Introduction to various file infection techniques

An overview with some examples, written by ir3t

Table of contents

1. Introduction

1.1 About this paper

This paper was written within the scope of the author´s own learning process, during her studies of computer science and on the basis of her own malware analyses.

It was published in order to offer an introduction to several file infection techniques used by viruses, showing examples and explaining appropriate countermeasures.
It is aimed at everyone who is interested in the topic of malware and does not require previous knowledge, though a basic understanding of the PE file format, virtual memory management and the Windows OS in general is advantageous.

The author apologizes for grammar and/or contentual mistakes and is happy about any feedback. You can contact her by E- Mail (ir3t@freeunix.net).

1.2 Content and structure

In order to properly understand this paper, it is important to define the meaning of the word "virus" in the context of information technology. It will be explained within the chapter "Theoretical Foundations". The definition is followed by a raw overview over some tools used for the analysis and an introduction to the PE file format, which all viruses and infected files mentioned here refer to.

A general description of the analysis process is given in chapter 3.

The main part (chapters 4 to 7) contains excerpts of the written results which were acquired during the author's virus analyses, as well as general descriptions of the corresponding infection techniques.

Originally, these analyses did not focus on the infection techniques, but covered all the characteristics of the described viruses, including their payload, polymorphic engines etc.
To match this paper's topic, the descriptions were modified; parts of them were shortened or removed, while other details concerning the infection of files were emphasized more than before.

There are a lot of virus types like overwriting viruses, companion viruses and combinations of various techniques which this paper does not cover, therefore chapter 8, "Other file infection techniques" shortly describes some additional ways to infect files, without the intention to be exhaustive.

General thoughts about the experience and knowledge that was gained during the analyses are summarized within the chapter "Résumé".

Additionally, the appendix contains a list of all API (Application Programming Interface) functions which are mentioned in the context of this paper. The listed URLs refer to the function explanations at the Microsoft Developer Network (MSDN).

Short explanations of some technical terms are given within the glossary.

2. Theoretical Foundations

2.1 Definition of the term “Virus”

When people talk about “viruses”**,** they often refer to malware in general, i.e. programs which were written to serve any kind of (hidden) “malicious” purpose, such as stealing data, collecting user information, damage or delete files, slow down the user´s computer, manipulate or, in the worst case, destroy components of the operation system to make it unusable.

Although a virus can have one or all of the properties described above, none of them classifies it as a virus. In his book “The art of computer virus research and defense”, Peter Szor gives the following definition: *“A computer virus is code that recursively replicates a possibly evolved copy of itself. Viruses infect a host file or system area, or they simply modify a reference to such objects to take control and then multiply again to form new generations”.*

So the main feature which makes a virus a virus is its ability to replicate itself.
Although this paper only provides examples for file infections, there are also ways for viruses to replicate without making use of hostfiles, such as modifying the boot sector (which became rare today) or making use of system properties to execute malicious code instead of a certain program (companion viruses; see chapter 8).

Some viruses additionally have the ability of changing appearance (polymorphism) or even re- writing themselves (metamorphism).

Almost every “modern” virus has some features of, for example, a worm or a trojan, to be able to spread faster and more efficiently and/or to provide ways for other malicious programs to enter the system.

