MALWARE ANALYSIS 1

THIS SIG DOESN'T RUN

Peter Ferrie
Microsoft, USA

Some virus writers like to brag about themselves or their creations. Sometimes the bragging is done via the virus author's choice of name for the virus. Of course, it's rare that the content justifies the bragging, since lots of viruses contain bugs. Here we have the ultimate combination of bragging and bugs. The author of the virus gave it the name 'Sigrún', which is Old Norse for 'victory rune'. However, there is no victory because the virus does not work (the reason why will not be described here). Just in case the bug is fixed, let's call it W64/Svafa, because 'Sváfa' is the previous incarnation of Sigrún, and the name is thought to derive from the word for 'sleep-maker', which seems appropriate.

Í UPPHAFI (INITIALLY)

The first generation of the virus begins by saving the relative address of the original entrypoint on the stack. However, depending on the imagebase value that was used when building it, this value might be completely wrong. The virus applies the current imagebase value from the ImageBaseAddress field in the Process Environment Block, in order to account for Address Space Layout Randomization (ASLR). This is an interesting way to deal with ASLR. It is more common simply to calculate the difference between a branch instruction and the host entrypoint.

The virus also saves the current stack pointer to a field in its body. Using this value the virus can undo any changes to the stack at any point during the execution of the code. This is particularly important during API resolution, since the virus cannot easily know how many APIs have been saved before something goes wrong.

The virus begins by retrieving the base address of ntdll.dll. It does this by walking the InMemoryOrderModuleList from the PEB_LDR_DATA structure in the Process Environment Block. This is compatible with the changes that were made in *Windows 7*. The virus also saves the pointer to the current position in the list so that it can resume the parsing later to find the base address of kernel32.dll. If the virus finds the PE header for ntdll.dll, it resolves the two required APIs: RtlAddVectoredExceptionHandler and RtlRemoveVectoredExceptionHandler.

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. 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 also checks that the exports really 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'. The virus retrieves the base address of kernel32.dll by fetching the next entry in the list, using the pointer that was saved earlier. The same routine is used to retrieve the addresses of the API functions that it requires, and which is the absolute minimum set of APIs that it needs for replication – find first/next, open, map, unmap, close.

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'. This is intended to be restricted to files, but can also include any directories that have such a name, and there is no filtering to distinguish between the two cases.

For each such file that is found, the virus attempts to open it and map a view of the contents. There is no attempt to remove the read-only attribute, so files that have that attribute set cannot be infected. In the case of a directory, the open will fail, and the map will be empty. The virus registers an exception handler at this point, and then checks whether the file can be infected.

RELOCATION ALLOWANCE

The virus is interested in Portable Executable files for the x64 platform. Renamed DLL files are not excluded, nor are files that are digitally signed. The subsystem value is checked, but incorrectly. The check is supposed to limit the types to GUI or CUI but only the low byte is checked. Thus, if a file uses a (currently non-existent) subsystem with a value in the high byte, then it could potentially be infected too.

The virus checks the Base Relocation Table data directory to see if the relocation table begins at the start of the last section. If so, then the virus assumes that the entire section is devoted to relocation information. This could be considered to be a bug. The virus checks that the physical size of the section is large enough to hold the virus code. There are multiple bugs with this check alone. The first bug is that the size of the relocation table could be much smaller than the size of the section, and other data might follow it. The data will be overwritten when the virus infects the file.

Further, the value in the Size field of the Base Relocation Table data directory cannot be less than the size of the

relocation information, and it cannot be larger than the size of the section. This is because the value in the Size field is used as the input to a loop that applies the relocation information. It must be at least as large as the sum of the sizes of the relocation data structures. However, if the value were larger than the size of the relocation information, then the loop would access data after the relocation table, and that data would be interpreted as relocation data. If the relocation type were not a valid value, then the file would not load. If the value in the Size field were less than the size of the relocation information, then it would eventually become negative and the loop would parse data until it hit the end of the image and caused an exception.

The second bug is that by checking only the physical size and not the virtual size, whatever the virus places in the file might be truncated in memory if the virtual size of the section is smaller than the physical size of the section. Both of these bugs are also present in the W64/Holey virus [1], by the same virus author.

However, it is the third bug that is very serious, very silly, and should have been very obvious. It is that the size of the virus code is less than one third of the total size that is needed to hold the data that the virus adds to a file. The true size of the virus is the size of the virus code multiplied by three, plus the size of the decoder. Thus, the section might be nowhere near large enough for the file to be infected correctly, but the virus won't notice the problem.

I'M LOOKING FOR... MY MASK!

If the section appears to be large enough, then its attributes are marked as executable and writable. The virus 'encrypts' its code using a byte-mask technique. There are two tables involved here, both of which are the same size as the virus code. One table contains the byte-mask, and the other contains a selection of host bytes. For each byte in an eight-byte set, the virus chooses randomly if it will be encoded or not.

If the byte is to be encoded, then the byte-mask will have the top bit set in the mask table, and the other seven bits will be set to a random value. The corresponding entry in the host table will contain the host byte, and the value at the original location in the host will be set to a random value.

If the byte is not to be encoded, then the byte mask will have the top bit clear in the mask table, and the other seven bits will be set to a random value. The corresponding entry in the host table will contain a random value, and the value at the original location in the host will maintain its original value. This process is repeated over the entire host body.

Once the encoding is complete, the virus zeroes the values in the Offset and Size fields of the Base Relocation Table data directory, saves the original entrypoint in the virus body, and then sets the host entrypoint to point directly to the virus code.

TOUCH AND GO

The virus code ends with an instruction to force an exception to occur. This is used as a common exit condition. However, the virus does not recalculate the file checksum, even though it might have changed as a result of infection. It also does not restore the file's date and timestamps, making it very easy to see which files have been infected, even though the file size does not change.

I LIKE TO 'MOV' IT

When an infected file is executed, the virus decodes itself. The special thing here is that the decoding is done using three MMX instructions, one of which might be considered to be a bit obscure: MASKMOVQ. The MASKMOVQ instruction accepts two parameters which correspond to the two tables that the virus constructed. For each byte in the mask table whose high bit is set, the corresponding byte in the host table is copied into the host body at the location to which the EDI register points at that moment. If the high bit is clear, then no copy occurs. Thus, prior to decoding, the virus body is a mixture of real values and random values, and so is the host table.

There have been suggestions that MMX is not safe to use on the 64-bit platform, because the floating-point state (and thus the MMX state, which shares the same memory region) is not saved, but this is not the case. In user mode, the floating-point state is always saved during a context switch. Therefore, there is no problem at all for user-mode processes. In kernel mode, the context is not saved, but it was not saved on the 32-bit platform either, so there is no new behaviour here.

CONCLUSION

The MASKMOVQ technique is another surprise from the MMX instruction set, and one which makes static analysis a bit inconvenient. However, anti-malware emulators will see just another instruction, and shortly afterwards, just another virus.

REFERENCES

[1] Ferrie, P. 'Holey' virus, Batman! Virus Bulletin, September 2011, p.4. http://www.virusbtn.com/pdf/magazine/2011/201109.pdf.