ATTACKS ON WIN32

Péter Ször

Data Fellows Ltd, PL 24, Fin-02231, Espoo, Finland.
Tel +358 9 859 900 • Fax +358 9 8599 0714 • Email Peter.Szor@datafellows.com

ABSTRACT

The world of computer anti-virus research has changed drastically since Windows 95 appeared on the market. One of the reasons why this happened was that a certain number of DOS viruses became incompatible with Windows 95. In particular, the tricky viruses which used stealth techniques and undocumented DOS features failed to replicate under the new system. Many simple viruses remained compatible with Windows 95, for instance Yankee Doodle – a very successful old Bulgarian virus. Regardless of this, virus writers felt the new challenge was to investigate the new operating system, to create new DOS executable viruses and boot viruses with special attention to Windows 95 compatibility. Since most virus writers did not have enough in depth knowledge of the internal mechanisms of Windows 95 they looked for shortcuts to enable them to write viruses for the new platform. They quickly found the first one – macro viruses, which are generally not dependent on the operating system or hardware differences.

Some young virus writers are still happy with macro viruses and develop them endlessly. However, after writing a few successful macro viruses they mostly get bored and stop developing them. Someone may think 'fortunately', but the facts are different. They are looking for other challenges and usually find new and much more difficult ways to infect the system.

The first Windows 95 virus, Boza, appeared in the same year that Windows 95 was introduced, written by a member of the Australian VLAD virus writing group. It took a long time for others to understand the insides of the system but, during, 1997 new Windows 95 viruses appeared, some of them in the wild.

At the end of 1997, the first Win32, Windows NT-compatible virus, Cabanas, had been written by the same young virus writer (Jacky/29A) who wrote the infamous WM/Cap.A virus. Cabanas is compatible with Windows 9x, Windows NT and Win32s (it is also compatible with Windows 98 and Windows NT 5.0, even though the virus code had never been tested on these systems by the virus writer since these systems appeared later than the actual virus). Cabanas turns Microsoft's Win32 compatibility dream into a nightmare.

Even if it is difficult to write such viruses, we suspect that file-infecting DOS viruses from the early years of computer viruses will be replaced eventually by Win32 creations. This paper describes all Windows 95 and Win32 attack methods known by the author and examines ways to prevent them.

1 INTRODUCTION: THE WIN32 API AND PLATFORMS THAT SUPPORT IT

Windows 95 was introduced by *Microsoft* as a new major operating system platform three years ago. While the system is strongly based on *Windows 3.x* and DOS technologies, it gives a real meaning to the term Win32. What is Win32? Originally, programmers did not even understand the difference between Win32 and *Windows NT*. Win32 is the name of an API (Application Program Interface), no more, no less. The set of system functions, available to call from a 32-bit *Windows* application, are contained in the Win32 API. The Win32 API is implemented on several platforms – one of them is *Windows NT*, the most important Win32 platform. *Windows NT* is capable of executing 16-bit *Windows* programs, OS/2 1.x character applications (and, with some extensions, even Presentation Manager-based 1.3 programs with some limitations) and DOS programs. In addition, *Windows NT* introduced a new executable file structure called Portable Executable (PE) file format (a file format which is very similar to, if not based on, the Unix COFF format) that can run Win32 applications (that call functions in the Win32 API set). As the 'Portable' prefix suggests, this format is supposed to be an easily portable file format, which is actually the most common and important one to run on *Windows NT*.

Other platforms are also capable of running Win32 applications. In fact, one of them was shipped before *Windows NT*. This platform is called Win32s. Everybody who has tried to develop software for Win32s knows that it was a very unstable solution. Since *Windows NT* is a robust system which needs strong hardware to run on, Win32 technology did not take the market position *Microsoft* wanted. That process ended up with the development of *Windows 95*. *Windows 95* supports the new PE format by default. Therefore it supports a special set of Win32 APIs. *Windows 95* is a much better implementation of the Win32 API than Win32s. However, *Windows 95* does not contain the full implementation of the Win32 APIs as found in *Windows NT*. *Windows 9*5 fills a very large and strategic marketing gap, serving users with old 386 (or better) machines with limited amounts of memory like 4MB (or more). The number of machines which fall into this category is still very significant. However, we should not forget that *Microsoft*'s real business dream is *Windows NT Server and Workstation* in the long run. Until *Windows NT* gains more momentum, *Windows 9x* is *Microsoft*'s Win32 platform.

Last but not least, the Win32 API and the PE format are supported by *Windows CE* which is used primarily by hand-held PCs. *Windows CE* is *Microsoft*'s most recent Win32 platform. This new operating system was created to fit the needs of new hardware devices. The main hardware requirement includes 486 and above *Intel* and *AMD* processors for a *Windows CE* platform. However, current implementations seem to use mostly non-*Intel* processors. Now we get to the issue of CPUs. Both *Windows NT* and *Windows CE* are capable of running on machines that have different CPUs. The same PE file format is used on the different machines, but the actual executed code contains the compiled binary for the actual processor and the PE header contains information of the actual processor type needed to execute the image.

All of these platforms contain different implementations of Win32 functions. Most functions are available on all implementations, which means that a program can call them regardless of the actual platform it is running on. Most of the API differences are related to the actual operating system capabilities and available hardware resources. For instance, CreateThread simply returns NULL when called under Win32s. The *Windows CE* API set consists of several hundreds of functions but it does not support trivial functions like GetWindowsDirectory at all since the *Windows CE* KERNEL is designed to be placed in ROM of the hand-held PC. Due to the hardware's severe restrictions (*Windows CE* has to run on machines with 2MB or 4MB of RAM without disk storage), Microsoft was forced to create a new operating system that had a smaller footprint than either *Windows NT* or *Windows 95*. While different implementations of the Win32 API implement some of the Win32 APIs differently or not at all, generally it is feasible to write a single program which will work on any platform that supports Win32 APIs. Virus writers already understand this fact well. Their first creations attacked *Windows 95* specifically, but they

slowly improved the infection methods to attack the Portable Executable file format so that the actual infected program remains compatible and executed correctly under *Windows NT*, too.

Most *Windows 95* viruses depend on *Windows 95* system behaviours and functionality such as VxD (Virtual Device Driver) and VMM (Virtual Machine Manager)-related features, but some of them contain only a certain amount of bugs and need only slight fixes to be able to run under more than one Win32 platform, such as *Windows 95/Windows NT*. Detection and disinfection of such viruses is not a trivial task. The disinfection part can be essentially difficult to implement. This is because, so far, the Portable Executable structure is much more complicated than any other executable file format used by DOS or *Windows 3.x* that scanners support.

As an anti-virus researcher, my wish is to prevent the Win32 virus situation from escalating. This is what prompted me to write this paper. I shall describe the known infection techniques as well as the possible new ones needed to design a heuristic detector against this trend of computer viruses.

2 INFECTION TECHNIQUES

This section describes the different ways in which a 32-bit *Windows* virus can infect different kinds of executable programs used by *Windows 95/Windows NT*. Since the most common file format is the Portable Executable format, most of the infection methods are related to that. The Portable Executable format makes it possible for viruses to jump from one 32-bit *Windows* platform to another easily. We shall concentrate on infection techniques which attack that particular format since these viruses have a strong chance of becoming a new trend in the near future.

Early *Windows 95* viruses have a VxD part which is dropped by other infected objects such as DOS, EXE and COM executables or a PE application. Some of these infection methods are not related to Win32 platforms on the API level. For instance, VxDs are only supported by *Windows 9x* and *Windows 3.x*, not *Windows NT*. VxDs have their own 32-bit, undocumented, Linear Executable (LE) file format. It is interesting to note that this format was 32-bit even at the time of 16-bit *Windows*. *Microsoft* was not able to drop the support of VxDs from *Windows 95* because of the many third party drivers developed to handle special hardware components. The LE file format remained undocumented by *Microsoft* but there is already at least one virus (Navrhar) which infects this format correctly. I will describe these infection techniques shortly in order to explain the evolution of Win32 viruses.

2.1 INTRODUCTION TO PORTABLE EXECUTABLE FILE FORMAT

In the following section, I will provide an introductory tour of the Portable Executable (PE) file format that *Microsoft* designed for use by all its Win32 operating systems (*Windows NT*, *Windows 95*, Win32s and *Windows CE*). There are several good descriptions of the format on the *Microsoft Developer Network* CD-ROM as well as in many other *Windows 95*-related books. Therefore, I shall describe the PE format from the point of view of known virus infection techniques. To understand how Win32 viruses work, you need to understand the PE format. It is that simple.

The PE file format plays a key role in all of *Microsoft*'s operating systems for the foreseeable future. It is common knowledge that *Windows NT* has a VAX VMS and Unix heritage. The PE format is very similar to COFF (Common Object File Format) but it is an updated version of it. It is called *portable* because the same file format is used under various platforms.

The most important thing to know about PE files is that the executable code on disk is very similar to what the module looks like after Windows has loaded it for execution. This makes the system loader's job much more simple. In 16-bit *Windows*, the loader has to spend a long time preparing the code for execution. This is because in 16-bit *Windows* applications all the functions which call out to a DLL have to be relocated. Some huge applications can have thousands of relocations for API calls which have to be

patched by the system loader while reading the file in portions and allocating memory for its structures one by one. PE applications do not need relocation for library calls any more. Instead, a special area of the PE file, the Import Address Table (IAT) is used for that functionality by the system loader. The IAT plays a key rule in Win32 viruses and I shall describe it later on in detail.

For Win32 all the memory used by the module for code, data, resources, import tables and export tables is in one continuous range of linear address space. The only thing that an application knows is the address where the loader mapped the executable file into memory. When the Base Address is known, the various pieces of the module can easily be found by following pointers stored as part of the image.

Another idea we should be familiar with is the Relative Virtual Address, or RVA. Many fields in the PE files are specified in terms of RVAs. An RVA is simply the offset of an item, to where the file is mapped. For instance the Windows loader might map a PE application into memory starting at address 0x400000 (most common Base Address) in the virtual address space. If a certain item of the image starts at address 0x401234 then the item's RVA is 0x1234.

