MALWARE ANALYSIS 2

SURF'S UP

Peter Ferrie
Microsoft, USA

The *010 Editor* is a powerful tool for analysing files. By using templates, the editor can decompose a file into its parts and display them in a form that is easily understood. The editor can also alter files, and it supports a scripting language to automate certain tasks. Who would have guessed that one of those tasks would be to infect files, as {W32/1SC}/Toobin demonstrates?

111 EDITOR

The virus begins by pushing the RVA of the host's original entrypoint onto the stack. This allows the virus to work correctly in processes that have ASLR enabled. The virus determines its load address by using a call->pop sequence that contains no zeroes. This is implemented in an unsual way. Usually, the call-pop sequence begins with a jump at the start of the code to the end of the code, and then a call backwards to the second line of the code, where the pop instruction exists. An alternative method is a jump at the start of the code to the third line of the code, and then a call to the second line of code, which contains a jump to the fourth line of code, where the pop instruction exists. Of course, once these techniques were established, rampant copying-without-thinking ensued and essentially, until now, no one has thought about how to improve them. The new technique uses an overlapping call instruction, followed by a long-form increment instruction. The call instruction calls into the last byte of itself, where an 'FF' opcode exists. The 'FF' opcode is followed by a 'C0' opcode, to form an increment instruction, and the increment instruction is followed by the pop instruction. Fewer instructions and fewer bytes, but it seems unlikely that we will see this technique replacing the existing ones.

Once the loading address has been determined, the code falls through to a base64 decoder that does not carry a dictionary. In fact, the entire decoder is smaller than the base64 dictionary itself. The decoding is done algorithmically, which is possible because the transformation is really quite simple. The decoder also uses no zeroes (the reason for which will be described later in the article). The first instruction that the base64 decoder decodes forms the parameter of the penultimate instruction and the last instruction. Thus, part of the base64 decoder is also encoded as base64. This is possible on the *Pentium* and later CPUs because of a change in prefetch queue behaviour. Previously, a set of instructions would be prefetched into a local cache and executed from there, no matter what changes were made to the memory while those instructions were running. This allowed for some interesting anti-debugging tricks, because the presence of the debugger would cause the prefetch queue to be emptied when the debugger gained control. When the debugger yielded control and the original instructions resumed execution, the prefetched instructions would include any modification that was made to the memory, thus the presence of the debugger could be inferred. The *Pentium* changed that behaviour to detect the alteration of memory in the range that was prefetched. When an alteration was detected, the CPU would fetch the modified bytes, just like when a debugger is running. The result is that an instruction can modify the following instruction and the modified instruction will be executed in its modified form.

IMPORT/EXPORT BUSINESS

The decoded code begins by retrieving the base address of kernel32.dll by walking the InLoadOrderModuleList from the PEB_LDR_DATA structure in the Process Environment Block. The address of kernel32.dll is always the second entry on the list. 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. 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 resolves the address of a small number of APIs: open, size, read, seek, write, close, malloc and expand strings. After resolving the APIs from kernel32.dll, the virus loads advapi32.dll in order to fetch the address of some registry-access APIs. The virus needs only two registry APIs – one for opening a key, and one for querying a value.

GETTING PERSONAL

The virus opens the 'HKCU\Software\Microsoft\Windows\CurrentVersion\Explorer\User Shell Folders' registry key, and queries the 'Personal' registry value. The returned data is saved for later. The virus encodes itself using the base64 algorithm and inserts the result into the body of an *010 Editor* script. The virus expands the 'Personal' registry data that was returned earlier, to replace indirect variables with their absolute value, then creates a file named 'r' in the resulting directory. The virus writes the *010 Editor* script into this file.

The virus queries the 'Local AppData' registry value which is located under the 'User Shell Folders' that the virus opened earlier. The returned data is expanded in the same way as for the 'Personal' registry data. The virus appends the name of the *010 Editor*'s global configuration file to the string. Note that this path is specific to version 3 of the *010 Editor*. In verson 4, the directory to which the 'AppData' registry value points was used as the base directory. The subdirectory structure was also changed from ‘<product>’ to ‘<company>\<product>’.

In any case, the virus attempts to open the global configuration file. If the open fails, it will simply skip altering the configuration file. Otherwise, it will parse the file in order to register its script.

When a file cannot be opened, the returned file handle is -1. Viruses typically check for this value indirectly, by incrementing the returned value and checking for zero. This behaviour is very common because it is smaller than checking for -1. In the event that the file open succeeds, viruses will usually decrement the file handle again to restore it to its original value. However, this virus does not restore the value. Instead, it uses the misaligned file handle as though it were a regular value.

