MALWARE ANALYSIS

'HOLEY' VIRUS, BATMAN!

Peter Ferrie
Microsoft, USA

Some might think that all of the entrypoints in Portable Executable (PE) files are known – but they would be wrong. As we saw with the W32/Deelae family [1], a table that has been overlooked for more than a decade can be redirected to run code in an unexpected manner. Now, a table that was used in *Windows* on the *Itanium* platform also exists on the x64 platform, and (surprise!) it can be misused too. The W64/Holey virus shows us how.

HAPI HAPI, JOY JOY

The virus begins by retrieving the address of ntdll.dll by walking the InMemoryOrderModuleList list from the PEB_LDR_DATA structure in the Process Environment Block. This is an unusual choice – the InLoadOrderModuleList list is more common – but it is not incorrect, and it 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 address of kernel32.dll.

If the virus finds the PE header for ntdll.dll, it resolves the required APIs. It 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, rather than parsed 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 API addresses end up in reverse order in memory. Interestingly, the virus checks that the exports really exist by limiting the parsing to the number of exports in the table. This is probably a great situation for emulators that don't export all of the right functions – the sample will run the host code instead of crashing – but it doesn't benefit the virus in any way.

The table is terminated with a single byte whose value is 0x2a (the '*' character). This is used to allow the file mask to follow immediately in the form of '*.exe'. We do not know whether the character was chosen because of the mask, or whether the mask was placed there simply because of the chosen character.

The virus retrieves the 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. Despite the strong similarities with some other viruses that support Unicode, 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 files whose names end in '.exe'. 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. The virus registers an exception handler at this point, and then checks if 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. In fact, there are many common checks that are missing from this virus. Perhaps it was written in a hurry to meet some kind of deadline. The code also lacks some obvious optimizations, again suggesting that it was written hastily.

The virus checks the Base Relocation Table data directory to see if the relocation table is the last section. The check is very specific – it is not enough that the relocation table exists *within* the last section, but it must *be* the last section. That is, it starts at the start of the section, and the assumption is that the entire section is devoted to relocation information – which can cause a problem. The virus checks that the value in the SizeOfRawData field is at least 687 bytes long. Of course, the relocation table could be much smaller than this, and other data might follow it. This data will be overwritten when the virus infects the file.

We do not know why the SizeOfRawData field was used instead of the value in the Size field in the data directory, because the value in the Size field 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 was 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.

EXCEPTIONAL BEHAVIOUR

The virus also requires that the file has no Exception Table. If this is the case, then the virus creates a RUNTIME_FUNCTION structure and places it at the start of the last section. The RUNTIME_FUNCTION structure contains the begin and end addresses of the code which will be described by the UNWIND_INFO structure, and a pointer to that structure. The virus sets the begin address to equal the host entrypoint value, and the end address to one byte later than that. The UNWIND_INFO structure pointer is set to the address immediately after the pointer, and the UNWIND_INFO structure is placed directly after the RUNTIME_FUNCTION structure. The UNWIND_INFO structure exists to allow *Windows* to unwind the stack if an exception occurs. However, the virus does not need to worry about such a thing. All it has to do is set the appropriate flags and store a callback pointer in the structure. Then, when an exception occurs, the virus code will be called.

The virus makes the last section both writable and executable. It sets the Exception Table data directory entry to point to the start of the last section, and sets the Size field appropriately. The virus copies itself to the file, and zeroes some flags in the header. Of particular interest are the flags that correspond to ASLR (Address Space Layout Randomization), NX (No eXecute) and NO_SEH (No Structured Exception Handling). Zeroing the ASLR flag ensures that the image will not move in memory. This is irrelevant for the virus, though, because the virus code is entirely position-independent. It might be a bit of left-over code from a previous virus by the same author. Zeroing the NX flag enables the virus to run from a section that is not marked as executable. Again, this is irrelevant for the virus because the virus marks its code section as executable. However, by zeroing the NO_SEH flag, the virus enables exception handling to be called by the file. If the flag were not cleared, then *Windows* would terminate the application at the moment that an exception occurred.