Another concept to be familiar with when investigating PE files and the viruses which infect them is the 'section'. A section in a PE file is roughly equivalent to a segment in a 16-bit NE file. Sections contain either code or data. Some sections contain code or data declared by the actual application, while other data sections contain important information for the operating system. Before jumping into important details of the PE file, examine figure 1 which shows the overall structure of a PE file.

2.2.1 The PE header

The first important part of the PE format is the PE header. Just like all the other *Microsoft* executable file formats, the Portable Executable file has a header area with a collection of fields at an easy to find location. The PE header describes vital pieces of the Portable Executable image and it is not at the very beginning of the file, but the old DOS stub program is presented there. The DOS stub is just a minimal DOS EXE program which displays an error message (usually 'This program cannot be run in DOS mode'). Since this header is presented at the beginning of the file, some DOS viruses can infect Portable Executable images correctly at their DOS stub. However, Windows 95 and *Windows NT*'s system loaders will execute PE applications correctly as 32-bit images and the DOS stub program remains as a compatibility issue with 16-bit *Windows* systems.

The loader picks up the PE header's file address from the DOS header 'lfanew' field. The PE header starts with an important magic value PE\0\0. After that is the Image File Header structure, followed by the Image Optional Header.

From now on I will only describe the important fields of the PE header which are involved with *Windows 9x*/Win32 viruses. The fields are in order, but I will concentrate on the most commonly used values and therefore several of them will be missing from this list.

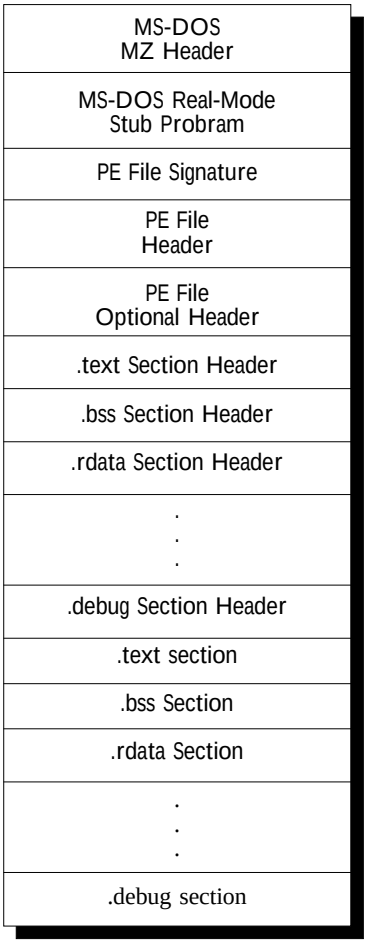

Figure 1. Overall structure of a Portable Executable file image.

Important fields of the Image File Header:

WORD Machine

Indicates the CPU for which this file is intended. Almost every *Windows 9x* virus checks this field by looking for the Intel i386 magic value before actual infection. That is because all the known Win32 viruses are compiled to be executed on Intel-based platforms. (There is a certain risk that we will see RISC-based Win32 viruses as well as Win32 viruses with multiprocessor support in the future.) The *Machine* value is not usually changed by viruses.

WORD NumberOfSections

The number of sections in the EXE (DLL). This field is used by viruses for many different reasons. For instance, the *NumberOfSections* field is incremented by viruses which add a new section to the PE image and place the virus body in that section. (When this field is changed by the virus code, the section table is patched at the same time.)

WORD Characteristics

The flags with information about the file. Most viruses check these flags to be sure that the executable image is not a DLL but a program. (Some of the *Windows 9x* viruses infect KERNEL32.DLL. If so, the field is used to make sure that the executable is a DLL.) This field is not usually changed by viruses.

Important fields of the Image Optional Header:

WORD Magic

The Optional Header starts with a 'Magic' field. The value of the field is checked by some viruses to make sure that the actual program is a normal executable and not a ROM image or something else.

DWORD SizeOfCode

This field describes the rounded-up size of all executable sections. Usually viruses do not fix the value when adding a new code section to the host program. However, some future viruses may change this value also.

DWORD AddressOfEntryPoint

The address where the execution of the image begins. This value is an RVA which normally points to the .text (or CODE) section. This is a crucial field for most *Windows 9x*/Win32 viruses. The field is changed by most of the known virus infection types to point to the actual entry point of the virus code.

DWORD ImageBase

When the linker creates a PE executable, it assumes that the image will be mapped to a specific memory location. That address is stored in this field. If the image can be loaded to the specified address (nowadays 0x400000 in *Microsoft* programs) then the image does not need relocation patches by the loader. This field is used by most viruses before infection to calculate the actual address of certain items, but usually not changed.

DWORD SectionAlignment

When the executable is mapped into memory each section has to start at a virtual address that is a multiple of this value. This field minimum is 0x1000 (4096 bytes) but linkers from *Borland* use much bigger defaults such as 0x10000 (64KB). Most Win32 viruses use this field to calculate the correct location for the virus body, but do not change the field.

DWORD FileAlignment

In the PE file the raw data starts at a multiple of this value. Viruses do not change this value, but use it in a similar way to SectionAlignment.

DWORD SizeOfImage

When the linker creates the image, it calculates the total size of the portions of the image that the loader has to load. This includes the size of the region starting at the image base, up through the end of the last section. The end of the last section is rounded up to the nearest multiple of section alignment. Almost every PE infection method uses and changes the SizeOfImage value of the PE header. Not surprisingly many viruses calculate this field incorrectly which makes the image execution impossible under *Windows NT*. This is because the *Windows 9x*'s loader does not bother to check this value when executing the image. Usually (and fortunately) virus writers do not test their creations for long if at all. Most *Windows 95* viruses contain this bug. Some anti-virus software used to calculate this field incorrectly when disinfecting files. This causes a side-effect – a *Windows NT*-compatible Win32 program will not be executed by *Windows NT* but only by *Windows 9x*, even when the application has been disinfected.

DWORD Checksum

This is a checksum of the file. Most executables contain zero in this field. All DLLs and drivers however have to have a checksum. It is another fact that *Windows 95*'s loader simply ignores the checking of this field before loading DLLs which makes it possible for some *Windows 95* viruses to infect

KERNEL32.DLL very easily. This field is used by some viruses to represent an infection marker to avoid double infections.

2.2.2 The Section Table and commonly encountered sections

Between the PE header and the raw data for the image's sections lies the Section Table. The Section Table contains information about each section of the actual PE image (figure 2). Basically, sections are used to separate different functioning modules from each other – such as executable code, data, global data, debug information, relocation, etc. The Section Table modification is important for viruses in order to specify their own code section or to patch an already existing section to fit into it with actual virus code. Each section in the image has a section header in the section table. These headers describe the name of each section (.text, .bss… reloc) as well as its actual, virtual and raw data locations and sizes. First generation viruses, like Boza, patch a new section header in the section table. (Boza adds its own *.vlad* section there which describes the location and size of the virus section.) Sometimes there is no place for a section header in the file and the patch cannot take its place easily. Therefore, nowadays, viruses (for instance Win95/Anxiety variants) attack the last existing section header and modify its fields to fit the virus code in that section. This makes the virus code section less visible and the infection method less risky.

```
01 .text  VirtSize: 000096B0 VirtAddr: 00001000
  raw data offs: 00000400 raw data size: 00009800
  relocation offs: 00000000 relocations: 00000000
  line # offs:  00000000 line #'s:  00000000
  characteristics: 60000020 CODE MEM_EXECUTE MEM_READ
02 .bss  VirtSize: 0000094C VirtAddr: 0000B000
  raw data offs: 00000000 raw data size: 00000000
  relocation offs: 00000000 relocations: 00000000
  line # offs:  00000000 line #'s:  00000000
  characteristics: C0000080 UNINITIALIZED_DATA MEM_READ MEM_WRITE
03 .data  VirtSize: 00001700 VirtAddr: 0000C000
  raw data offs: 00009C00 raw data size: 00001800
  relocation offs: 00000000 relocations: 00000000
  line # offs:  00000000 line #'s:  00000000
  characteristics: C0000040 INITIALIZED_DATA MEM_READ MEM_WRITE
04 .idata VirtSize: 00000B64 VirtAddr: 0000E000
  raw data offs: 0000B400 raw data size: 00000C00
  relocation offs: 00000000 relocations: 00000000
  line # offs:  00000000 line #'s:  00000000
  characteristics: 40000040 INITIALIZED_DATA MEM_READ
05 .rsrc  VirtSize: 000015CC VirtAddr: 0000F000
  raw data offs: 0000C000 raw data size: 00001600
  relocation offs: 00000000 relocations: 00000000
  line # offs:  00000000 line #'s:  00000000
  characteristics: 40000040 INITIALIZED_DATA MEM_READ
06 .reloc VirtSize: 00001040 VirtAddr: 00011000
  raw data offs: 0000D600 raw data size: 00001200
  relocation offs: 00000000 relocations: 00000000
  line # offs:  00000000 line #'s:  00000000
  characteristics: 42000040 INITIALIZED_DATA MEM_DISCARDABLE MEM_READ
```

Figure 2. A typical Section Table from CALC.EXE.

The name of the section can be anything. It could even contain just zeros, the loader does not seem to worry about that much. Generally, the name field describes the actual functionality of the section. There is a chance for confusion here because the actual code is placed to a .text section of the PE files. This is the traditional name, the same as in the old COFF format. The linker concentrates all the .text section of the various OBJ files to one big .text section and places this in the first position of the Section Table. As I will describe later, the .text section contains not only code, but an additional jump table for DLL library calls. The *Borland* linker calls the .text section as CODE, which is not a traditional name but not beyond normal understanding.

Another common section name is *.data* where the initialized data goes, while the *.bss* section contains uninitialized static and global variables. The *.rsrc* contains stores the resources for the application.

The *.idata* section contains the Import Table – a very important part of PE format for viruses. The *.edata* section is also very important part for viruses since this one lists all the APIs that the actual module exports for other executables. (More on that in sections 2.2.3 and 2.2.4.)

The *.reloc* section stores the Base Relocation Table. Some viruses take special care of relocation entries of the executables, this section however seems to disappear from most Windows 98 executables from *Microsoft*. The reason is that somehow the .reloc section has an early PE format design problem. Since the actual program is loaded before its DLLs and the application is executed in its own virtual address space, there seems to be no real need for that.

