MALWARE ANALYSIS 1

NOTEVEN CLOSE

Peter Ferrie
Microsoft, USA

Code virtualization is a popular technique for making programs difficult to reverse engineer and analyse. While its use is seen mainly in commercial products such as *VMProtect*, some viruses use the technique for the same purpose. The interpreter is the weak point in any virtualization implementation, because if the interpreter can be understood, then the virtualization can be reversed to some degree. In some cases, the interpreter is intentionally made difficult to read. In the case of the W32/Noteven virus, the interpreter is difficult to read due to sloppy coding and numerous bugs. We can safely assume that this is unintentional.

UNSTRUCTURED EXCEPTIONS

The interpreter begins by installing a Structured Exception Handler to point to a location within the interpreter. There is a bug in this code, which is that the virus breaks the SEH chain, so any code that needs to walk the chain will fail and possibly cause an exception that will result in the program being terminated. The interpreter retrieves the address of a system DLL by walking the InInitializationOrderModuleList from the PEB_LDR_DATA structure in the Process Environment Block. This results in the interpreter retrieving the address of kernelbase.dll on *Windows 7* and later versions, but kernel32.dll on earlier versions of *Windows*. The interpreter resolves the addresses of the functions that it needs in order to infect files (FindFirstFile, FindNextFile, CreateFile, GetFileSize, ReadFile, WriteFile, CloseHandle, VirtualAlloc, VirtualProtect and GetCurrentDirectory [which is unused]). Fortunately for the virus author, all of these functions are present in kernelbase.dll.

The interpreter allocates space on the host stack for the encoded instructions. There is a minor bug in this code, which is that depending on the size of the instructions, the resulting stack pointer value might be misaligned. Fortunately for the virus author, the interpreter uses no APIs that require the stack to be aligned (such as FindFirstFile – which is called by the virtualized code, and the interpreter merely resolves it on behalf of the virtualized code). The interpreter copies the array of instruction lengths to the host stack for processing later, then allocates two blocks of memory. The first block is the stack for the virtualized code, and the second block is the virtual machine buffer that will hold the virtualized code. The interpreter then copies the virtualized code to the virtual machine buffer, and unmaps the virtual machine buffer in order to ‘protect’ it from being read externally (though nothing stops the original copy from being read instead).

The interpreter swaps to the virtual machine stack, saves the CPU flags there, allocates space for the virtual machine registers, and then swaps back to the host stack. The interpreter copies the current register values to the virtual machine stack, and then begins parsing the virtualized instructions using the instruction lengths that were copied to the host stack earlier. The parsing is performed in the reverse direction, for no obvious reason. The interpreter maps the virtual machine buffer, copies the virtualized instruction from the virtual machine buffer to the virtual machine stack, according to instruction length, decodes the instruction, and then unmaps the virtual machine buffer again.

WHAT’S IN A NAME?

The decoded instruction is the original native CPU instruction. The virus makes no attempt to transform the opcodes in any way. Each instruction is examined before execution, because the interpreter will perform a controlled execution of the ‘safe’ instructions. Certain instructions cannot be executed directly because they will cause a loss of control and possibly a crash. This means that the environment is not code virtualization in the typical sense, but rather buffered code execution. The difference is not important for the purpose of this article.

The instructions that are considered to be special by the interpreter are: E8 (call rel32), C3 (ret), FF (various), EB (jmp rel8), E9 (jmp rel32), 0F 80-8F (jcc rel32), 70-7F (jcc rel8) and E0-EF. There is a bug in the last set, which is that it should be restricted to E0-E3, but the way in which the comparison is made throws away the entire low nibble instead of just the low two bits. This prevents the interpreter from supporting the missing instructions and can result in unexpected behaviour.

If the instruction is not considered to be special, then the interpreter appends a call/fault/jmp sequence before allowing the instruction to run. The call is used to save the instruction pointer onto the stack. The faulting instruction is used to transfer control back to the interpreter. The jump is supposed to transfer control back to the interpreter in the event that somehow the faulting instruction did not do so, but the jump offset is completely wrong, so if the jump were ever hit, then the virus would crash. The interpreter intercepts the exception and ensures that it is the expected kind. The faulting instruction is an interrupt 3 instruction, which can interfere with a debugger and make the code difficult to trace. The context that is saved when an exception occurs will be used to update the registers in the virtual machine.

In order to execute an instruction, the interpreter saves the host registers on the host stack, swaps to the virtual machine stack, restores the virtual machine registers from the virtual machine stack, and then runs the instruction. When the interpreter intercepts the exception, it reinstalls the broken Structured Exception Handler vector and installs another Structured Exception Handler to point to a location within the interpreter. It also saves the virtual machine registers on the virtual machine stack, swaps to the host stack, and then restores the host registers from the host stack.

CALL ME CRAZY

If the instruction opcode is 'E8', then the interpreter checks if the relative offset is within the limit of the virtual machine buffer. There are two bugs in this routine. The first is an off-by-one boundary condition, which allows the call to land on the first byte beyond the end of the buffer. The second bug is an off-by-n boundary condition, which does not require that the next instruction to execute fits in the space remaining in the buffer. The interpreter also implicitly limits the size of the virtualized code, so anything outside of the buffer is treated as though the relative offset were zero for the call. If the instruction is accepted, then the interpreter inserts space for a dword on the virtual machine stack. There is a bug in that routine, too, which is that there is no check against the stack limits. As a result, a stack fault will occur when the stack becomes full. Otherwise, the interpreter places the return address in the newly created space, updates the stack pointer, and determines the new instruction pointer location. There is yet another bug in this code, which is that a missing instruction prevents the interpreter from supporting calling backwards in the code. Not only is it not supported, it is also misinterpreted – any attempt to call backwards will be treated as a call forwards by the absolute relative value (that is, a call backwards by six bytes will become a call forwards by six bytes).

