

VIRUS ANALYSIS 1

Coping with Cabanas

Péter Ször
Data Fellows

1997 seems to be the year of the *Windows 95* virus breakthrough – although, so far, all Win32 viruses have been different in implementation. Virus writers have tried to attack the system in various ways – with some viruses using PE (Portable Executable) infection, or going resident as VxDs (Virtual Device Drivers), while others were able to infect DOS programs too. No Win32 virus to date worked under *NT*. Win32.Cabanas, recently received from its author, changes that.

Cabanas is a per-process resident, anti-debugging, partially packed/encrypted, anti-heuristic, semi-stealth virus. Its author, who also wrote the infamous WM/CAP virus, is a member of the 29A group. Cabanas, like CAP, uses a novel infection mechanism and has what was, initially, a barely comprehensible structure. Usually it is relatively easy to analyse a virus with a disassembler, some DOS viruses taking only a few hours.

This was not the case with Cabanas; it took days to analyse, consuming lots of energy and good hacking utilities. For instance, Cabanas cannot be traced in application level debuggers such as *TD32*; it requires *SoftIce* – ‘a debugger on steroids’. This is because typical debuggers focus on the address space of the program and cannot go to the operating system level. Cabanas cannot be loaded into a normal disassembler without it modifying the characteristics of the virus’ code section. This trick itself was enough to cause the first headache in my analysis.

Incompatibilities with Windows NT

I have to say that I have a different view of *NT*, in terms of security, now that I understand Cabanas. I drew incorrect conclusions from tests of the first *Windows 95* virus, Boza (see *VB*, February 1996, p.15). Most anti-virus researchers tested Boza on *Windows NT*, which, reassuringly, did not even try to execute the infected image. This led some to



think that *NT* had superior virus detection or prevention properties. I

patched some Boza-infected files to find out why. The PE format was designed by *Microsoft* for use in all its Win32 systems (*Windows NT*, *Windows 95* and Win32s). However, the implementation of the loader is different from system to system. The *NT* loader simply checks a few more things in PE files than the *Windows 95* one, thus finding Boza-infected files ‘suspicious’. One field in the header of Boza’s

.vld section is not correctly calculated by its infection routine, but if this was fixed Boza-infected PE files should be able to run under *NT*. However, even if Boza did not have this problem, it would still not be able to replicate under *NT*.

Every *Windows 95* virus has to call two Win32 KERNEL APIs – *GetModuleHandleA* and *GetProcAddress*. Since these are in *KERNEL32.DLL*, it is possible for *Windows 95* viruses to get those functions directly, with a hack. Most *Windows 95* viruses so far have hard-coded pointers to these APIs.

When the linker creates an executable, it assumes that the file will be memory-mapped to a specific location. In the Image File Header of PE files, there is a field called Image Base which holds this address. For executables, this address defaults to 0x400000. *Windows 95*’s *KERNEL32.DLL* has an Image Base address of 0xBFF70000. The two required API addresses will be at fixed offsets from *KERNEL32*’s base address in the same release of *Windows 95*. However, these offsets can be different in other releases, making viruses using these fixed addresses not compatible across *Windows 95* systems. In *NT*, the *KERNEL32* Image Base address defaults to 0x77F00000, therefore viruses with a *Windows 95*-specific base address *cannot* work under *NT*.

Furthermore, *NT* does not support VxDs. Thus, viruses like *Memorial* (see *VB*, September 1997, p.6) cannot operate under *NT*. They would have to include different infection algorithms for *Windows 95* and *NT* in order to succeed on both systems, making them unduly complicated.

If a Win32 virus could overcome these compatibility and implementation problems, it should be able to work equally well on both systems. Such viruses may even have Unicode support, but it would not be mandatory. Win32.Cabanas has all of these features!

The Role of the Import Table

Cabanas relies heavily on the Import Table. In Win32 environments, DLLs are linked through the PE file’s Import Table to the applications that use them. The Import Table holds the names of the imported DLLs and the names of the functions imported from them.

The executable code is located in the .text section of PE files (or in the CODE section, as the Borland linker calls it). When an application calls a function from a DLL, it does not call the DLL directly. Instead, the call goes to a JMP DWORD PTR <address> instruction in the executable’s .text section. The address is stored in the .idata section (or sometimes in .text). The JMP instruction transfers control to that target address. Thus, the DWORD in the .idata section contains the real address of the API’s entry point.

As all calls to a given DLL function are passed through one location, the loader need not patch every instruction. All the PE loader has to do is patch the correct address for each imported function into the list in the .idata section.

