MALWARE ANALYSIS 1

A(C)ES HIGH

Peter Ferrie
Microsoft, USA

Intel introduced a new set of CPUs in 2009 that included hardware support for the Advanced Encryption Standard (AES) in the instruction set. Such CPUs were not in wide circulation until relatively recently, and as a result, they did not attract much interest from virus writers. All that has changed, however, with the release of the W32/Brotinn virus.

REGISTER NOW

The first generation of the virus begins by registering a Structured Exception Handler to intercept any errors that occur, but the entry point of the host is used as the handler procedure. This means that if an exception occurs, the Structured Exception Handler structure remains registered, which can lead to unexpected behaviour if the host expects the initial Structured Exception Handler address to be in the kernel. The virus 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 virus resolves the addresses of the API functions that it requires, which is just a bit more than the bare minimum that it needs for infection: find first/next, set attributes, open, map, unmap, close and GetTickCount (which is used for seeding the random number generator). There is an interesting advancement here: the APIs in this virus are Unicode-only, instead of ANSI-only, as before. This means that the virus can infect any files that can be opened.

The virus uses hashes instead of names, but the hashes are sorted alphabetically according to the strings they represent. The virus uses a reverse polynomial to calculate the hash (that magical '0xEDB88320' value). Since the hashes are sorted alphabetically, 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 does not check that the exports exist, relying instead on the Structured Exception Handler to deal with any problems that occur. Of course, the required APIs should always be present in the kernel, so no errors should occur anyway. The hash table is not terminated explicitly. Instead, the virus checks the low byte of each hash that has been calculated, and exits when a particular value is seen. The assumption is that each hash is unique and thus when a particular value (which corresponds to the last entry in the list) is seen, the list has ended. While this is true in the case of this virus, it might result in unexpected behaviour if other APIs are added, for which the low byte happens to match.

'RANDOM' NUMBER GENERATION

The virus calls the GetTickCount() function 16 times in a row, to initialize the state for the random number generator. This is a very poor way to seed the generator, given that if the host machine is reasonably modern, all of the numbers will be the same. The random number generator is WELL512 (see below), but the code is not a direct translation from the available C code. Instead, the constants have been adjusted to integrate the effect of some of the shifts. It is not known why this was done – the implementation is larger than can be achieved simply by translating the original version. One reason might be to hide the constants to make the algorithm more difficult to identify, but that seems pointless for a random number generator. It is far more common to do that for an encryption algorithm. It is also not known whether it was the virus author who produced this version of the code.

WELL, WELL, WELL

WELL512 is the 'Well Equidistributed Long-period Linear' random number generator – a smaller and faster random number generator than the Mersenne Twister, and written nearly ten years later. One implementation of WELL has an equal length of period to the Mersenne Twister, and the two generators even share a co-author. The main difference between WELL and the Mersenne Twister is that WELL improves on the Mersenne Twister by producing a 'better' distribution of values.

BITS AND PIECES

The virus searches in the current directory (only) for PE files, regardless of their extension. It uses a nice trick to find the files, that was first seen in the Chiton [1] family: the file mask is '*' which, when pushed onto the stack, can be interpreted as a zero-terminated Unicode string because it is followed by three zeroes. Another 'advancement' is that the virus attempts to remove the read-only attribute from whatever is found. This is in contrast to all of the previous viruses by the same author, which could not infect files if the read-only attribute was set. The virus 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 the virus 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.

ADVANCED ENCRYPTION SOMETIMES

The virus constructs a decryptor that uses the AES instruction set to decrypt itself. The decryptor begins by pushing the host entry point RVA onto the stack, and then calling the decryption routine. The virus does not check if the CPU supports the AES instructions because the assumption is that the exception handler that the virus registers will intercept any errors. There is a minor problem with this assumption, which is that the host file has been altered irrevocably at this point – the relocation table data has partially been overwritten. However, the file will not be considered by *Windows* to be corrupted – it will continue to load as before, but it might not run properly (see below). The file will also remain a candidate for infection if it is transferred to a platform where the AES instructions are supported. The first generation of the virus carries the value '1986' repeatedly in the place where the AES key will be stored. This seems likely to be a reference to the virus writer's handle, and might be her birth year.