POINT OF NO RETURN

If the instruction opcode is 'C3' then the interpreter fetches the return address from the virtual machine stack, deletes the element from the virtual machine stack, and adjusts its position in the length array corresponding to the return address. There is a bug in this code, which is that the interpreter assumes that it will be returning to an earlier position in the array. Any attempt to return to a later position will cause the virus to crash.

If the instruction opcode is 'FF', then the interpreter checks if the eax register contains certain values. One value retrieves the in-memory address of the interpreter. Another causes a return of control to the host entry point, however there is a bug in the state that is passed to the host. Another value retrieves a pointer to the list of resolved API addresses. If the value is none of these, then the interpreter attempts to run the instruction as described above. There is a bug in this behaviour, which can result in escape from the virtual environment, because the interpreter does not prevent the interpreted code from jumping to an arbitrary address.

If the instruction opcode is 'EB', then there is a bug: the interpreter forces an exception to occur, perhaps for debugging purposes, but since the wrong stack is in use at that moment, the result is that another exception occurs. The second exception is intercepted by the interpreter, but during the handling, another exception occurs. This cycle repeats until a stack fault causes *Windows* to terminate the program.

If the instruction opcode is 'E9', the interpreter calculates the new instruction pointer and continues execution.

If the instruction opcode is '0F 80-8F' or '70-7F', then the interpreter attempts to simulate the branch. The way in which this is done is about as non-optimized as it is possible to be. Instead of simply loading the flags and then replicating the branch instruction locally, the virus examines the flags according to the conditions that they are supposed to represent, and then branches to a unique label for each of the true and false conditions. Of course, there is also a bug in this code. The 'jbe' simulation has its conditions reversed, so a branch is taken when it shouldn't be.

If the instruction opcode is 'E0-EF' then we encounter more bugs. The first is that the handling for the 'E0-E2' set (LOOPNE, LOOPE and LOOP) skips the instruction if the ecx register is zero. A real CPU will perform the loop as though it had an initial iteration count of 4GB. That is, the value of zero in the ecx register is not special. Only a value of one can cause a loop to exit immediately, and the value in the ecx register is altered in all cases. The routine is broken anyway, because it forces an exception to occur, and demonstrates the recursive exception problem described above.

The interpreter also attempts to intercept faults in the virtual machine buffer, but there is a bug in this code: the interpreter uses the wrong offset for fetching the faulting address from the Exception Record. This results in a crash in the interpreter while it is searching the length buffer for the exception address. This bug also demonstrates the recursive exception problem described above.

VIRTUAL MALWARE

The virus that the interpreter runs begins by requesting the API addresses from the interpreter. The virus copies them to a local buffer, and caches the FindFirstFile, FindNextFile, and CreateFile addresses in registers, but the values are lost during a memory allocation that follows immediately. The caching might have been for debugging purposes to see the values, because the addresses are loaded again later, as needed. The virus enumerates the objects in the current directory, but discards the first two results, and begins examining the objects from the third one that is found. There is a minor bug in this idea, which is that the virus might miss some files whose names cause the directory sorting to place them before the '.' and '..' directories. The virus attempts to check that the extension is '.exe', but forgets to check for the '.', so a file named 'exe' is also accepted.

The virus attempts to open the file, allocate memory to hold the entire file plus another 4KB, read the whole file, and then close it. The only operation whose result is checked is the file open. The virus assumes that the other operations will succeed, or that the exception handler in the interpreter will intercept any problem (but, as we have seen above, this is not the case). There is another bug in this code, which is that neither the interpreter nor the virus frees any of the memory that they allocate. If enough large files exist in the directory, then eventually the process will be terminated by *Windows* due to memory exhaustion. The virus skips the file if it is infected already. The infection marker is the value 'EEEE' stored in the OEM ID space in the MZ header.

The infection routine contains numerous bugs, the most obvious of which is that the virus does not check the Machine field when infecting files, so 64-bit files will be corrupted. The virus does not check other important fields either, so DLLs and drivers will be infected, too. The virus marks the last section as writable but not executable, so it will fail to run on DEP systems if the last section was not already marked as executable. The virus saves the absolute address of the host entry point in the virus body, so the virus will fail to run the host if the host supports ASLR.

The virus fetches the Load Configuration Table data directory information, and assumes that if it is present, then it is located in the first section. This can result in a pointer to an unexpected location in the file. The virus attempts to zero out the actual table, instead of the data directory entry. The virus appends itself to the last section and changes the host entry point to point to the interpreter code. It then attempts to open the file, write itself, and close the file again. The virus does not check that the file attributes allow the file to be written to, and it does not do anything with the file's date and time stamps or the checksum. Files with appended data will have their appended data overwritten by the virus code. Once the virus has finished enumerating the files, it returns control to the host.

CONCLUSION

Analysing virtualized code can leave us wondering how it works. In this case, we're left wondering how it works, given how buggy it is.