Last but not least, there is a common section name, the *.debug* section, which holds the debug information of the executable (if there is any) – not important for viruses however.

Since the name of the section can be specified by the programmer, a certain amount of executables contain all kind of special names by default.

Three of the section table header's fields are very important for most viruses: *VirtualSize* (holds the virtual size of the section, the size before rounding up to the nearest file-alignment) *SizeOfRawData* (the size of the section after it has been rounded up to the nearest file-alignment) and the Characteristic field.

The Characteristic field holds a set of flags which indicates the section's attributes (code, data, readable, writable, executable, etc.). The code section has an executable flag, but does not need writable attributes since the data are separated. This is not the same with appended virus code which has to keep its data area somewhere in its code. Therefore, viruses have to check for and change the Characteristic field of the section in which their code will be presented.

All of these indicate that the actual disinfection of a *Win95*/Win32 virus can be more complicated than that of a normal DOS EXE virus. The infection itself is not trivial in most methods but so many sources are available on various Internet locations that virus writers have all the necessary support from each other to be able to write a new virus easily.

2.2.3 PE File Imports: How are DLL's linked to executables?

Most of the *Windows 9*x and *Windows NT* viruses are based heavily on the understanding of the Import Table which is a very important part of the Portable Executable structure. In Win32 environments, DLLs are linked through the PE file's Import Table to the application which uses them. The Import Table holds the names of the imported DLLs as well the name of the imported functions from those DLLs, too (figure 3).

```
ADVAPI32.DLL
Ordn Name
 285 RegCreateKeyW
 279 RegCloseKey

KERNEL32.DLL
Ordn Name
 292 GetProfileStringW
 415 LocalSize
 254 GetModuleHandleA
 52 CreateFileW
 278 GetProcAddress
 171 GetCommandLineW
 659 lstrcatW
 126 FindClose
 133 FindFirstFileW
 470 ReadFile
 635 WriteFile
 24 CloseHandle
 79 DeleteFileW
```

Figure 3. Partial Import Table dump.

The executable code is located in the .text section of PE files (or the CODE section as the *Borland* linker calls it). When the application calls a function which is in a DLL the actual CALL instruction does not call the DLL directly. Instead, the CALL instruction goes first to a JMP DWORD PTR [XXXXXXXX] instruction somewhere in the executable's .text section (or .icode section in the case of *Borland* linkers). The address that the JMP DWORD PTR instruction looks up is stored in the .idata section (sometimes in .text). The JMP instruction transfers control to that address, which is the intended target address. Thus, the DWORD in the .idata section contains the real address of the operating system function entry point as shown in figure 4.

```
.text (CODE)

0041008E E85A370000  CALL 004137ED ; KERNEL32!FindFirstFileA

004137E7 FF2568004300 JMP  [KERNEL32!GetProcAddress] ; Þ00430068
004137ED FF256C004300 JMP  [KERNEL32!FindFirstFileA] ; Þ0043006C
004137F3 FF2570004300 JMP  [KERNEL32!ExitProcess] ;Þ 00430070
004137F9 FF2574004300 JMP  [KERNEL32!GetVersion] ;Þ 00430074

.idata (00430000)
.
00430068 1E3CF177  ;-> 77F13C1E Entry of KERNEL32!GetProcAddress
0043006C DBC3F077  ;-> 77F0C3DB Entry of KERNEL32!FindFirstFileA
00430070 6995F177  ;-> 77F19569 Entry of KERNEL32!ExitProcess
00430074 9C3CF177  ;-> 77F13C9C Entry of KERNEL32!GetVersion
```

Figure 4. Application calls FindFirstFileA, KERNEL32 function.

The calls are implemented this way to make the loader's job easier and faster. By thunking all calls to a given DLL function through one location, there is no longer the need for the loader to patch every

instruction that calls a DLL. All the PE loader has to do is to patch the correct addresses into the list of DWORDs in the .idata section for each imported function.

The Import Table is very useful for modern 32-bit *Windows* viruses. Since the system loader has to patch the addresses of all the APIs that a Win32 program uses by importing, viruses can easily get the address of an API they need to call by looking into the host program's Import Table.

With traditional DOS viruses, the above problem does not exist. When a DOS virus wants to access a system service function it simply has to call a particular interrupt with the corresponding function number. The actual address of the interrupt is placed in the Interrupt Vector Table and it is picked up automatically during the execution of the program. The Interrupt Vector Table is not saved from the running programs, all applications can read and write into it because there are no privilege levels in DOS. The OS and all applications are sharing the same available memory with equivalent rights. Therefore, the access for a particular system function does cause problems for a DOS virus. It has access to everything it needs by default, regardless of the used infection method.

A *Windows 95* virus has to call APIs or system services to operate correctly. Most 32-bit applications use the Import Table which the linker prepares for them. There are a couple of ways to avoid imports which are sometimes a must. When an application is linked to a DLL, the actual program cannot be executed if the system loader cannot load all the DLLs which are specified in the import table. Moreover, the system loader checks all the necessary API calls and patches their addresses into the Import Table. If the loader is unable to locate a particular API by its name or ordinal value, the application cannot be executed. Some applications have to overcome this problem. For instance, if a Win32 program wants to list all the running processes by name under both *Windows 95* and *NT*, it has to use system DLLs and API calls under *Windows 95* different from those under *Windows NT*. In such a case, the application is not linked statically for all the DLLs it wants to access, since the program could not be executed on any system. Instead, the LoadLibrary function is used to load the necessary DLLs and GetProcAddress is used to get the API's addresses. The actual program can access the API address of LoadLibrary and GetProcAddress from its Import Table. That solves the chicken and egg problem: how to call an API without knowing its address if an API call is needed.

As we will see later on, Boza solves the problem by using hard-coded API addresses. Modern Win32 viruses, however, are capable of searching the Import Table during infection time and saving pointers to the .idata section's important entries. Whenever the application has imports for a particular API, the attached virus will be able to call it.

2.2.4 PE File Exports

The opposite of importing a function is 'exporting' a function for use by EXEs or other DLLs. A PE file stores information about its exported functions in the .edata section (figure 5).

```
Entry Pt Ordn Name
000079CA  1 AddAtomA
.
0000EE2B 38 CopyFileA
.
0000C3DB 131 FindFirstFileA
.
00013C1E 279 GetProcAddress
```

Figure 5. Exports in KERNEL32.DLL.

KERNEL32.DLL's Export Table consists of an Image_Export_directory which has pointers to three different lists: Function Address Table, Function Name Table and Function Ordinal Table. Modern *Windows 95/NT* viruses search for the GetProcAddress string in the Function Address Table in order to be able to retrieve the API function Entry Point value. When this value is added to the ImageBase it gives back the 32-bit address of the API in the DLL. In fact, this is almost the same algorithm that the real GetProcAddress from KERNEL32.DLL follows internally. This function is one of the most important ones for *Windows 95* viruses which want to be compatible with more than one Win32-based system.

When the address of GetProcAddress is available, the virus can get all the API addresses it wants to use.

2.3.1 First generation Windows 95 viruses

The first *Windows 95* virus, known as Win95/Boza.A was introduced in the VLAD virus writer magazine. Boza's authors obviously wanted to be the first with their creation, and they had to find a *Windows 95* beta version very quickly in order to do so. Pioneer viruses used to be very buggy and Boza was not an exception. Basically, the virus cannot work on more than two *Windows 95* versions, a beta release and the final version. Even on those two *Windows 95* releases, the virus causes many GP faults during replication. Infected files are often badly corrupted.

Boza is a typical appending virus which infects Portable Executable applications. The virus body is placed in a new section called .vlad. First the .vlad section header is patched into the Section Table as the last entry and the number of sections field is incremented in the PE header. The body of the virus is appended at the end of the original host program and the PE header's entry-point is modified to point to the new entry-point in the virus section.

Boza uses hard-coded addresses for all the APIs it has to call. That approach is the easiest, but fortunately it is not very successful. The authors of the virus worked on a beta version of *Windows 95* first and used addresses hard-coded for that particular implementation of KERNEL32.DLL. Later they noticed that the actual virus did not remain compatible with the final release of *Windows 9*5. That happened because *Microsoft* did not have to provide the same ordinal values and addresses for all the APIs for every system DLL in all releases. This would be impossible. Different *Windows 95* implementations – betas, language versions, OSR2 releases – do not share the same addresses for their APIs. For instance, the first API call in Boza happens to be GetCurrentDirectoryA. Figure 6 shows that the ordinal value and entry point of GetCurrentDirectoryA is different in the English version of *Windows 95* and in the Hungarian OSR2 *Windows 95* release of KERNEL32.DLL.

```
  Entry Pt Ordn
A. 00007744 304 GetCurrentDirectoryA (Windows 95 ENG)
B. 0000774C 307 GetCurrentDirectoryA (Windows 95 OSR2-HUN)
```

Figure 6. GetCurrentDirectoryA exports in two different *Windows 95* releases.

ImageBase is 0xBFF70000 in both KERNEL32.DLL releases, but the procedure address of GetCurrentDirectoryA is 0xBFF77744 in the English release while it is 0xBFF7774C in the Hungarian OSR2 version. When Boza wants to replicate on the Hungarian version of *Windows 95* it simply calls an incorrect address and obviously fails to replicate. Therefore Boza cannot be called a real *Windows 95*-compatible virus. It turns out that Boza is incompatible with most *Windows 95* releases. Regardless of these facts, many viruses try to operate with hard-coded API addresses. Most of these *Windows 95* viruses cannot become in the wild. Virus writers seem to understand Win32 systems much better already and create viruses which are compatible not only with all *Windows 95* releases but with *Windows 98* and *Windows NT* versions too.

2.3.2 Header infection

This type of *Windows 95* virus inserts itself between the end of PE header (after the Section Table) and the beginning of the first section and modifies the AddressOfEntryPoint field in the PE header to point to the entry point of the virus instead. The first known virus to use this technique is Win95/Murkry.

