TWIST

After finishing with the API trickiness, the virus initializes its Random Number Generator (RNG). The RNG is interesting in itself, since it is neither the usual GetTickCount()-based randomizer, nor the Knuth-inspired algorithm. Instead, the virus uses a complex RNG known

as the 'Mersenne Twister', named after the kind of prime number at its heart. The virus author has used this RNG in each of his viruses for which he requires a source of random numbers. Curiously, only one virus created by a different virus author has ever used the same RNG.

The virus then searches for files in the current directory and all subdirectories, using a linked list instead of a recursive function. This is important from the point of view of the virus author, because the virus infects DLLs, whose stack size can be very small. The virus avoids any directory that begins with a '.'. This is intended to skip the '.' and '..' directories, but in *Windows NT* and later, directories can legitimately begin with this character if other characters follow. As a result, those directories will also be skipped.

FILTRATION SYSTEM

Files are examined for their potential to be infected, regardless of their suffix, and will be infected if they pass a very strict set of filters. The first of these filters is the support for the System File Checker that exists in *Windows 98/Me*, and *Windows 2000* and later. Since the directory searching on the *Windows 9x/Me* platforms uses ANSI paths, and since the SfcIsFileProtected() API requires a Unicode path, the virus converts the path from ANSI to Unicode, if appropriate, before calling the API.

The remaining filters include the condition that the file being examined must be a *Windows* Portable Executable file, a character mode or GUI application for the *Intel* 386+ CPU, that the file must have no digital certificates, and that it must have no bytes outside of the image. Additionally, if the file is a DLL, then it must have an entrypoint.

TOUCH AND GO

When a file is found that meets the infection criteria, it will be infected. The virus resizes the file by a random amount in the range of 4 to 6KB in addition to the size of the virus. This data will exist outside of the image, and serves as the infection marker.

If relocation data is present at the end of the file, the virus will move the data to a larger offset in the file and place its own code in the gap that has been created. If no relocation data is present at the end of the file, the virus code will be placed there. The virus checks for the presence of relocation data by checking a flag in the PE header. However, this method is unreliable because *Windows* ignores this flag, and relies instead on the base relocation table data directory entry.

The virus increases the physical size of the last section by the size of the virus code, then aligns the result. If the virtual size of the last section is less than its new physical size, then the virus sets the virtual size to be equal to the physical size, and increases and aligns the size of the image to compensate for the change. The virus also changes the attributes of the last section to include the executable and writable bits. The executable bit is set in order to allow the program to run if DEP is enabled, and the writable bit is set because the RNG writes some data into variables within the virus body.

The virus alters the host entrypoint to point to the last section, and changes the original entrypoint to a virtual address prior to storing the value within the virus body. This act will prevent the host from executing later if the host is built to take advantage of Address Space Layout Randomization (ASLR). However, it does not prevent the virus from infecting files first. The lack of ASLR support might be considered a bug unless we remember that ASLR was not introduced until *Windows Vista*, which, as noted above, the virus does not support. What is strange, though, is that changing the entrypoint in this way affects DLLs in the same way. Thus, if an infected DLL is relocated because of an address conflict, then it, too, will fail to run. This is despite the fact that in other viruses the virus author has demonstrated the ability to infect DLLs correctly, by calculating the virtual address of the entrypoint dynamically. Since this method is equally applicable to ASLR-aware files, the same method could have been used in both cases.

ROOT-BEER FLOATS

At this point, the virus generates a new decryptor for the virus body. It begins by choosing a random CPU register (with the exception of the ESP register), whose purpose depends on whether or not the decryptor was using it previously. In the .A and .C variants, the virus examines the register initialization code in the decryptor (the .B variant has no such section) and makes a note of which registers are in use. At the same time, it checks whether the chosen register is already in use (there is one register which is not used in the register initialization code – this is used as the base register for the memory accesses). This is a very elegant routine.

The decryptor contains three sections (in the .A and .C variants; two in the .B variant) where the chosen register might have been used: it might have been used in the register initialization code, it might have been used as the base register for the memory accesses, and it might have been used as the counter register. In any case, if the chosen register is used already, then the virus replaces it with the

unused register. The virus always replaces the scale register for the memory accesses with the chosen register. The construction of the decryptor then proceeds differently for each of the variants.

.A DECRYPTOR

The .A variant replaces any references to the chosen register in the arithmetic instructions with the unused register. It swaps the register initialization lines randomly. The decryptor is a set of simple arithmetic operations, but it uses all of the registers, and it is sufficiently complex that it cannot be x-rayed.

The virus generates an FPU 'fsave' instruction using the unused register, and assigns random initial values to all of the registers except for the counter register. The virus also generates a series of FPU 'fld' (Floating-point LoaD) instructions using the unused register: one fld instruction for each ten bytes of the decryptor, for a total of 80 bytes. The offset for the fld instructions is a random multiple of ten within that 80-byte block, but since the FPU registers (known as 'stn', where 'n' is the slot number, from zero to seven) exist on a stack, the lines are loaded in a fixed order. That is, the first ten bytes that are loaded correspond to the last ten bytes in the decryptor; the second ten bytes that are loaded correspond to the second-to-last ten bytes in the decryptor, and so on.

