VIRUS ANALYSIS

CAIN AND ABUL

Peter Ferrie
Symantec Security Response, USA

As the decline in file-infecting viruses continues, it is perhaps fitting that the newest virus for the 64-bit platform, W64/Abul, is less advanced than the one that came before it. Despite this, though, Abul implements some new features that make it interesting in its own way.

BASIC INSTINCT

The virus begins by retrieving the base address of kernel32.dll by following some pointers in the Process Environment Block. This is in contrast to the method that was used previously, which was to search memory from either a return address or an API within the kernel32.dll image. The newer method of retrieving the base address of kernel32.dll is quite common now on the 32-bit platform, though it was first documented in 2002. It is used by a lot of shellcode in exploits, because the Process Environment Block is always available, whereas an API or return address might not be accessible at the time of exploitation.

The first bug appears here. Though the base address of kernel32.dll is now known to the virus, it is not stored anywhere. There is a variable that contains this address already, and it is used later in the code, but its value was assigned by the first-generation code, and not by the virus code itself. Thus, if an infected file is placed on a machine where the base address of kernel32.dll is different from that of the virus writer's machine, then the virus will not work. It seems that the virus writer didn't notice the problem because all of his replications worked perfectly on his own machine. That situation is a nightmare for any developer, but fortunately virus writers don't do tech support. In the words of Dogbert, 'I'm sorry, our software is perfect. The problem must be you'.

SUMTIMES I WONDER

Given the base address of kernel32.dll, the virus proceeds to retrieve the addresses of 30 APIs. Those APIs are mostly related to memory management and file infection. ReleaseMutex() and MessageBoxA() are also in the list, though neither is used in the code. Perhaps the virus writer intended to make a multi-threaded version, but then gave up on the idea. The MessageBoxA() API is probably left over from debugging.

The names of the APIs are not stored as strings. Instead, the virus stores them as values calculated by summing the value of each character in the name, along with the length of the name. This is faster than the more common CRC32 method, but is more likely to suffer from name collisions, resulting in the retrieval of the wrong API address.

Even though the virus retrieves all 30 API addresses, it uses only two of them at this point: VirtualAlloc() and VirtualProtect(). VirtualAlloc() is used to allocate a memory block within the process memory space, but outside of the memory image. VirtualProtect() is used to make that new memory block executable. The virus then copies itself into the new memory block and continues execution from there.

HAVEN'T I SEEN YOU BEFORE?

Once in the new memory block, the virus checks for the presence of a debugger, by looking in a field within the Process Environment Block. This mimics the behaviour of the IsDebuggerPresent() API. If no debugger was found, then the virus retrieves the same 30 API addresses as before, but this time using the kernel32.dll variable instead of the Process Environment Block pointers.

The virus also retrieves the addresses of some compression-related APIs from ntdll.dll, some message-related APIs from user32.dll, some process-related APIs from psapi.dll, and some token-related APIs from advapi32.dll. The compression APIs remain undocumented by *Microsoft*, and marked as 'reserved for system use'. They are intended to be used by the file system for compression of individual files. However, they have been reversed-engineered and well documented (see, for example, http://www.alex-ionescu.com/Native.pdf).

The host code section is then made writable, and the original host code is decompressed into the space originally occupied by the virus code. At this point, the virus attempts to open a mutex, to see if any other copies of the virus are running on the system. If they are, then the virus simply transfers control to the host. Otherwise, the virus prepares to go resident and infect the system.

PRIVILEGED AND CONFIDENTIAL

In order to go resident, the virus attempts to acquire debug privileges. This is necessary for process enumeration and the thread injection that it requires. However, the virus ignores the result of the attempt, even though that subroutine returns a status.

The virus then attempts to enumerate the currently running processes, looking for the csrss.exe process. This attempt will fail if the debug privilege has not been acquired, and another bug appears here. The virus does not check whether the function fails. Instead, the virus checks the number of

process IDs that were returned. However, a quick analysis of the EnumProcesses() API function reveals that the variable that receives the number of process IDs is not initialised if an error occurs within the function. Thus, if the virus has not acquired the debug privilege, it could end up using an unpredictable value for the number of process IDs, and a corresponding list of unpredictable values for the process IDs themselves. If the number of process IDs is large enough, the virus will attempt to access an illegal memory region and crash. In some cases, too, at least some of the unpredictable process ID values could match real process IDs on the system, however it seems unlikely that any of them will match the process ID of csrss.exe.

STILL WONDERING

As with the API names, the 'csrss.exe' string is stored as the sum of the value of each character in the name, along with the length of the name. While that works well for API names, for which the character case is constant, the 'csrss.exe' process name could easily have a different case on some systems, in which case the sum will be different. However, if the virus successfully finds the csrss.exe process, it will inject itself as a new thread within the csrss.exe process. The thread priority is set to the idle level, so that it runs very rarely.

The new thread in the csrss.exe process begins by enumerating the currently running processes, looking for the winlogon.exe process. If it is found, then the virus injects a thread into it. The new thread in the winlogon.exe process is very short. It begins by retrieving the address of the SfcTerminateWatcherThread() API from sfc.dll, then calling it. This API can be called only by a thread within the winlogon.exe process, hence the need for the injected thread. The API does exactly what the name suggests: it terminates the watcher thread. This allows arbitrary modification of all files, including protected ones, until reboot. During boot, the winlogon.exe process will restart the SFC thread and potentially reveal the presence of altered files. To protect against that, the virus deletes '%system%\sfcfiles.dll', which houses the list of protected files. This disables the SFC permanently. The thread then exits.

WAITING FOR GODOT

Meanwhile, the new thread in the csrss.exe process sleeps for two seconds, then creates the mutex to prevent other copies of the virus code from running. The virus does not check the result. By waiting for so long, the virus runs the risk of there being other copies of the virus code running, resulting in several threads fighting for control.

After creating the mutex, the virus begins searching for .EXE files in the c: drive, beginning with the root directory and continuing recursively through all subdirectories. Once the search has completed, the thread will sleep forever.

For any .EXE file that is found, the virus opens it and maps a view of the whole file. The virus writer assumes that any file with the .EXE extension is of the correct format, so there is no check for the 'MZ' or 'PE' signatures. There is also no exception handling, so a malformed file will cause the code to crash, and since csrss.exe is a privileged process, a crash in there will cause significant system instability.

The virus parses the file format, assuming that the file is a Portable Executable. It checks that the executable flag is set in the header, that the COFF magic number corresponds to a 64-bit file, that the values in the CPU field correspond to the *AMD* x64 (the value is identical for the *Intel EM64T*), and that the subsystem is GUI or CUI. The virus avoids infecting DLL and system files.

If all of these checks pass, then the virus attempts to compress the first section in the file. This could be considered the infection marker: if the section cannot be compressed, the file cannot be infected, and presumably an already infected file cannot be compressed further. However, there is an additional requirement: the compression ratio must be sufficiently high that the virus code can fit into the remaining space in the section. The idea of host compression is not new. It was first implemented in the Cruncher virus in about 1993, and more recently in viruses such as Aldebara, Redemption, HybrisF, and Detnat.

If the compression leaves enough space for the virus code, then the virus will append itself to the compressed block, and alter the host entrypoint to point to the virus code.

CONCLUSION

Abul was written to demonstrate that viruses written in C can be almost as small as viruses written in assembler, but it also demonstrates that they can be just as buggy. With nothing left to prove, perhaps the decline in file-infecting viruses can continue.

W64/Abul

Type:	Parasitic memory-resident PE infector.
Size:	3,696 bytes.
Payload:	None.
Removal:	Delete infected files and restore them from backup.