Windows accepts this as though it were the regular value and still behaves correctly. This could interfere with some behaviour-monitoring programs that watch for exact values being used to access files.

CFG PARSING

While parsing the configuration file, the first check the virus makes is the version of the cfg file. This check restricts support to versions 3.06 and 3.13 (even though 3.2 was available at the apparent time of writing the virus). While there were no significant changes to the format of the cfg in later versions of the program, the virus writer was presumably being careful to support only the version(s) that he had at the time.

The virus increments the number of registered scripts, and then checks if the first script in the list is marked to run on start-up. If it is not marked to run on start-up, then the virus assumes that its script has not been altered yet, and proceeds to make some changes. The virus changes the configuration file by inserting the name of the virus script, and marking the script to run on start-up. This is equivalent to an ‘autorun’ setting for the *010 Editor*.

The virus uses this behaviour to infect a file that is opened when the *010 Editor* starts. The script also runs whenever a file is opened after the *010 Editor* starts. For some reason, the *010 Editor* allows a registered script to have no display name. The virus makes use of this fact for some light stealth – since the virus script does not appear in the list of registered scripts, its execution potential is not obvious.

After altering the configuration file, the virus runs the host code.

THE SCRIPT

The *010 Editor* supports a C-like scripting language. The virus script begins by querying the filesize of the currently opened file (if any). It also creates a string that holds the virus body in encoded form. This is the reason for using an encoding method that avoids zeroes: if the string contained an embedded zero, then it would appear to be shorter than its actual length, because the first zero would be considered the sentinel character for the string.

The string is also optimized for size. The non-printable characters are escaped, but the leading zeroes are omitted. The printable characters are interspersed with the non-printable ones, and the base64-encoded body follows immediately. The script is written in such a way that it contains no space characters at all, and all of the statements are combined onto a single line. Further, the script makes very heavy use of the order of operations to allow a number of parentheses to be omitted. This makes it very difficult to read, and such a style would deserve a fail in a computer science class. Perhaps the virus writer intends to submit a future work to an obfuscated ‘C’ contest.

Amazingly, the script does contain strict bounds-checking to prevent the virus from attempting to read beyond the end of the file. The virus also uses a very nice trick while validating the file format. There is no function to change the file attributes and re-open the file, so the virus cannot infect read-only files. However, instead of performing an isolated check for a file having the read-only attribute set, the virus uses the return value of the GetReadOnly() function as the file offset for reading the file header. If the GetReadOnly() function indicates that the file is read-only then the read offset will be non-zero, and the format signature will not be read correctly. As a result, a file which has the read-only attribute set will fail to validate as an infectable file.

The virus performs further validation by comparing a large set of ‘magic’ numbers, like so:

```
if(ReadShort(GetReadOnly())==0x5a4d&&ReadInt(d)==0
x4550&&e[4]==76&&e[5]==1&&e[22]&2&&(e[23]&49)==1&&
!e[93]&&(e[92]-2)<2&&!(e[95]&32)&&!ReadInt(d+152)&&g
+h==c)
```

EVIL ALIGNMENT

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 virus is interested in files that are *Windows* Portable Executable files, and that are character mode or GUI applications for the *Intel* 386+ CPU. The files must not be DLLs or system files or WDM drivers. They must have no digital certificates, and they must have no bytes outside of the image.

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 4KB to 6KB in addition to the size of the virus and the size of the file alignment. The data will exist outside of the image, and serves as an infection marker. The presence of the file alignment bytes is to avoid a bug in some of the virus writer's earliest viruses. The bug occurred in files with a file alignment value that was larger than the number of bytes that the virus would append without including the file alignment. In that case, the infected file would have a structure that would still appear to have no appended data according to the algorithm that the virus uses to detect it. (The virus determines the presence of appended data by summing the physical offset and size of the last section. This works in most cases, but is far from proper.). As a result, the file could be infected as many times as it would take for the last section size to exceed a multiple of the file alignment value. By including the file alignment in the calculation for the number of bytes to append, the infected file always has appended data after one pass.

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. It 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 base64 decoder overwrites the encoded data with the decoded data.

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 saves the original entrypoint within the virus body, then alters the host entrypoint to point to the last section. The virus zeroes the file checksum then saves the file. Finally, it closes the infected file to flush the data to disk. This has the effect of requiring the user to make a second request to open the file. This is required only if the file is newly infected. Files that cannot be infected (because they are infected already or are not suitable) will be opened on the first request.

CONCLUSION

This virus demonstrates the case of 'when tools attack'. We have seen viruses for *IDA* and *HIEW* that infect the file that is being examined. Fortunately, we have not yet seen a virus that can escape from the tool's environment and begin the infection on a clean machine – but it might be only a matter of time.