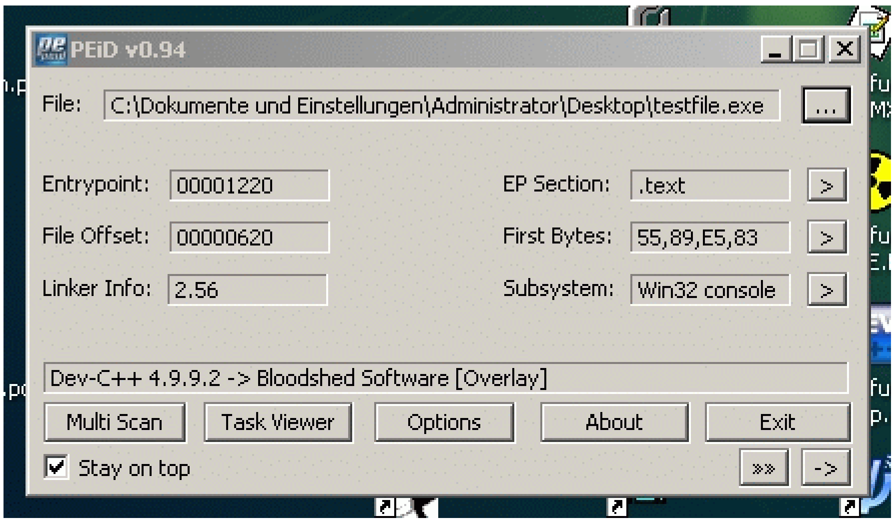
2.2 Software and tools

The following software and tools were used during the analyses:

- **VMWare:** VMWare was used to run and analyse the virus samples within the context of a virtual machine in order not to damage the “real” system.
- **OllyDbg:** OllyDbg is a well- known 32- bit debugger developed by Oleh Yushuk. It runs in Ring 3 mode and offers a lot of useful features such as setting hardware and software breakpoints, multi- thread- debugging, attachment to running processes and many more.
- **IDA (The Interactive Disassembler) Pro:** For static code analysis, IDA was used. It offers a lot of possibilities of automated code analysis for various file formats and platforms and can also be used as a debugger and decompiler when used with the Hex-Rays Plug- In.
- **Beyond Compare:** For examining the structure of files before and after the infection, a tool called “Beyond Compare” was used. It is able to compare folders and files in

various formats to each other and offers various additional options. For the purposes of this project, the hexadecimal comparison was the interesting feature.

- **PEiD:** A tool which quickly determines whether an executable is packed or not, which packer and packer version was used and, after unpacking, in which language it was written. There are also plug- ins which help to find out if common encryption algorithms were used and also the possibility to perform automatized unpacking on some well- known packers.
- **Process Monitor:** This tool combines the functions of Filemon and Regmon, two older tools. It is helpful for monitoring running processes as well as changes within the registry.
- **Process Explorer:** A tool which shows running processes, dependant dlls and opened handles. Like Process Monitor, it is part of the Windows Sysinternals.


PEiD v0.94

File: C:\Dokumente und Einstellungen\Administrator\Desktop\testfile.exe

Entrypoint: 00001220 | EP Section: .text

File Offset: 00000620 | First Bytes: 55,89,E5,83

Linker Info: 2.56 | Subsystem: Win32 console

Dev-C++ 4.9.9.2 -> Bloodshed Software [Overlay]

Image 1: The automatized analysis of a program with PEiD

2.3 Structure of the PE file format

PE stands for “Portable Executable”. The PE file format is used for executable files on 32- and 64- bit Windows platforms. Not only EXE, but also for example DLL, SYS and SCR files have the PE file structure, which is a modified version of the COFF (Common Object File) format. Also, the format can be used for object files.

Every portable executable starts with the MZ header and DOS stub. The first two bytes within the header are always “MZ”, followed by some fields containing additional information and the offset to the actual PE header.

In case the file is executed in DOS mode, instead of continuing at the PE header offset, the DOS stub will be ran. It is a small, independent application that will print out a sentence like “This program must be run under Win32” or “This program cannot be run in DOS mode”.

The PE header starts with a 4- byte- signature; it is mostly “PE\0\0” (in ASCII), but can also start with “N” (16- bit application), “L” (for Windows 3.x VxD´s) or “L” (file for OS/2 2.0).

The informations within the PE header are structured as they are within the COFF file format: Right after the PE signature, the following fields can be found[1]:

Size	**Field**	**Description**
2	Machine	The number that identifies the type of target machine.
2	NumberOfSections	The number of sections. This indicates the size of the section table, which immediately follows the headers.
4	TimeDateStamp	The low 32 bits of the number of seconds since 00:00 January 1, 1970 (a C run-time time_t value), that indicates when the file was created.
4	PointerToSymbolTable	The file offset of the COFF symbol table, or zero if no COFF symbol table is present. This value should be zero for an image because COFF debugging information is deprecated.
4	NumberOfSymbols	The number of entries in the symbol table. This data can be used to locate the string table, which immediately follows the symbol table. This value should be zero for an image because COFF debugging information is deprecated.
2	SizeOfOptionalHeader	The size of the optional header, which is required for executable files but not for object files. This value should be zero for an object file.
2	Characteristics	The flags that indicate the attributes of the file.