The virus zeroes the Base Relocation Table data directory. This is the only way to ensure that *Windows* does not attempt to read the relocation data. Even though there exists a flag that can be set in the PE header which is supposed to tell *Windows* that the relocation data has been removed, *Windows* ignores this flag in certain circumstances.

YOU HAVE BEEN INTERRUPTED

At this point the virus attempts to find the section that contains the entrypoint by searching for the first section which ends after the entrypoint. There are two bugs here, one is relatively minor, however the other is fatal for the host. The minor bug is that the virus assumes that the entrypoint is located within a section. It is quite possible to place the entrypoint in the file header. It is possible to place the entrypoint outside of the image, and through a bit of trickery, cause it to 'move' back inside the image (the details about how this is done are not relevant here). It is possible for the file to contain no sections, but still have the appropriate values in the appropriate places. Of course, these are edge cases that are not very interesting to consider.

However, the fatal bug relates to the section scanning. When the virus finds a section which ends after the entrypoint, it marks that section as writable and attempts to fetch the byte at the relative virtual address that correponds to the value of the entrypoint. The byte is saved in the virus body, and replaced with an interrupt 3 instruction. The idea here is that when the host is executed, the interrupt 3 instruction will cause an exception that will be handled by the callback whose pointer is in the Exception Table. The bug is that even after the first such section is found (the entrypoint section), the loop is not exited. Instead, every section after the entrypoint section will also be treated as though it were the entrypoint section. What happens next depends on several conditions. The entrypoint value is converted to a physical address by adding the difference between the PointerToRawData and the VirtualAddress. The addition will occur for each section after the entrypoint section, usually resulting in a continually decreasing value that still points within the entrypoint section. If the value becomes negative (for example, if the initial entrypoint value is small, and the difference is large), then an exception will occur when attempting to fetch the byte from the section. The exception will cause the loop to exit, and everything will appear to be fine – this is probably what happened when the virus was tested, and is probably the reason the bug was not found. However, if the entrypoint value is large enough and if the difference is small enough (it can even be zero, if the file alignment value matches the section alignment value), then the value can survive several, and possibly all of the iterations of the loop, resulting in multiple bytes being replaced in the host entrypoint section.

The bug leads to two other problems, one of which is benign, and the other one causes the host to be damaged. The first problem is that when the last section is examined,

if the entrypoint RVA is larger than the size of that section, then an exception will occur and the infection routine will exit. This is fine because when the loop completes, an exception is raised anyway. The second problem is that, as noted above, the virus might alter one byte in multiple places within the entrypoint section. If those bytes are not all the same (or if they are, but the file alignment and section alignment match, such that the same byte is fetched more than once), then when the exception occurs during execution, the original byte cannot be restored. Further, if those bytes are data, then the host might not behave correctly even if the bytes are all the same because the virus will never have a chance to restore them.

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, and it 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.

ANCIENT HISTORY

It's funny, in a way, that I described a variation of this technique at a time when *Windows NT* was still current. The idea was that by changing the file format in a particular way, an exception would be raised during the file *load*. At that point, nothing in the host had been executed. Given an Exception Table with the right layout, it should have been possible to cause the handler to be executed. (Try emulating that...) However, as noted above, the Exception Table was not used by *Windows* until the introduction of the *Itanium* platform, so fortunately the technique was not viable.

CONCLUSION

The Exception Table hook is an interesting technique. It allows for light entrypoint obscuring, in much the same way as the Thread Local Storage technique did a decade ago, and it becomes yet another place in the file that needs to be scanned. Only time will tell if it will become as popular.

REFERENCES

[1] Ferrie, P. Deelaed learning. Virus Bulletin, November 2010, p.8. http://www.virusbtn.com/pdf/magazine/2010/201011.pdf.