The virus code has to be very short in *Windows 95* header infections. Since sections have to start at an offset which is a multiple of the FileAlignment, the maximum available place to overwrite cannot reach much more than the FileAligment value. When the application contains too many sections and the FileAligment is 512 bytes, there is no place for the virus code there. The AddressOfEntryPoint field is an RVA; however, the virus code is not placed in any of the sections and therefore the actual RVA is the real physical offset in the file which the virus has to place into the header. It is very interesting to note that the entry point does not point into any code section but, regardless of that fact, *Windows 95*'s loader happily executes the infected program.

There is a chance that a scanner will fail to detect the second generation of such viruses. This happens when the scanner is only tested on first generation samples. In first generation samples the AddressOfEntryPoint points to a valid section. When the scanner looks for the entry point of the program it has to check all the section headers and check if the AddressOfEntryPoint points into any of them. There is a certain chance that this function is not implemented to handle those cases when the entry point does not point into any of the sections. Some scanners may skip the file instead of scanning it from the real entry point and fail to detect the infection in second generation samples.

2.3.3 Prepending viruses

The easiest way to infect Portable Executable files is to overwrite their beginning. Some DOS viruses infect PE files this way, but none of the known *Windows 95* viruses use this infection method. Of course, the application will not work correctly after the infection. Such viruses are discovered almost immediately because of that. This is why viruses which do not want to handle the complicated file format of PE files use the prepending method instead. These viruses are usually written in a high level language (HLL) such as C and even Delphi . This method consists of prepending the virus code to the PE file. The infected program starts with the EXE header of the virus. When the virus wants to transfer the control to the original program code, it has to extract it to a temporary file and execute it from there.

Disinfection of such viruses is easy. The original header information is available at the very end of the infected program in non-encrypted format. Virus writers will recognize that and will encrypt the original header information later on. This will make the task of the disinfector more complicated.

2.3.4 Appending viruses which do not add a new Section Header

A more advanced way of the appending method is used by the Win95/Harry virus. Actually, Harry has a bug in its activation routine which has been fixed in Win95/Anxiety. The Anxiety virus is very similar to Boza in its infection mechanism. However, the virus does not add a new section header at the end of the Section Table, but patches the last section's section header to fit into it. This way the virus can infect all PE EXE files easily. There is no need to worry that the actual section header does not fit into the Section Table. By modifying the VirtualSize and SizeOfRawData fields the virus code can be placed at the end of the executable. This way the NumberOfSection field of the PE header should not need to be modified. The AddressOfEntryPoint field is changed to point to the virus body and the SizeOfImage is recalculated to represent the new size of the program.

```
06 .reloc VirtSize: 00001040 VirtAddr: 00011000
 raw data offs: 0000D600 raw data size: 00001200
 relocation offs: 00000000 relocations: 00000000
 line # offs:  00000000 line #'s:  00000000
 characteristics: 42000040 INITIALIZED_DATA MEM_DISCARDABLE MEM_READ

06 .reloc VirtSize: 00002040 VirtAddr: 00011000
 raw data offs: 0000D600 raw data size: 00001640
 relocation offs: 00000000 relocations: 00000000
 line # offs:  00000000 line #'s:  00000000
 characteristics: E0000040 INITIALIZED_DATA MEM_EXECUTE MEM_READ
MEM_WRITE
```

Figure 7. The last section of CALC.EXE before and after Win95/Anxiety.1358 infection.

The characteristics field of the last section header is changed to have writable/executable attributes (figure 7). The writable characteristic is enough in itself to execute code from any section.

2.3.5 Appending viruses which do not modify the AddressOfEntryPoint

Some *Windows 95* and Win32 viruses do not modify the AddressOfEntryPoint field of the infected program. The virus appends its code to the PE file, but gets control in a more sophisticated way. It calculates where the original AddressOfEntryPoint points to and places a JMP instruction there which will point to the virus body. Fortunately, it is much harder to write such viruses. This is because the virus has to take care of the relocation entries which are pointing into the overwritten part of the code. The Win32/Cabanas virus masks out the relocation entries which are pointing to that area. Win95/Marburg does not place a JMP instruction at the entry point if it finds relocations for that area; instead it modifies the AddressOfEntryPoint field. The JMP instruction should not be the first instruction in the program. Win95/Marburg shows this by placing the JMP instruction after a random garbage block of code when no relocations are presented in the first 256 bytes of entry point code. This way, it is not obvious to scanners and integrity checkers how to figure out the entry point of the virus code.

2.3.6 KERNEL32.DLL infectors

Most *Windows 95* viruses are Portable Executable (PE) infectors, some of them infect DOS COM, EXE programs, VxDs, *Word* documents and 16-bit *Windows* New Executables (NE). Others may infect Dynamic-Link Libraries (DLLs) accidentally, since these are linked in PE (or NE) formats, but the infection is not able to spread further because the standard entry point of the DLLs is not called by the system loader. Instead the DLL's execution normally starts at its specified DLLEntry-point. Thus, KERNEL32.DLL infectors do not attack the entry point. Instead, these types of virus have to gain control differently. Portable Executable files have many other entry points which are useful for viruses, especially DLLs which are export APIs (their entry points) by their nature. Therefore the easiest way to attack KERNEL32.DLLs is to patch the export RVA of one of the APIs (for instance, GetFileAttributesA) to point to the virus code at the end of the DLL image. Win95/Lorez uses this approach. Viruses like this are able to go 'resident' easily. The system loads the infected DLL during the system initialization period. After that, every program which has KERNEL32.DLL imports will be attached to this infected DLL. Whenever the application has a call to the API in which the virus code has been attached, the virus code gets control.

All the system DLLs contain a pre-calculated checksum in their PE header, placed there by the linker. Unlike *Windows 95, Windows NT* recalculates this checksum before it loads the DLL. If the calculated checksum is not the same as in the header of the DLL, the system loader will stop with an error message

during the blue screen boot up period. However, this does not mean that such a virus cannot be implemented for *Windows NT* – it just makes it a bit more complicated. While the checksum algorithm is not documented by *Microsoft*, there are APIs available in IMAGEHLP.DLL for these purposes (like CheckSumMappedFile) which are efficient enough to calculate a new, correct checksum after the actual infection is done. However, this is not enough for *Windows NT*'s loader. There are several other steps to take but there is no doubt that virus writers will be able to solve these questions soon. There is a need for virus scanners to check the consistency of a KERNEL32.DLL by recalculating the PE header checksum, especially if the scanner is a Win32 application itself and is attached to an infected KERNEL32.DLL.

2.3.7 Companion infection

Companion viruses are not very common. Nevertheless, some virus writers do develop *Windows 95* companion viruses. A path companion virus depends on the fact that the operating system always executes files which have a COM extension first in preference to an EXE extension, if the names of two files in the same directory differ only in their extensions. These viruses simply look for a PE application which has an EXE extension and then copy themselves by the same name in the same directory (or somewhere on the path) with a COM extension using the host's name. Win95/Spawn.4096 uses this technique. This functionality is implemented by using FindFirstFileA, FindNextFileA APIs for search CopyFileA to copy the virus code and CreateProcessA to execute the original host program.

2.3.8 Fragmented Cavity Infection

Originally I wanted to describe this infection technique as one which would possibly be developed in the near future. However, the Win95/CIH virus has already introduced this technique during the writing of this paper.

There is slack space between most sections which is usually filled with zeros (or 0xCC) by the linker. This is because the sections have to start at the file alignment as described in the PE header's FileAlignment field. The actual virtual size each section uses is usually different from the raw data representation. Usually the virtual size is a smaller value. In most cases *Microsoft's Link* program generates PE files like that. The difference between the RAW data size of the section and the virtual size is the actual alignment area which is filled by zeros and not loaded when the program is mapped into its own address space.

Since the default value of FileAligment is 512 bytes (usual sector size) the usual slack area size is smaller than 512 bytes. When I first considered this kind of infection method, I thought that no viruses like that would be developed. This is because less than 512 bytes is not big enough for an average PE infector virus of that kind. However, two minutes later I had to recognize that this simple problem itself would not stop virus writers developing such viruses. The only thing which has to be done by the virus is to split its virus body into several parts, then into as many section alignments as available. The loader code for these blocks can be a very short one, first moving the separated code blocks to an allocated memory area one by one. This code itself fits into a big enough section alignment area.

This is the precise method used by the Win95/CIH virus. This makes the job of the scanner and the disinfector much harder. The virus changes the virtual size of the section to be the same as the raw data size in each Section Header into which it injects a part of its virus body. This makes exact identification of such viruses much harder than that of normal viruses. This is because the virus body has to be fetched from different areas of the PE image first.

Win95/CIH uses the header infection method (see 2.3.2) at the same time and infects *Microsoft Linker*-created images without any problem. The fragmented cavity infection technique has a very important advantage from the virus's point of view. The size of the infected file does not get bigger after the infection, but remains the same. This makes noticing the virus much harder. The identification of the virus has to be done very carefully. This is because a virus like that may split its body at any offset which

may separate the actual search string to several parts too. This fact shows that it is very important to analyse new *Windows 95* viruses with extreme care, otherwise the scanner may not find all generations of the same virus code.

2.3.9 Modification of lfanew field in old EXE header

This is the second infection method which I originally wanted to describe as one that is not developed yet. However – just like with the Fragmented Cavity Infection (see 2.3.8) method – this technique appeared in a virus during the time of writing this paper. This infection method is one of the simplest to implement and therefore will be used in many viruses. The first known virus to use this method is Win95/Cerebrus. The method itself would work on *Windows NT*, but there is a trivial bug in the virus and it makes this impossible. Basically, this infection method is an appending type – the virus body is attached to the very end of the original program. The important difference is that the virus code itself contains its own PE header. When the virus infects a Portable Executable application it modifies the lfanew field (at 0x3c address) in the old EXE header. As I have described earlier, the lfanew field holds the file address of the PE header. Since this field will point to a new PE header, the program is executed as it would contain only the virus code. The virus functions as a normal Win32 application. It has its own imports and can easily access any APIs it wants to call. When the replication is done, the virus creates a temporary file with a copy of the infected program. In this file the lfanew field will point correctly to the original PE header. Thus, the original program is functional again when the virus executes the temporary file.