The virus calculates four random numbers and stores them in the virus body (technically, they are four parts of a single 16-byte number). The virus also pushes the last of those four numbers onto the stack. There is a funny piece of code at this point – the virus explicitly caches the stack pointer in a register and then uses the 'enter' instruction, which implicitly caches the stack pointer in the ebp register. Later, the original stack pointer is restored from the cache register. This is particularly surprising, given that the 'leave' instruction exists for that exact purpose, and the virus author has made correct use of the 'enter' instruction in another virus [2].

The virus copies the 16-byte number onto the stack. It calls a routine which executes the 'aeskeygenassist' instruction on the 16-byte number and then performs various bit-shufflings and XORs on the result, which is also saved on the stack for use later. This routine is called separately once, and then seven more times in a loop. The round constant begins with 1, and is shifted once per iteration of the loop. It is not known why the virus author didn't rotate the value instead of shifting it, which would have allowed the routine to be called eight times in a loop, thus avoiding the initial call. The routine is then called twice more, once with a round constant of 27 and once with a round constant of 54. It is also not known why these numbers were chosen.

The result is a total of 11 random numbers, each of which is 16 bytes in length. The first random number is XORed against 16 bytes of the virus body, and then the next ten random numbers are used to encrypt the result. There is an additional encryption operation using two registers that are not initialized and have no effect on the result. This do-nothing operation is actually a place holder for a decryption operation in the decryptor. The instruction is replaced with an 'aesimc' instruction in the decryptor, which is used to prepare the round key for decryption. It is also interesting to note that the act of single-stepping through the code using the *WinDbg* debugger causes all of the XMM registers to be zeroed, completely breaking the encryption operations. Finally, the values on the stack are discarded by restoring the stack pointer using the cache register. The stack is ultimately balanced by discarding the random number that was pushed originally, and which served no purpose at all.

TOUCH AND GO

The virus overwrites the relocation table with the encrypted virus body, changes the section characteristics to writable and executable, and sets the host entry point to point directly to the virus code. 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 probably intended to disable Address Space Layout Randomization (ASLR) for the host, but it also serves as the infection marker. Unfortunately for the virus writer, this has no effect at all against ASLR. The 'problem' 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 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 attempts to close the handle, but there is a bug at this point (technically, there are two bugs of the same kind): the wrong offset is used when indexing into the structure that holds the API addresses. Instead of calling the CloseHandle() function, the virus calls the UnmapViewOfFile() function again. Twice, in fact: once for each of the handles that is supposed to be closed. As a result, the virus leaks one handle for every file that is opened successfully, and an additional handle for every file that is mapped successfully.

DECRYPTOR

The decryptor begins by caching the stack pointer in a register and then rounding down the stack pointer to the nearest multiple of 64KB, before saving the extended floating point state onto the stack. The reason for aligning the stack pointer is that the ‘fxsave’ instruction requires that the destination address is aligned to a multiple of 16 bytes. However, the state is 512 bytes long, so the virus makes sure that saving the state will not corrupt the variables that are already on the stack. The virus decrypts itself by executing the same encryption instructions as before, but in reverse order. After replication is complete, the virus restores the extended floating point state and returns the stack pointer to its original value before passing control to the host.

CONCLUSION

The use of the AES instruction set will certainly challenge CPU emulators in the short term, but the lack of a preceding CPUID check for its support might be considered quite suspicious for now (in the same way that almost no one checks for the presence of support for the MMX instruction set before attempting to use it). For now, at least, time is on our side.

REFERENCES

[1] http://www.virusbtn.com/pdf/magazine/2002/200206.pdf.

[2] http://www.virusbtn.com/pdf/magazine/2012/201206.pdf.