.B DECRYPTOR

The .B variant uses a decryptor that is a set of simple arithmetic operations that use immediate values, but it is sufficiently complex that it cannot be x-rayed.

The virus generates an FPU 'fsave' instruction using the unused register, and assigns random values to all of the arithmetic operations. The virus also generates a series of MMX 'movq' (MOVe Quadword) instructions using the unused register: one movq instruction for each eight bytes of the decryptor, for a total of 64 bytes. The offset for the movq instructions is a random multiple of eight within that 64-byte block, and since the MMX registers (known as 'mmn' – strangely, not 'mmxn' – where 'n' is the register number, from zero to seven) can be assigned explicitly, the order of the loads is also random. That is, the first register that is loaded might be any one of the eight MMX registers, however the bytes that the register holds will always correspond to the same eight bytes in the decryptor.

The decryptor of the .B variant is somewhat weaker than the decryptor in either the .A or the .C variant, partly because of the way in which the MMX registers are used within the CPU. Specifically, the MMX registers share the same slots within the FPU as the standard FPU registers. However, since the FPU registers are each ten bytes long, while the MMX registers are only eight bytes long, the FPU automatically fills the last two bytes of the slot with the value 0xFF. Because of the way in which the decryptor works (see below), these 0xFF bytes must be skipped. The virus achieves this by further shortening the contents of the slots (hence the simple arithmetic instructions accepting only immediate values), and placing a jump instruction at the end of the slot.

The virus author could have used an instruction that would incorporate the 0xFF bytes, which would have avoided the jump, and thus would have increased the usable size of the slots by one byte. There are two candidate values that would serve the purpose: 0x80 and 0x82. Both values decode to the same instruction when followed by 0xFF 0xFF: CMP BH, FF. It seems likely that the virus author knew this, but given the style of the decryptor, the additional bytes might not have seemed sufficient to insert any further instructions (the result of the compare could have been used, for example, but the decryptor would look quite different).

.C DECRYPTOR

The .C variant replaces any references to the chosen register in the arithmetic instructions with the unused register. It swaps the register initialization lines randomly. The decryptor is a set of simple arithmetic operations, but it uses all of the registers, and it is sufficiently complex that it cannot be x-rayed.

The virus generates an FPU 'fxsave' instruction using the unused register, and assigns random initial values to all of the registers except for the counter register. The virus also generates a series of SSE 'movdqu' (MOVe Double-Quadword Unaligned) instructions using the unused register: one movqdu instruction for each 16(!) bytes of the decryptor, for a total of 128 bytes(!). The offset for the movdqu instructions is a random multiple of 16 within that 128-byte block, and since the XMM registers (known as 'xmmn', where 'n' is the slot number, from zero to seven) can be assigned explicitly, the order of the loads is also random. That is, the first register that is loaded might be any one of the eight XMM registers, however the bytes that the register holds will always correspond to the same 16 bytes in the decryptor. Further, since the XMM registers occupy their own space within the FPU, the entire slot is available for use, and the virus takes advantage of this. The virus places a jump instruction after the last movdqu instruction in order to reach the fxsave instruction.

FSAVE THE WORLD

The virus uses the fsave instruction (or the fxsave instruction in the .C variant) in order to do something special with the loaded registers.

Prior to the execution of the f[x]save instruction, the registers exist essentially in isolation. While the registers can be manipulated individually, they exist as separate data items. However, when the f[x]save instruction is executed, the registers are stored in a particular order to the memory location that is specified by the instruction. The order is first to last (st0 or [x]mm0, then st1 or [x]mm1, then ... st7 or [x]mm7). There is no padding between the stored registers, allowing them to form a block of executable code if the contents are valid instructions. That is the case here. However, the virus goes further, by specifying an address for the f[x]save instruction such that the next instruction to execute comes from the first of the stored registers, and execution proceeds from there. This act is self-modifying in an interesting way, since the f[x]save instruction is overwritten by the data that the f[x]save instruction causes to be stored.

APPENDICITIS

After constructing the decryptor, the virus will append its body and encrypt it with a routine that performs the reverse actions of the decryptor.

Once the infection is complete, the virus calculates a new file checksum, if one existed previously, before continuing to search for more files.

Once the file searching has finished, the virus will allow the host code to execute by forcing an exception to occur. This technique appears a number of times in the virus code and is an elegant way to reduce the code size, in addition to 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.

CONCLUSION

Causing the FPU to reorder some data, such that it then becomes meaningful in a different context, is an interesting idea. It's a bit like a word puzzle, where the letters have been arranged randomly. Who knew that the FPU could solve anagrams?