2.3.10 VxD-based Windows 95 viruses

Most *Windows 95* viruses are direct action infectors. Virus writers recognized the importance of fast infection and tried to look for solutions to implement *Windows 95* resident viruses. Not the easiest, but the evident solution was to write a VxD virus. One of the first VxD-based virus was Win95/Memorial. It infects DOS COM, EXE and PE applications. The virus does not replicate without *Windows 95*. The infected programs are droppers which are supposed to extract the real virus code – a VxD into the root directory of drive C: as CLINT.VXD. When the VxD is loaded, the virus code is executed on ring 0, thus the virus can do anything it wants. VxDs can hook the file system easily and that is exactly what most of the VxD viruses want to do. They simply hook the Installable File System (IFS) with one simple VxD service routine. After that, the virus is able to monitor access to all files. The VxD code has to be extracted and the dropper code needs different implementation for each and every format what the virus wants to infect. This makes the virus code very complicated and relatively big (12,413 bytes) and therefore it is very unlikely that many viruses like this will be developed in the future.

2.3.11 PE viruses which operate as VxDs

A much easier solution (compared to method 2.3.10) has been introduced by the Win95/Harry and Win95/Anxiety viruses. These viruses are able to overcome complications by patching their code into the VMM (Virtual Machine Manager) of *Windows 95*.

When an infected PE program is executed, the virus code takes control. Programs are executed on the application level, which is why they cannot call system level functions (VxD calls) normally. These viruses bypass the system by installing their code into the VMM which runs on Ring 0. The installation routine of such a virus searches for a big enough hole in the VMM's code area after 0C0001000h address. If a large enough area, consisting of only 0FFh bytes, is detected, the virus looks for the VMM header at 0x0C000157Fh and checks this area by comparing it to 'VMM'. If this is detected, the virus picks up the Schedule_VM_Event system function's address from the VMM and saves it for later use. Then it copies its code into the VMM by overwriting the previously located hole and changes the original Schedule_VM_Event's address to point to a new function. Finally, it executes the original host program, by jumping to the original entry point. This all is possible because *Microsoft* was unable to protect that

area from changes to keep backward compatibility with old Windows 3.x VxDs. The full VMM area is available for read and write functions for application level programs.

Before the host program can be executed, the VMM will call Schedule_VM_Event which is now replaced by the initialization routine of the virus. This code is executed on Ring 0 already which makes it able to call VxD functions. Anxiety hooks the IFS by calling IFSMgr_InstallFileSystemApiHook from there. This installs the new hook API of the virus.

The virus replication code needs special care. When VxD code is executed, VxD calls are patched by the VMM. The VMM turns the 0CDh, 20h, DWORD function id (INT 20H, DWORD ID) to FAR CALLS. Some of the VxD functions consist of a single instruction. In this case, the VMM patches the six bytes with this single instruction which fits there. The VMM does this dynamically with all the executed VxDs to speed up their executions.

When the virus code is executed, the VxD functions in the virus body are patched by the VMM and the virus therefore cannot copy this image immediately to files again, since the virus code would not work in a different *Windows 95* environment. These viruses contain a function which patches all their VxD functions back to their normal format first and only after that replicates the code into the host program. Even if this technique looks very complicated, it is not very difficult for virus writers. Win95/Anxiety variants are in the wild in many countries.

There is no doubt that several viruses will try to overcome the Ring 3 ⩾ Ring 0 problem by using similar methods. Win95/CIH uses instructions which are available only from *Intel* 386 processors and above. It is interesting to note that the Interrupt Descriptor Table is available to write under *Windows 95* (since it is part of the VMM). Win95/CIH uses SIDT (Store IDT) instruction to get a pointer to the IDT. This way, the virus is able to modify the gate descriptor of INT 3 (debug interrupt) in the IDT and allocate memory by using VxD services. The INT 3 routine will be executed as a Ring 0 interrupt, from its PE virus body. This trick shows how easy is it is for virus writers to overcome the Ring 3, Ring 0 problem. Similar methods will be discovered by *Windows 95* virus writers in the near future which will result in an even simpler method.

2.3.12 VxD Infection

There is at least one virus, Navrhar, which infects VxDs. Navrhar also infects *Word* documents which are in the OLE2 format and some standard system VxDs. The virus does not infect unknown VxDs, but only known system VxDs which are listed in its PE dropper. When an infected *Word* document is opened, the virus extracts its PE dropper which is attached to the very end of the document and therefore the only way to access this code is to use Win32 APIs. That is why the virus imports KERNEL32.DLL APIs in its macro code. When the dropper's code is extracted from the document, it is executed. The dropper checks for the listed VxDs and infects them one by one. When the system is rebooted, one of the infected VxDs will be loaded by *Windows 95*. The virus takes control from the infected VxD, hooks the file system and checks for *Word* document accesses from then on and infects them by adding a new macro body and the binary code at the end of the document.

Navrhar illustrates that, unlike DOC files, PE applications are not yet exchanged frequently enough by users, not to mention VxDs, which are not normally exchanged at all. This is why modern Win32 viruses will use some kind of *Word* document multipartite mechanism in order to spread faster.

2.4 WIN32 VIRUSES: DESIGNED FOR MICROSOFT WINDOWS NT/WINDOWS 98?

Microsoft's strategy is clear: The 'Designed for Microsoft Windows NT/Windows 98' logo program's important requirement is that every application in your product must be a *Microsoft* Win32 program compiled with a 32-bit compiler that generates an executable file of the PE format. This indicates that the number of Win32 programs developed by third parties will grow intensively in the future and therefore

people will exchange more Portable Executable programs. The main reason *Windows 95* viruses have not caused big problems so far is because virus writers had to learn a lot to 'support' the new systems. From the other point of view, *Windows 95* viruses were not very successful because people did not exchange enough PE programs. This fact seems to be changing now and this will change in the virus situation. Young virus writers understood *Microsoft*'s message: 'Win32 everywhere' and their answer seems to be 'Win32 viruses everywhere'. These young guys do not waste their time with DOS viruses, but will try to explore Win32 platforms instead. Writing a DOS virus is not very challenging for them any more. Virus scanners are much weaker in handling Win32 viruses – detection and disinfection are not that easy. Vendors have to understand and learn the new file formats and spend a reasonable time researching and designing new scanning technology. When the number of different viruses for a particular platform reaches 50, the next year we can expect hundreds of viruses of that type. It is likely to happen with Win32 viruses. It took about three years for the number of DOS viruses to reach 500. Since *Windows 95* and *Windows NT* are more complicated systems, it is natural that the first period of such viruses will be more than three years. In the following section I will describe some important issues which make a *Windows 95* virus incompatible with *Windows NT*. This specifies the differences between the *Windows 95* and Win32 prefixes that scanners use to identify 32-bit Windows viruses.

2.4.1 Important Windows 95/Windows NT system loader differences

I have to say that I had a different picture of *Windows NT* from the security point of view before understanding Win32/Cabanas, because I had incorrect conclusions about the level of system security when the first *Windows 95* virus, Boza appeared. Most of the anti-virus researchers almost immediately performed some tests with Boza on *Windows NT*. The result looked reassuring: *Windows NT* did not even try to execute the infected image (Figure 8). Actually, what is good for *Window 95*'s loader is not good for that of *Windows NT*. Why? I answered this question myself by patching PE files. The PE file format has been designed by *Microsoft* for use by all its Win32 operating systems (*Windows NT*, *Windows 95* and Win32s and *Windows CE*). That is why all the system loaders in Win32 systems have to understand this executable structure. However the implementation of the loader is different from implementation to implementation. *Windows NT*'s loader simply checks more things in the PE file before it executes the image than *Windows 95*'s. Thus it finds the Boza-infected file suspicious. This happens because one field in the .vlad Section Header (which is patched into the Section Table of the host program) is not precisely calculated by the virus. As a result, correctly calculated sections and section headers can be added to a PE file without any problem. Thus *Windows NT*'s loader does not have any superior virus detection as some may think.

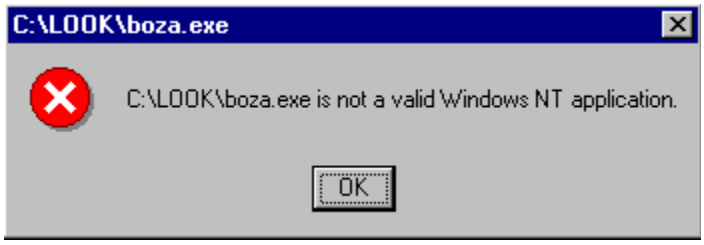

Figure 8. *Windows NT* does not execute a Boza infected application.

If this problem was fixed in Boza, the virus would be capable of starting the host program even on a *Windows NT* platform, however the virus would still not be able to replicate.

This is because of another incompatibly problem, which all the *Windows 95* viruses have had so far. Every *Windows 95* virus has to overcome a specific problem: it has to be able to call two Win32, KERNEL APIs: GetModuleHandleA and GetProcAddress. Since those APIs are in KERNEL32.DLL,

Windows 95 viruses could access those functions from KERNEL32.DLL directly with a hack. Most *Windows 95* viruses have hard coded pointers to GetModuleHandleA and GetProcAddress KERNEL APIs. By using GetProcAddress, the virus can access all APIs it wants to call.

When the linker creates an executable, it assumes that the file will be memory mapped to a specific location in memory. In the PE File Header there is a field called ImageBase holding this address. For executables, this address is usually 0x400000 as default. In the case of *Windows 95*, the KERNEL32.DLL's ImageBase address is 0xBFF70000. Thus, the address of GetModuleHandleA and GetProcAddress will be at a certain fixed location in the same release of KERNEL32.DLL. However, this address can be different in a new release, which makes *Windows 95* viruses incompatible even with other Windows 95 releases. This ImageBase address is 0x77F00000 in *Windows NT* as the default, thus, *Windows 95* viruses which operate with a *Windows 95*-specific base address cannot work on *Windows NT*.

The third incompatibility reason is obvious: *Windows NT* does not support VxDs. Viruses like Memorial cannot operate on *Windows NT*, because they are VxD-based. They should have included different infection algorithms at the driver level for *Windows NT* and *Windows 95* to operate on both systems, which would make them complicated.

If a *Windows 95* virus can overcome the above incompatibility and implementation problems, it will eventually work on *Windows NT* as well. Such viruses may have Unicode support, but it is not a mandatory. Win32/Cabanas supports all of these features, being able to trespass the OS barrier imposed by early *Windows 95* creations.

