MALWARE ANALYSIS

IF SVAR IS THE ANSWER...

Peter Ferrie
Microsoft, USA

The *Intel* MMX instruction set is full of instructions whose usefulness might not be immediately clear to anyone who does not work with graphics. However, it's not just the graphic designers who can do interesting things with them. Virus writers are finding ways to (ab)use some of the instructions, too. This time, we have W32/Svar, and another way to encode.

IN THE BEGINNING

The first generation of the virus begins by saving the relative address of the original entrypoint on the stack. Unlike in W64/Svafa[1], this value is always correct. The virus applies the current imagebase value from the ImageBaseAddress field in the Process Environment Block, which would normally be required to account for Address Space Layout Randomization (ASLR). However, the virus disables ASLR for the file during infection, so the technique is not required.

The virus continues by setting up a Structured Exception Handler in order to intercept any errors that occur during infection. 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. This is compatible with the changes that were made in *Windows 7*. The address of kernel32.dll is always the second entry in the list. The virus 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 virus has registered.

The virus resolves the addresses of the API functions that it requires, which is the bare minimum set of APIs that it needs for replication – find first/next, open, map, unmap, close. 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 also checks that the exports exist by limiting the parsing to the number of exports in the table. The hash table is terminated with a single byte whose value is 0x2a (the '*' character). This is a convenience that allows the file mask to follow immediately in the form of '*.exe', however it does prevent the use of any API whose hash ends with that value.

As with previous viruses by the same author, 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 objects whose names end in '.exe'. The search is intended to be restricted to files, but can also include any directories that have such a name, and there is no filtering to distinguish between the two cases. For each such file that is found, the virus attempts to open it and map an enlarged view of the contents. There is no attempt to remove the read-only attribute, so files that have the read-only attribute set cannot be infected. In the case of a directory, the open will fail, and the map will be empty. The map size is equal to the file size plus 8KB, to allow the file to be infected immediately if it is acceptable. This 8KB value is hard-coded in the virus, which could interfere with variants being made based on it, and which could lead to a crash during decryption. Using the post-infection size during the validation stage allows the virus to avoid having to close the file and re-open it with a larger map later. The virus assumes that the handle can be used, and then checks whether the file can be infected.

BITS AND PIECES

The virus is interested in Portable Executable files for the *Intel x86* platform that have no appended data. Renamed DLL files are not excluded, nor are files that are digitally signed (at least, not explicitly – most of them will be filtered implicitly, because it is common for the signature to be placed after the end of the last section, but this is not a requirement). The subsystem value is checked, but this is done 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.

The section attributes are marked as executable and writable. The virus encodes its body using a bit-mask technique. There is only one table involved this time, which is eight times the size of the virus code. The table contains the bit-mask. Each byte of the host is split into eight bits, and each bit is stored individually in the table after combining it with seven bits that are set to a random value. This process is repeated over the entire host body. Interestingly, most of the code is optimized for small size, but the encoding routine is not optimized at all. Instead of simply rotating the bit into the top of the random value in order to combine it, the virus performs the equivalent of an 'if...then...else' for each bit in the code, and ORs or ANDs the value as appropriate. Once the encoding is complete, the virus stores four bytes of zero, which are intended to mark the end of the virus body, but there are two problems with this. The first problem is that there is a small, but real chance that if the top four bits were zero in any byte in the virus code, and if the random number generator happened to return a zero in the low byte four times in a row, then the output would match exactly what the

[1] http://www.virusbtn.com/pdf/magazine/2012/201201.pdf.

virus uses to mark the end of the virus body. In that case, the decoder would stop too soon, and the virus would crash.

The virus zeroes the RVA of the Load Configuration Table in the data directory. This has the effect of disabling SafeSEH, but it affects the per-process GlobalFlags settings, among other things. The virus also zeroes the DLLCharacteristics field. This has the effect of disabling ASLR and 'No eXecute' for the process (allowing the virus to run in a section that does not have the executable flag set, but the virus sets it explicitly anyway, as noted above), and enabling Structured Exception Handling. The virus saves the original entrypoint in the virus body, and then sets the host entrypoint to point directly to the virus code.

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. It also does not restore the file's date and timestamps, making it very easy to see which files have been infected.

When an infected file is executed, the virus decodes itself. The decoding is done using two MMX instructions, one of which might be considered to be a bit obscure: PMOVMASKB. The PMOVMASKB instruction accepts two parameters which correspond to the table that the virus constructed, and the register to receive the result. The instruction works with eight bytes at a time, and combines a single bit from each byte into a single byte which the virus stores. The result is the decoded host byte. The decoder does not use a register to hold the number of bytes to decode. Instead, it checks if four bytes of zero were read at a particular alignment. However, there is a bug in this check, and this is the second problem: the alignment is incorrect for the purpose. As a result, the decoder interprets its own code as though it were also encoded, and 'decodes' this, too. Fortunately for the virus, this action is harmless because the table is so large that the decoder cannot be overwritten. However, there is a small potential problem which stems from the hard-coded 8KB value noted above: if the table and the decoder happened to end at exactly a multiple of 8KB, then the decoder bug would cause the decoder to access memory beyond the end of the image and crash.

CONCLUSION

The PMOVMASKB technique is yet another surprise from the MMX instruction set, but the entire body is encoded so it does not look like corrupted code. However, anti-malware emulators will have no problem emulating through the code and won't appreciate the technique.