MALWARE ANALYSIS 2

IT'S MENTAL STATIC!

Peter Ferrie
Microsoft, USA

We have seen viruses with binary components, and viruses with script components, and viruses with binary components that drop script components. Now comes W32/Mikasa, a virus whose binary component executes its script component directly in memory by using a binary interface, instead of dropping the script component first.

RANDAMN

The first-generation code begins by constructing an initial 128-bit key for RC4 by calling the GetTickCount() API four times in a row, with no delay between each call. This is a poor way to seed a random-number generator, as any reasonably modern machine will return the same value each time. Also, depending on for how long the system has been running, the top few bits of the returned value might be zero.

The first-generation code uses the RC4 key-scheduling algorithm which, while correct, is very strange. Since the key is 16 bytes in length, a simple AND operation can be used to index the key array. Instead, the first-generation code uses a divide operation and extracts the remainder to use as the index. This may have been done to allow the key length to be changed without needing to change any of the code.

The random generation algorithm is also very strange, in the sense of being a very inefficient implementation. The same values are fetched multiple times instead of caching the results in registers. Perhaps someone was in a rush while writing the code. Fortunately for the virus writer, since this is all part of the dropper code, it doesn't matter how slow it is, but it is unusual to see both loose and tight code in the same module.

The first-generation code encrypts its body and converts the encrypted body and the RC4 key to a textual representation of decimal values. The decimal values are placed in respective arrays, and then a script is appended which will perform the decryption and re-encryption of the body. It is not known why the first generation even encrypts the body (that is, it could use a shorter and constant key), since the encrypted copy is never used. The virus will re-encrypt itself first and place that copy in infected files.

MIRROR MIRROR

The script itself is interesting. It uses a nice reflection trick to avoid having to carry a copy of its own source code: it

declares a single function that holds the entire script. All the script needs to do to access its own source is to refer to the function by name. The reference will cause the script source to be returned, and the source can be assigned to a variable and manipulated at will. The script implements RC4 but it seems to contain a typographical error, resulting in a key length of only 120 bits instead of the expected 128 bits.

The first-generation code converts the script to Unicode and saves it for later. The first-generation code also modifies two variables in the virus body using a constant. It is not known why the constants weren't used in the first place. Finally, the first-generation code pushes the original entry point onto the stack, and then the dropper code is reached.

The dropper registers a Structured Exception Handler in order to intercept any errors that occur during infection. The dropper retrieves the base address of kernel32.dll. It does this by walking the InMemoryOrderModuleList from the PEB_LDR_DATA structure in the Process Environment Block. The address of kernel32.dll is always the second entry in the list. The dropper assumes that the entry is valid and that a PE header is present there. This is fine, though, because of the Structured Exception Handler that the dropper has registered.

STACKING THE DECK

The dropper resolves the addresses of the API functions that it requires: find, set attributes, open, map, unmap, close, malloc, free, write and LoadLibrary. The dropper uses hashes instead of names and uses a reverse polynomial to calculate the hash. Since the hashes are sorted alphabetically according to the strings they represent, 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 hash table is terminated with a single byte whose value is zero. While this saves three bytes of data, it also prevents the use of any API whose hash ends with that value. This is obviously not a problem for the virus in its current form, since none of the needed APIs have such a value, but it could cause some surprises for any virus writer who tries to extend the code.

The dropper allocates some memory for the file header of the dropped file, and then unpacks the header using a value-offset pair. Since the header will be written to a buffer that is known to contain all zeroes, there is no need to store the zeroes again. Instead, the dropper specifies the offsets of only the non-zero bytes, and the value of each of those non-zero bytes. The header is constant, and contains only one section. The section characteristics specify that the section is writable and executable. Even though the section does not have the readable flag set, it is still readable because the writable flag is set. The virus code is appended to the header, and the code is marked as 'dropped', which changes the code path that is executed later. The file is created using the name 'hh86.exe', a reference to the virus author.

The file creation method is also interesting. Instead of using the traditional GENERIC_READ and GENERIC_WRITE flags, which, as the names imply, are used to cover any object type, and which are used by probably just about everyone else, the dropper uses the file-specific flags instead: FILE_READ_DATA and FILE_WRITE_DATA. These flags have much smaller values than the GENERIC equivalents, allowing the virus writer to save several bytes of code. It also obscures to a slight degree the requested access rights, for those people who are unfamiliar with the file-specific flags.

After the content is written, the dropped file is closed and then executed. The dropper stage ends by freeing the allocated memory, and then forces an exception to occur. The exception will be intercepted by the exception handler, which will unregister itself and then transfer control to the host. This technique appears a number of times in the code and is an elegant way to reduce the code size, in addition to functioning as an effective anti-debugging method.