Both Boza and Cabanas are 32-bit Win32 programs. Cabanas spreads under *Windows 95/Windows 98* (and any other localized versions) and under all major *Windows NT* releases such as 3.51, 4.0 and 5.0 respectively, while Boza replicates only under the English *Windows 95* release. Therefore the prefix part of the virus name is 'Win32' for Cabanas and 'Win95' for Boza. Currently there are no viruses which could be classified with the 'NT' prefix, but we will see viruses which replicate only under *Windows NT*.

2.4.2 DIRECT ACTION WIN32 VIRUSES

Does the direct action method remain a used method in Win32 viruses? The answer is certainly yes. The direct action infection can be implemented to work surprisingly fast. Even if the virus goes through all the files in *Windows* directory, *Windows* System directory and in the current directory respectively, the file infection is fast enough to go unnoticed in most of the systems. This is because these viruses can use 'memory-mapped' files, a new feature implemented in Win32 based systems which simplifies file handling and increases overall system performance. This makes the virus writer's task much easier, since handling of the internal structure of PE files becomes obvious. The full infection process does not take more than a few seconds on a 486 and usually people use *Pentium* PCs for running *Windows NT*. This does not mean, however, that virus writers will not discover new ways to write viruses which can be 'resident' under all major Win32 platforms.

3 ATTACKS AGAINST SCANNERS AND INTEGRITY CHECKERS

In this section I shall describe some new features of *Windows 95*/Win32 viruses which cause problems for scanners and integrity checkers. By understanding these techniques we can design new functions against these kind of viruses to make our scanners stronger for such attacks.

3.1 STEALTH VIRUSES – DIFFICULTIES WITH MEMORY SCANNING

The stealth technique is a typical example of an attack which can be very effective against scanners as well as integrity checkers. Therefore, the stealth technique is implemented in all upcoming virus types

sooner or later. The development of the stealth mechanism in *Windows 95*/Win32 viruses is at a very early stage. Only directory stealth is implemented so far. However, virus writers do already have the necessary knowledge to implement full stealth viruses and we can be pretty sure that they will create such viruses in the near future.

The only effective defence against stealth viruses is some sort of memory scanning in the anti-virus product. The actual implementation of memory scanning under *Windows 95/Windows NT,* however, is not a trivial task. This opens several questions. Even if we just take the problem of memory scanning for traditional boot and DOS file viruses, we can quickly find some problems. As an example, the first 4K of memory is not readable by a Win32 application. This area of memory is protected to catch null pointer bugs more easily under *Windows 95*. While this memory area is easily readable by a DOS application or a VxD, it is not accessible by Win32 programs. A simple stealth virus can be active in an area which the actual Win32 port of your virus scanner may not check because it has never been implemented. One possible solution for this problem is to use a 16-bit DLL for that purpose with a thunking technique. That way, the Win32 process can gain access to this area of memory easily.

Another problem is when a Win95 virus is active as a VxD. For instance, Win95/Anxiety patches itself into the VMM area where VxDs are loaded normally and functioning as a Ring 0 VxD (see 2.3.10). It is not obvious that a scanner is checking this area of memory for active viruses. This area is not accessible by real mode DOS applications at all, but for a Win32 program it can be a trivial task as well as for a VxD.

Memory scanning under *Windows 95* and *Windows NT* has common problems, but the actual implementation has to be very different and is not a simple task for the virus scanner at all. (Explaining the correct implementation of memory scanning is a very big subject and therefore beyond the scope of this paper.) Let us have a look at stealth methods which have been implemented already.

3.1.1 VxDCall INT21_Dispatch Handler

This technique was introduced by Win95/HPS first. Win95/HPS monitors 714Eh, 714Fh LFN FindFirst/FindNext functions which is a mandatory under *Windows 95* from the virus point of view. The actual implementation of the stealth handler is unique. The virus patches the return address of FindFirst/FindNext functions on the fly on the stack to its own handler. This handler checks that the actual program size is divisible by 101 without a remainder. In this case, the virus opens the program with an extended open LFN function, then reads the virus size from the last four bytes of the infected program and subtracts this value as a 32-bit variable from the original return value of FindFirst/FindNext on the stack. Finally it returns to the caller of the function. This way, the virus is able to hide its file size differences from most applications while the size of the virus body should not be a constant value. HPS has to hook VxDCall, an undocumented API in KERNEL32.DLL, first. To be able to do that, the virus has to allocate memory from the shared memory area which is mapped to the address space of all 32-bit processes. This means that this area of memory has to be scanned to be sure about memory infections, otherwise the product may not find a similar virus which uses the full stealth technique, which hides the changes of the file internally as well as its size difference.

3.1.2 Hook on Import Address Table

This method was introduced by Win32/Cabanas. The technique is likely to be reused in new Win32 viruses. The same technique can work under most major Win32 platforms by using the same algorithm.

The hook function is based on the manipulation of the Import Address Table. Since the host program holds the addresses of all imported API in its .idata section, all the virus has to do is to replace those addresses to point to its own API handlers.

First it searches in the Import Address Table for all the possible function names it wants to hook: GetProcAddress, GetFileAttributesA, GetFileAttributesW, MoveFileExA, MoveFileExW, _lopen,CopyFileA, CopyFileW, OpenFile, MoveFileA, MoveFileW, CreateProcessA, CreateProcessW, CreateFileA, CreateFileW, FindClose, FindFirstFileA, FindFirstFileW, FindNextFileA, FindNextFileW, SetFileAttrA, SetFileAttrW. Whenever it finds one it saves the original address to its own JMP table and replaces the .idata section's DWORD (which holds the original address of the API) with a pointer to its own API handlers (figure 9).

```
.text (CODE)

0041008E  E85A370000  CALL 004137ED

004137E7  FF2568004300 JMP  [00430068]
004137ED  FF256C004300 JMP  [0043006C]
004137F3  FF2570004300 JMP  [KERNEL32!ExitProcess]
004137F9  FF2574004300 JMP  [KERNEL32!GetVersion]

.idata (00430000)

00430068  830DFA77 ;-> 77FA0D83 Entry of new GetProcAddress
0043006C  A10DFA77 ;-> 77FA0DA1 Entry of new FindFirstFileA
00430070  6995F177 ;-> 77F19569 Entry of KERNEL32!ExitProcess
00430074  9C3CF177 ;-> 77F13C9C Entry of KERNEL32!GetVersion

NewJMPTable:

77FA0D83  B81E3CF177  MOV  EAX,KERNEL32!GetProcAddress ; Original
77FA0D88  E961F6FFFF  JMP  77FA03EE ;-> New handler
.
.
77FA0DA1  B8DBC3F077  MOV  EAX,KERNEL32!FindFirstFileA ; Original
77FA0DA6  E9F3F6FFFF  JMP  77FA049E ;-> New handler
```

Figure 9. Hooked GetProcAddress and FindFirstFileA APIs through the .idata entries.

Some *Windows* programmers may say now '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 else than GetProcAddress API.' Yes, but this is the exact reason why the virus wants to hook GetProcAddress too.

GetProcAddress is used by the most of the applications. When the application calls GetProcAddress, the new handler first calls the original GetProcAddress to get the address of the requested API. Afterwards it checks if the function is a KERNEL32 API and if it is one of the APIs that the virus wants to hook. If it is an API like that and not hooked yet, the virus returns a new API address which will point into the NewJMPTable. Thus, the application will also get an address to the new handler in such cases.

Win32/Cabanas is a directory stealth virus – during FindFirstFileA, FindFirstFileW, FindNextFileA and FindNextFileW the virus checks for already infected programs. If the program is not infected, the virus will infect it, otherwise it hides the file size difference by returning the original size of the host program. There are no known *Windows* scanners with efficient self-checks against such an attack, before they start to scan for viruses. If the scanner checks for its size difference by calling the above APIs, it cannot detect any difference.

Since the CMD.EXE (Command Interpreter of Windows NT) uses the above APIs during DIR command, every non-infected file will be infected (if the CMD.EXE was infected previously by Win32/Cabanas).

It is evident that a Win32 scanner has to contain a strong self-check on its modules. This self-check function has to recalculate the CRC of each used module in the program, but before that it has to check that none of its APIs is hooked. This can be easily done by searching out the GetProcAddress API's address directly from KERNEL32.DLL. By using the original address of GetProcAddress, the anti-virus product can check all its imported APIs for hooks by comparing the address in the Import Address Table with the one which the original GetProcAddress returns. This way, the application is capable of removing the hook from itself by using WriteProcessMemory debug API. Of course, there is a chance that a future virus will attack KERNEL32.DLL at its GetProcAddress Export RVA by using the method introduced by Win95/Lorez (section 2.3.6). Therefore the consistency of KERNEL32.DLL has to be checked too.

3.2 ANTI-DEBUG WIN95/WIN32 VIRUSES

Currently there are only few Win95/Win32 viruses of this type. Win32/Cabanas and Win95/CIH use very effective anti-debug techniques. These viruses are introducing a new kind of anti-debug method. When Cabanas generates an exception, control is given to a Ring 0 exception handlers first and therefore the program cannot be traced with an application level debugger such as *Borland* TD32. Advanced debuggers can help a bit, but Win95/CIH modifies INT 3's (break point) gate descriptor in the local Interrupt Descriptor Table and this trick makes the virus a bit more difficult to debug. There is a chance that anti-emulating/anti-heuristic virus code will be implemented on the top of these functionalities in upcoming viruses. Thus, modern emulators have to take care of exception handlers and descriptor table modifications.

3.3 EVOLUTION OF WIN32 POLYMORPHISM

Polymorphism is one of the strongest techniques that viruses can use against scanners. Fortunately, it is not easy to write such a virus for Win32 platforms. However, some virus writers have already solved the problems and written successful polymorphic viruses. It is natural that a 32-bit virus code uses 32-bit polymorphic engine and, obviously, 32-bit keys to encrypt the virus code. Modern processors execute 32-bit code much faster. Therefore, these engines will be executed fast even if the encryption is much stronger than one in a 16-bit virus which very often uses 8-bit encryption keys only. It is not a particular problem to deal with an encryption which uses 8-bit keys. However, 32-bit keys are not that easy to handle, because the brute-force attack is just too slow to be included in a scanner. Modern, code emulation-based scanning will face new challenges in the following years. The tendencies of DOS polymorphic viruses show that more and more viruses are using stronger encryption schemes to make the detection and disinfection harder. For instance, the IDEA virus from Spanska uses IDEA (International Data Encryption Standard) cipher as one of its three encryption layers. We suspect that new Win32 mutation engines will be developed by using similar models.