Running Infected PE Files

Execution of a Cabanas-infected file starts at the original entry point. Cabanas does not touch the entry point field in the Image File Header, patching the host program at its entry point instead. Leapfrog was the first DOS virus to use this trick. Five bytes at the entry point are replaced with a FAR JMP to the address where the original program ended. You may ask: 'But there can be relocations which may overwrite this location. How can the virus avoid this?' The answer is simple; Cabanas handles the relocation table too.

The first function in Cabanas simply unpacks and decrypts a table of Win32 KERNEL API names. The word 'File' is replaced in the names that would normally contain it. GetProcAddress is not packed at the beginning of the string table, but the next function name is 'encrypted' as 'Ge',t+80h,'AttributesA' or GetFileAttributesA when unpacked. Since Cabanas has Unicode support, the next string is GetFileAttributesW which is described in two bytes: 80h, SizeOfPreviousUnpackedString. The other strings are packed in the same way.

The real problem is that the virus uses Structured Exception Handling (SEH) as an anti-debug function. Not knowing the form of C++'s __try and _except functions in assembly, I ran into this trap several times before it dawned on me – the goal of this function is to set a new SEH FRAME and generate an exception. When execution reaches the instruction which caused the exception, control is redirected to the operating system's Exception Handler (EIP will point into the kernel). This is very annoying and needs *SoftIce* to trace. The operating system's exception handler sets the exception type and returns to the application. As a result, no general protection fault will be displayed and the SEH FRAME will be removed.

When the unpack/decryptor function is ready, the virus calls a routine to find KERNEL32's original Base Address. During infection, the virus searches the Import Table for GetModuleHandleA and GetModuleHandleW. When it finds them, it saves pointers to the DWORDs in the .idata list. If the application does not have either import, the virus uses another, unreliable way to get the address. This is probably the worst bug in the virus. (I should note here that in Win32 environments the Module Handle and Base Address are the same.) When the virus has the Base Address, it calls its own routine to get the function address of GetProcAddress. The first method is based on the search in the Import Table during infection time. In most cases, Win32 applications import the GetProcAddress API, thus the virus should not use a secondary routine to get the same result. If the first method fails, the virus searches for GetProcAddress and GetProcAddressFromExportsTable exports in KERNEL32's .edata section.

Cabanas searches for the GetProcAddress string in the Function Name Table of KERNEL32's Export Table. When it finds the correct string, it gets the entry point from the Function Address Table and returns. This function is one of the most important from the virus' point of view and is compatible with all Win32 systems.

If the entry point of GetProcAddress was returned by the GetProcAddressFromExportsTable function, the virus saves it to use later. If not, the function will be used several times, having been 'secured' with Structured Exception Handling to avoid possible exceptions. The virus can now get the addresses of all the Win32 APIs it needs to use. Cabanas is ready to replicate.

Direct Action Infection

The infection code is surprisingly fast, in spite of the fact it runs through all the files in the *Windows*, *Windows System* and current directories. This is because the virus uses memory-mapped files. The full process takes no more than a few seconds on a 486. First, the virus gets the name of the *Windows* directory with the GetSystemDirectoryA API, then searches it for non-infected executables. This uses the FindFirstFileA and FindNextFileA APIs, searching for non-directory entries and checking file sizes.

Those divisible by 101 are assumed infected. Those larger than 64MB are left alone. Targeted files are opened and mapped using the CreateFileA and CreateFileMappingA APIs. If a file is shorter than 128 bytes, it is closed and infection aborted. Cabanas checks for the 'MZ' marker at the beginning of the image, then repositions to the PE header area. It checks that the executable is for 386+ machines and looks for the internal file type, which must be an executable file, not a DLL.

Next, the virus calculates a special checksum using the checksum field of the PE file's Optional Header and the file-stamp field of the Image File Header. If the file seems to be infected, the virus closes it. If not, Cabanas saves the original file attributes, changing them so it can write to the file. It opens and maps the potential host in write mode and searches for the GetModuleHandleA, GetModuleHandleW and GetProcAddress API imports in the host's Import Table and calculates pointers to the .idata section. Then it calls a routine to patch the virus into the file.

This routine sets the MEM_WRITE flag of the .idata section if it is not already set, but only if this section is not located in an executable area. This means that there are some extreme cases when this table is part of the .text (CODE) section. The first five bytes at the host's entry point are replaced with a FAR JMP to the end of the host. The infection procedure checks for relocations that may overwrite this FAR JMP. If the relocation table size is non-zero, a routine searches the .reloc area. If a relocation points into the FAR JMP area, its relocation type is cleared so that it will not be used by the loader. This also marks the relocation so Cabanas will be able to find the host later.