WORKING FROM A SCRIPT

The virus begins by saving the process image base on the stack, and then adding the original entry point RVA to that value. This makes the virus compatible with Address Space Layout Randomization. However, there is a bug in this behaviour (detailed below), regarding the value of the entry point RVA that is used. From here, the virus behaves like the dropper up to the point where the kernel32 API resolution is complete. At that point, the virus loads ole32.dll, and resolves the CoCreateInstance(), CoInitialize() and CoUninitialize() API addresses. The virus initializes the ScriptControl object as in-proc server, and queries the interface for the entry point of the IScriptControl object. The virus sets the scripting language to 'JScript', and then runs the script to produce the decrypted body and a new encrypted copy. The results from the script are returned to the virus as a BSTR object.

At no time is the script written to disk, thus it would evade traditional script-scanning technologies. However, any script scanner that hooks into the scripting interface itself (for example, by replacing the name of the scripting DLL in the registry with the script-scanning DLL, and exposing the identical interface) would have a chance to examine the script before it executes.

The virus registers another Structured Exception Handler, decodes the BSTR object to executable code and then executes it. The decoder is another strange routine – there are simpler ways to do it, but this one works well enough for the purpose.

SEEK AND DESTROY

The virus searches for all objects in the current directory (only). Yet more strangeness exists here, in that the virus writer has reverted to ANSI APIs for file handling. The result is that some files cannot be opened because of the characters in their names, and thus cannot be infected. However, the virus does attempt to remove the read-only attribute from whatever is found. It attempts to open the found object and map a view of it. If the object is a directory, then this action will fail and the map pointer will be null. Any attempt to inspect such an object will cause an exception to occur, which the virus will intercept. If the map can be created, then the virus will inspect the file for its ability to be infected.

The virus is interested in Portable Executable files for the *Intel* x86 platform that are not DLLs or system files. The check for system files could serve as a light inoculation method, since *Windows* ignores this flag. The virus checks the COFF magic number, which is unusual, but correct. The reason for checking the value of the COFF magic number is to be sure that the file is a 32-bit image. This is the safest way to determine that fact because, apart from the IMAGE_FILE_EXECUTABLE_IMAGE and IMAGE_FILE_DLL flags in the Characteristics field, all of the other flags are ignored by *Windows*. This includes the flag (IMAGE_FILE_32BIT_MACHINE) that specifies that the file is for 32-bit systems. As an added precaution, the virus checks for the size of the optional header being the standard value. The virus also requires that the file has no Load Configuration Table, because the table includes the SafeSEH structures, which will prevent it from using arbitrary exceptions to transfer control to other locations within its body. The last two checks that the virus performs are that the file targets the GUI subsystem, and that it has a Base Relocation Table which begins at exactly the start of the last section, and which is at least as large as the virus body.

TOUCH AND GO

The virus overwrites the relocation table with the dropper code and the script, changes the section characteristics to writable and executable, and sets the host entry point to point directly to the dropper code. It then marks the file as a dropper in order to complete the cycle. The virus clears only two flags in the DLL Characteristics field: IMAGE_DLLCHARACTERISTICS_FORCE_INTEGRITY and IMAGE_DLLCHARACTERISTICS_NO_SEH. This allows signed files to be altered without triggering an error, and enables Structured Exception Handling. The virus also zeroes the Base Relocation Table data directory entry. This is intended to disable Address Space Layout Randomization (ASLR) for the host, but it also serves as the infection marker. Unfortunately for the virus writer, it has no effect at all against ASLR. The reason is that ASLR does not require relocation data for a process to be 'relocated'. If the file specifies that it supports ASLR, then it will always be loaded to a random address. The only difference between the presence and absence of relocation data is that without it, no content in the process will be altered. *Windows* assumes that if the process specifies that it supports ASLR, then it really does support ASLR, no matter what the structure of the file looks like. The result is that a process that has had a relocation table overwritten by the virus will crash when it attempts to access its variables using the original unrelocated addresses. Alternatively, if the platform does not support ASLR (i.e. *Windows XP* and earlier), and if something else is already present at the host load address (or if the load address is intentionally invalid to force the use of the relocation table), then the file will no longer load. After the infection is complete, the virus unmaps the view and then closes the handle.

After all files have been examined, the virus intends to free resources and uninitialize COM but there is a bug in this code. The bug is that the stack is unbalanced because of a missing POP instruction, resulting in the virus crashing instead, and being terminated silently by *Windows*. Of course, since this is the dropped file, the process termination was expected anyway, so this is probably the reason why the bug was not noticed. However, there is another bug in the code, which is that if the uninitialization phase does complete successfully, the virus forces an exception to occur, to transfer control to the exception handler. The exception handler unregisters itself, and then transfers control to the entry point that was current *for the infected file*. This can have completely unpredictable effects.

CONCLUSION

The technique of executing a script component from within a binary component introduces a complication for anti-malware engines, where the respective scanning engines are generally completely distinct. One way to tackle the problem could be to treat the binary component in a manner similar to an HTML page which holds the script. However, there is the added complexity in the binary case of potentially needing to emulate the code in the binary component first, in order to expose the script. We live in interesting times.