3.3.1 Encrypted viruses

The first encrypted *Windows 95* virus was Win95/Mad. This virus is very badly written and usually crashes the system instead of spreading on it. The actual code of the decryptor is not changed by the virus. The encryption itself does not cause real problems for scanners. The detection can be based on a sufficiently long search string picked up from the decryptor part of the virus. In the long run, however, it is not an efficient enough way to detect such viruses. For exact identification and disinfection the virus code has to be decrypted completely and 32-bit code emulation has to be used.

3.3.2 Oligomorphic technique

The first *Windows 95* virus with oligomorphic capabilities was Win95/Memorial. Memorial's engine is simple but effective. The basic decryptor consists of 11 different parts. The mutation engine modifies the order of these small blocks by swapping some of them with each other – block 1 with 2, 8 with 9, 6 with 7 , 3 with 4, 5 with 4 and 10 with 0. This gives 2*2*2*2*6 all together 96 different cases. That makes the detection of the virus more difficult in PE files, because 96 search strings are needed for complete detection. Not surprisingly, quite a few scanners miss some samples of Win95/Memorial while claiming to detect the virus correctly.

3.3.3 Real 32-bit polymorphic Windows viruses based on generated decryptors

We did not have to wait long to see fully polymorphic *Windows 95* viruses. Some of the first viruses with polymorphic engines are Win95/Marburg and Win95/HPS, written by the same virus writer (GriYo/29A) who wrote Implant, a very successful multipartite, polymorphic virus. These viruses have powerful and advanced polymorphic engines just like Implant's. Win95/HPS supports subroutines (using CALLs, RETs) and conditional jumps with non-zero displacement. The code of the polymorphic engine occupies about the half of the actual virus code. There are random bytes-based blocks inserted in-between the generated code chain of the decryptor. The full decryptor is built up during the first initialization phase, which makes the virus a slow polymorphic. This means that anti-virus vendors cannot test their scanner's detection rate against this virus efficiently, because the infected PC has to be rebooted in order to create a new decryptor. The decryptor consists of a 386-based instruction set. The image is encrypted and decrypted by different methods including XOR, INC, DEC, NOT, ADD and SUB instructions with 8, 16 or 32-bit keys respectively. This drastically reduces the range of ideas from the detection point of view. I am sad to say that the polymorphic engine part is well written, just like the rest of the virus. It was certainly not created by a beginner.

3.3.4 Polymorphism which is not based on generated decryptors

Another very interesting virus with polymorphic capabilities is Win32/Apparition. This virus is in the wild in many countries (for instance, in Russia and France). The polymorphism is not based on generated decryptors which is the most commonly used method. Instead of using a decryptor which contains garbage instructions, the virus mutates itself in source format. The original virus code is written in C++. The source is kept in the virus body. When the virus arrives in a developer environment where certain 32-bit compilers are installed, the virus extracts it source code from its body. Next it mutates its source code, in a very similar way to macro viruses, by inserting random source lines with comments and other do nothing code. Finally it recompiles itself and from that on it replicates in a different binary format. The actual virus code can be very different from the original one and much harder to detect than an average polymorphic virus which uses mutated decryptors.

4 FUTURE TRENDS

While binary viruses have attacked DOS and *Windows 3.x* executable formats during the 1990s this situation will change a lot during the next following years. It is very likely that Win32 viruses will become a trend before the year 2000. There is a chance that old, multipartite virus methods (Boot/DOS binary attackers) will be replaced by a new multipartite virus type in the form of a VBA (Visual Basic for Applications)/Win32 infector. Navrhar is a buggy virus, but it attacks *Word* documents on the file format level. Fortunately it is extremely difficult to write such a virus without major bugs, therefore it is unlikely that many viruses like this will be written in the future. However, it seems that some virus writers already recognize how easy it is to implement a macro virus which attacks more than one VBA file format by using dropping or importing mechanism. Other virus writers use macro viruses to drop binary viruses (including Win95 viruses such us Win95/Anxiety variants). This mechanism will be used for creating very successful VBA/Win32 multipartite viruses in the future.

Let us look at some possible features which can be implemented in Win32 viruses. It is hard to provide such information, because there is always a possibility that virus writers could get hold of these ideas. However, they are more or less obvious to *Windows* virus writers. In fact, I am not sure that some of these ideas are not developed already or will not be seen until this paper gets published. I have had to rewrite a few entries of this section already to accommodate existing infection technique methods. This shows that there is nothing unique in these ideas; every virus writer could come up with them. Moreover, they will find ways that a virus researcher will not able to discover. I am positive that most of these ideas will be implemented during the next few years.

4.1 INSERTING POLYMORPHIC VIRUSES

This type of virus could be much harder for scanners to detect and virus writers will recognize this advantage as an obvious combination. Similar to DOS-inserting viruses, the PE format can be attacked at many different points. The entry point modification based viruses can be detected relatively easily while a virus which attaches itself at a non-obvious entry point is much harder to detect. Basically anything which takes control in the host program's code could be used as an entry-point. For instance calls to any API can be replaced with calls to the virus entry. Such infection does not cause a real problem until the virus code is not polymorphic. A combination of the inserting technique with polymorphism is an existing method in traditional DOS viruses and therefore it is very likely to be implemented in Win32 viruses also. The inserting method will be extended to insert a new virus section in between two existing PE sections.

4.2 ENCRYPTION OF HOST PROGRAMS

Encryption of the host program during infection time can be a very effective way of preventing disinfection of the virus. Unfortunately, it is too easy to create such viruses by using the Win32 file-mapping functionality. That way, even a huge Win32 application can be encrypted during infection which will not be decrypted without the attached virus code. When the virus is disinfected from the application, the host program's code has to be decrypted too. This kind of encryption algorithm can be very hard to analyse especially if the virus code is written in a high level language. There are several Win32 viruses (for instance Win32/Semisoft) which are bigger than 64K of compiled and optimized C code. It is very demanding to analyse such a long virus code even with the best utilities.

4.4 WINDOWS NT SPECIFIC VIRUSES

It is very likely that service viruses and device driver-based viruses will appear on *Windows NT*. Services are typically background applications without any user interface. Services have the advantage of being loaded by the system before any users log in and therefore they need an administrator rights to be installed in the registry. This way, the virus could load itself for background execution. The service could be implemented to wake up at certain time intervals and infect new files. It is very important to note here that *Windows NT*'s administrator rights should not be used by default on machines which do not really need it.

I do not think that we will have to wait long until the first device driver virus is implemented for *Windows NT*. Device drivers, just like services, are PE applications. A virus like that has the advantage of filtering file accesses whenever they happen. The virus has to implement a filter driver to reach that goal. While this is not a very easy task and there are not too many people with enough knowledge to write a filter driver for *NT*, some virus writer will develop this technique for sure. The additional problem for scanners is that the virus code, just like the on-access scanner, is running in kernel mode and has the access for any resources it wants to. It will be very challenging to analyse and fight back against such viruses.

4.5 WINDOWS CE COMPATIBILITY?

Windows CE is advertised very aggressively by *Microsoft* as one of the latest Win32 platforms. While *Windows CE* is running *Pocket Excel* and *Pocket Word* applications, these versions do not support macros, at least not yet, and hopefully they will not. Therefore, macro viruses cannot effect these systems yet. *Windows CE* systems are typically non-Intel based machines. However, there is a 486 and above implementation of it already available. Since *Intel* processors are not used yet in any known *Windows CE* machine and the DOS subsystem is not supported by the OS, there are not any known viruses which can infect a *Windows CE* platform yet. This does not mean that there is no way to implement a virus for this platform. It is very likely that a Win32 virus with multiprocessor platform will be developed which could jump between *Windows CE, Windows 9x* and *Windows NT* systems without any major problem. Such a virus could be developed in C relatively easily. That means that anti-virus vendors have to consider *Windows CE* as a platform which should be supported by a scanner in the future.

4.6 SECTION REPLACING VIRUSES

It is possible that we will see viruses which replace existing code sections in PE files with their own code and shift the overwritten section's code to the end of the file. There are no such viruses currently, because most PE files have relocations in *Windows 95*. Even if it is not used, it is problematic to infect PE files that way. However, in *Windows 98* PE executables there are no .reloc sections in standard applications. This means that the code section replacement becomes an obvious idea for virus writers.

4.7 SECTION INSERTING VIRUSES

I can think of one more way in which virus writers may develop in upcoming Win32 viruses. A new type of virus could insert a new code section after the original code section and shift the rest of the file after the virus body. Such viruses may insert a new section header for that code, but this could be done by making the original code section bigger in its section header and modifying all section headers at the same time. Relocations will not cause problems since these are used for the original code section only which stays at the same file and memory positions, while the rest of the code and data is shifted to a higher memory address by the virus.

5 HEURISTIC ANALYSIS AGAINST WIN95/WIN32 VIRUSES

Heuristic analysis has proved to be a successful way of detecting new viruses. The biggest disadvantage of heuristic analyser-based scanners is that they often find false positives which are not cost-effective for users. In some areas of the virus problem though, the heuristic analyser is a real benefit. For instance, a modern scanner cannot survive without a heuristic scanner for macro viruses. In the case of binary viruses, heuristic scanning can be very effective too, but the actual risk for a false positive is often higher than that caused by good macro heuristics. The capabilities of a heuristic analyser have to be reduced to a level where the number of possible false positives is not particularly high while the scanner is still able to catch a reasonable amount of new viruses. This is not an easy task to do. Heuristic scanning does not exist in a vacuum. Heuristics are very closely related to the good understanding of the actual infection techniques of a particular virus type. Different virus types need completely different rules on which the heuristic analysers logic can be built up. Obviously, heuristic analysers which have been designed to catch DOS viruses or boot viruses are useless in detecting modern Win32 viruses. My primary goal with this paper is to give an introduction to some ideas which can be useful to detect 32-bit Windows viruses heuristically.