A 'parameter block' of information needed to rebuild and start the host application is created, including the original five bytes from the host's entry point and its location. The self-recognition 'checksum' is calculated, as is the new file size. The size of the virus code is around 3000 bytes, but most infected files will grow by a little more than this because Cabanas pads itself to make the infected file size evenly divisible by 101.

The virus does not create a new section header for its code, but modifies the last section header in the file (usually .reloc) to be longer, 'making' enough space for the virus' code and setting the section's characteristics to include the MEM_WRITE flag. This makes infection less risky. The SizeOfImage field in the header is corrected and the file unmapped and closed. Finally, the file is truncated to the previously calculated size and the original time, date and file attributes reset.

Rebuilding the Host and Going Resident

Following the 'seek and infect' phase, Cabanas uses the GetCurrentProcess and WriteProcessMemory functions to write the original five bytes to the host's entry point. After this, it relocates the code area, if necessary, by searching the .reloc section for its specially marked entries. Now the virus goes resident, based on manipulation of the Import Table. With the addresses of imported functions in the host's .idata section, Cabanas need simply replace them with the addresses of its own API handlers.

To achieve this, Cabanas opens and maps the host. It allocates a 12232-byte block, copies itself there, then searches for the names of the functions it hooks: _lopen, CopyFileA, CopyFileW, CreateFileA, CreateFileW, CreateProcessA, CreateProcessW, FindClose, FindFirstFileA, FindFirstFileW, FindNextFileA, FindNextFileW, GetFileAttributesA, GetFileAttributesW, GetProcAddress, MoveFileA, MoveFileExA, MoveFileExW, MoveFileW, OpenFile, SetFileAttrA, and SetFileAttrW. Whenever it finds one, it saves the original address in its own JMP table and replaces the DWORD in the host's .idata section with a pointer to its own function. Finally, the virus closes and unmaps the host, then starts the application by jumping to the original entry point in the .text section.

Some *Windows* programmers may say: 'But this hook mechanism is not efficient enough. Whenever the application does not have imports for some of these APIs, but calls them directly by using GetProcAddress, the virus cannot hook anything other than the GetProcAddress API.' That is the reason that the virus hooks it.

When an infected program calls a Cabanas-hooked API, the virus' handler calls the original GetProcAddress for the address of the requested API. After this, it checks whether the function is a KERNEL32 API, and if it is one that it wants to hook. If so, and it is not yet hooked, the virus returns a new API address pointing into its jump table.

Stealth, *et al*

Cabanas implements semi-stealth: during FindFirstFileA, FindFirstFileW, FindNextFileA, FindNextFileW it checks for already infected programs. If a program is not infected, Cabanas infects it, otherwise it hides the change in file size. Thus, if a scanner checks its size by calling these APIs, it cannot detect the size change and will start scanning if no other checks are made. Anti-virus programs must have robust self-checks. One possible defence against Cabanas' stealth would be to compare the API addresses in your own Import Table with those in the KERNEL32 Export Table.

The virus can see all files accessed on an infected machine and since the *NT* command interpreter (CMD.EXE) uses the Win32 FindFirst/Next APIs during a DIR command, every non-infected file will be infected. The virus will also infect files during every other hooked API request.

Conclusion

Cabanas shows that a virus need not be *NT*-specific to work under *NT*. In fact, a working *NT* virus will more likely not have *Windows 95*-specific functionality. However, I expect virus writers will use knowledge they have gained from *Windows 95* to move to the more robust platform.

The author of Cabanas claims to have written a polymorphic engine, which was not included in this first version – 'One Half is not dead if you understand what I mean'. The next release of Cabanas could have a polymorphic decryptor in the .text section of the infected program. This will make the disinfection of such viruses very complicated in the future.

Win32.Cabanas

Aliases:	Cabanas.
Type:	Win32 (Windows NT, Windows 95, Win32s) PE infector. Per-process resident, semi-stealth, fast infector.
Self-recognition in Files:	Files with sizes divisible by 101 are assumed to be infected and a special checksum is stored in the file-stamp field of the PE Header (see text).
Self-recognition in Memory:	Not needed.
Hex Pattern in PE Files:	AB8B C6AB 6489 2360 8743 FE83 EF97 5857 57AC D2C0 34B5 7920
Intercepts:	Many Win32 kernel APIs – see text.
Payload:	None.
Removal:	Recover infected files from backup or replace with originals.