Win95 and Win32 viruses are developed in 32-bit code format. The usual method of binary heuristics is to emulate the program execution and look for suspicious code combinations. In the following sections I shall introduce some heuristic flags which are mostly not based on code emulation, but which describe

particular structural problems unlikely to happen in a Portable Executable programs that are compiled with a 32-bit compiler (such as: *Microsoft*, *Borland*, *Watcom*). Although not very advanced, structural checking is an effective way to detect even polymorphic viruses like Win95/Marburg or Win95/HPS. In some cases structural analysis is not efficient enough and will have to be extended with more code emulation-based features in the near future.

5.1.1 Code execution starts in last section

The PE format has a very important advantage in that different functional areas such as code, data areas are separated logically in sections. If we look back to the infection techniques, we can see that most Win32 viruses change the entry point of the application to point not to the .text (CODE) section, but to the last section of the program instead. By default, the linker merges all the object code into the .text section. It is possible to create several code sections, but this does not happen by default compiling and most Win32 applications will never have such a structure. It means that it can be considered very suspicious if the entry point of the PE image does not point into the .text section but to the last section of the program instead.

5.1.2 Suspicious section characteristics

All sections have a characteristic which describes certain attributes. The characteristic holds a set of flags which indicates the section's attributes. The code section has an executable flag, but does not need writable attributes since the data are separated. Very often the virus section does not have executable characteristics, but has writable only or both executable and writable. Both of these cases have to be considered suspicious. Some viruses fail to set the characteristic field and leave the field at 0. That is suspicious too.

5.1.3 Virtual size is incorrect in PE header

The SizeOfImage is not rounded up to the closest section alignment value by most *Windows 95* viruses. While *Windows 95*'s loader allows this to happen, *Windows NT* does not. It is suspicious enough, therefore, if the SizeOfImage field is incorrect. However, this may happen during incorrect disinfection too.

5.1.4 Possible 'gap' in between sections

Some viruses, such as Win95/Boza and Win95/Memorial, round up the file size to the nearest file alignment before adding a new section into it, very similar to DOS EXE infectors. However, the virus does not describe this size difference like in the last section header of the original program. This means that for *Windows NT*'s loader the image looks like it has a 'gap' in its raw data and is not considered a valid image therefore. Many *Windows 95* viruses have this bug and therefore it is a good heuristic flag.

5.1.5 Suspicious Code Redirection

Some viruses do not modify the entry point field of the code. Instead, they put a JMP to the entry point code area instead to point to a different section. It is very suspicious to detect that the code execution chain jumps out from the main code section to some other section close to the entry point of the program.

5.1.6 Suspicious code section name

It is suspicious if a section which normally does not contain code such as .reloc, .debug, etc. gets control. It has to be flagged if code is executed in such sections.

5.1.7 Possible header infection

If the entry point of a PE program does not point into any of the sections, but points to the area after the PE header and before the first section's raw data, then the PE file is likely to be infected with a header infector.

5.1.8 Suspicious imports from KERNEL32.DLL by ordinal

Some Win95 viruses patch the Import Table of the infected application and add ordinal value-based imports into it. Imports by ordinal from KERNEL32.DLL should be suspicious, but some *Windows 95* programmers do not understand that there is no guarantee that a program which imports from system DLLs by ordinals will work in a different *Windows 95* release, and still use them. In any case, it is suspicious if GetProcAddress or GetModuleHandleA functions are imported by ordinal values.

5.1.9 Import address table is patched

This is an addition to section 5.1.8. If the Import Table of the application has both GetProcAddress and GetModuleHandleA imports, and imports these two APIs by ordinal at the same time, then the Import Table is patched for sure.

5.1.10 Multiply PE headers

When a PE application has more than one PE header, the file has to be considered suspicious (see: 2.3.9). This is because the PE header contains many non-used or constant fields. This is the case if the lfanew field points to the second half of the program and it is possible to find another PE header close to the beginning of the file.

5.1.11 Multiply Windows headers and suspicious KERNEL32.DLL imports

Like section 5.1.10, structural analysis can detect prepending viruses by searching for multiple new executable headers such us 16-bit NE and 32-bit PE. This can be done by checking if the real image size is bigger than the actual representation of the code size as described in the header. As long as the virus does not encrypt the original header information at the end of the program, the multiple *Windows* headers can be detected. Additionally, the Import Table has to be checked for a combination of API imports. If there are KERNEL32.DLL imports for a combination of GetModuleHandle, Sleep, FindFirstFile, FindNextFile, MoveFile, GetWindowsDirectory, WinExec, DeleteFile, WriteFile, CreateFile, MoveFile, CreateProcess at the same time, then the application is likely to be infected with a prepending infector.

5.1.12 Suspicious relocations

This is a code-related flag. If the code contains instructions which can be used to determine the actual start address of the virus code, it should be flagged. For instance, if a CALL instruction is detected for the next possible offset, it is suspicious. Many Win95 viruses use the form of E80000's (CALL next address) 32-bit equivalent form E800000000 similar to DOS virus implementations.

5.1.13 Using KERNEL32 base address directly and looking For PE\0\0 mark

It is suspicious to detect code which operates with hard coded pointers to certain system areas, such as the KERNEL32.DLL or the VMM's memory area. Such viruses often search for the PE\0\0 mark at the same time in their code which should also be detected.

5.1.14 KERNEL32.DLL is Inconsistent

Last but not least, the consistency of KERNEL32.DLL can be checked by using one API from IMAGEHLP.DLL such as CheckSumMappedFile or MapFileAndCheckSum. This way, viruses which infect KERNEL32.DLL but do not recalculate the checksum field for it (such as Win95/Lorez, Win95/Yourn) can be detected easily.

5.1.15 Loading section into VMM

Unfortunately, it is possible to load a section into the Ring 0 memory area. The VMM memory area starts at address 0xC0001000. At 0xC0000000 there is a non used page. Since the VMM area is not protected from memory writes, *Microsoft* decided to put a non-used page as the first one to catch at least some applications which accidentally access this memory area. Win95/MarkJ.826 virus gets hold of this

unused page. The virus adds a new section header into the Section Table. This new section specifies the virtual address of the section which will point to 0xC0000000 of memory. The system loader will allocate this page automatically when the infected application is executed. This technique will be used for sure in many viruses in the future. The system loader could refuse this page allocation easily, but *Windows 9x*'s implementations do not contain such a feature. Therefore, it has to be considered suspicious when any section's virtual address points into the VMM area.

5.1.16 Incorrect size of code

Most viruses do not touch the SizeOfCode field of the PE header when adding a new executable section. If the recalculated size of all code sections is not the same as in the header, there is a chance that new executable sections have been patched into the executable.

5.2 EXAMPLES OF SUSPICIOUS FLAG COMBINATIONS

Let us look at some examples for the above flags in real viruses. All the outputs are generated by *Eureka!* (a heuristic detector which implements the above heuristic rule set and was developed by the author of this paper for future releases in anti virus products).

```
c:\winvirs\win32\CABANAS.VXE
-Execution starts in last section
-Suspicious code section characteristics
-Suspicious code redirection
→ Possibly infected with an unknown Win32 virus (Level: 5)

c:\winvirs\win95\ANXIETY.VXE
-Execution starts in last section
-Suspicious code section characteristics
-Virtual size is incorrect in header
-Suspicious code section name
→ Possibly infected with an unknown Win32 virus (Level: 6)

c:\winvirs\win95\MARBURG.VXE
-Execution starts in last section
-Suspicious code section characteristics
-Virtual size is incorrect in header
-Suspicious code redirection
-Suspicious code section name
→ Possibly infected with an unknown Win32 virus (Level: 8)

c:\winvirs\win95\SGWW2202.VXE
-Execution starts in last section
-Suspicious code section characteristics
-Virtual size is incorrect in header
-Suspicious relocation
-Suspicious code section name
-Using KERNEL32 address directly and looking for PE00
→ Possibly infected with an unknown Win32 virus (Level: 9)
```

6 CONCLUSION

The field of Win32 viruses is still in its very early stages. There are lots of things to discover in this new area of computer viruses for both virus writers and anti-virus researchers. As DOS is not the dominant

platform any more, viruses which spread on it are not dominant either. The evolution of computer viruses is a necessary bad tendency and virus writers know this well. Some anti-virus vendors, however, have not been able to recognize in time that the detection of DOS viruses is not enough to survive on the market and are redundant now because of the macro virus problem. The computer anti-virus field is changing rapidly and anti-virus products have to be improved against all the new trends. The *Windows* version of an anti-virus product should not be viewed as a simple porting issue any more, but has to be recognized as a major product release instead. Undoubtedly, Win32 viruses have the potential to become one of the major problems of the next century. It is time to prevent it!

7 BIBLIOGRAPHY

[1] Jeffrey Richter 'Advanced Windows NT', *Microsoft Press*, 1994.

[2] Matt Pietrek 'Windows 95 System Programming SECRETS', *IDG Books*, 1995.

[3] Andrew Schulman 'Unauthorized Windows 95', *IDG Books*, 1994.

[4] Eugene Kaspersky, personal communication.

[5] Bruce Schneier, 'Applied Cryptography', *John Wiley & Sons, Inc.*, 1996.

[6] Peter Ször 'HPS', *Virus Bulletin*, June 1998, pp. 11–13.

[7] Peter Ször 'High Anxiety', *Virus Bulletin*, January 1998, pp. 7–8.

[8] Peter Ször 'Coping with Cabanas', *Virus Bulletin*, November 1997, page 10–12.

[9] Peter Ször 'Junkie Memorial?', *Virus Bulletin*, September 1997, pp. 6–8.

[10] Peter Ször 'Lorez', *Virus Bulletin*, October 1998.

[11] Peter Ször 'Bad IDEA', *Virus Bulletin*, April 1998, pp. 18–19.

Geographical distribution of attacks by the HackTool.Win32.KMSAuto family. Geographical distribution of attacks during the period from 2 February 2017 to 2 February 2018. Top 10 countries with most attacked users (% of total attacks).