Now, the so- called "IMAGE_OPTIONAL_HEADER" struct follows, which is optional for object files (.bpl,.cpl,.dpl,.ocx...extensions), but not for image files.

The standard fields in COFF format are:

2	Magic	The unsigned integer that identifies the state of the image file. The most common number is 0x10B, which identifies it as a normal executable file. 0x107 identifies it as a ROM image, and 0x20B identifies it as a PE32+ executable.
1	MajorLinkerVersion	The linker major version number.
1	MinorLinkerVersion	The linker minor version number.
4	SizeOfCode	The size of the code (text) section, or the sum of all code sections if there are multiple sections.
4	SizeOfInitializedData	The size of the initialized data section, or the sum of all such sections if there are multiple data sections.
4	SizeOfUninitializedData	The size of the uninitialized data section (BSS), or the sum of all such sections if there are multiple BSS sections.
4	AddressOfEntryPoint	The address of the entry point relative to the image base when the executable file is loaded into memory. For program images, this is the starting address. For device drivers, this is the address of the initialization function. An entry point is optional for DLLs. When no entry point is present, this field must be zero.
4	BaseOfCode	The address that is relative to the image base of the beginning-of-code section when it is loaded into memory.
24	4	BaseOfData (this field does not exist in 64- bit PE files)

The next fields are Windows specific; they contain additional information needed by the linker and the PE loader. The 2nd number in the size field refers to PE32+ (64- bit) files.

4/8	ImageBase	The preferred address of the first byte of image when loaded into memory; must be a multiple of 64 K. The default for DLLs is 0x10000000. The default for Windows CE EXEs is 0x00010000. The default for Windows NT, Windows 2000, Windows XP, Windows 95, Windows 98, and Windows Me is 0x00400000.
4	SectionAlignment	The alignment (in bytes) of sections when they are loaded into memory. It must be greater than or equal to FileAlignment. The default is the page size for the architecture.
4	FileAlignment	The alignment factor (in bytes) that is used to align the raw data of sections in the image file. The value should be a power of 2 between 512 and 64 K, inclusive. The default is 512. If the SectionAlignment is less than the architecture's page size, then FileAlignment must match SectionAlignment.
2	MajorOperatingSystemVersion	The major version number of the required operating system.
2	MinorOperatingSystemVersion	The minor version number of the required operating system.
2	MajorImageVersion	The major version number of the image.
2	MinorImageVersion	The minor version number of the image.
2	MajorSubsystemVersion	The major version number of the subsystem.
2	MinorSubsystemVersion	The minor version number of the subsystem.
4	Win32VersionValue	Reserved, must be zero.
4	SizeOfImage	The size (in bytes) of the image, including all headers, as the image is loaded in memory. It must be a multiple of SectionAlignment.
4	SizeOfHeaders	The combined size of an MS-DOS stub, PE header, and section headers rounded up to a multiple of FileAlignment.
4	CheckSum	The image file checksum. The algorithm for computing the checksum is incorporated into IMAGHELP.DLL. The following are checked for validation at load time: all drivers, any DLL loaded at boot time, and any DLL that is loaded into a critical Windows process.
2	Subsystem	The subsystem that is required to run this image.
2	DllCharacteristics	DllCharacteristics
4/8	SizeOfStackReserve	The size of the stack to reserve. Only SizeOfStackCommit is committed; the rest is made available one page at a time until the reserve size is reached.
4/8	SizeOfStackCommit	The size of the stack to commit.
4/8	SizeOfHeapReserve	The size of the local heap space to reserve. Only SizeOfHeapCommit is committed; the rest is made available one page at a time until the reserve size is reached.
4/8	SizeOfHeapCommit	The size of the local heap space to commit.
4	LoaderFlags	Reserved, must be zero.
4	NumberOfRvaAndSizes	The number of data-directory entries in the remainder of the optional header. Each describes a location and size.

The last part of the optional header holds an array of structs like this:

```
typedef struct _IMAGE_DATA_DIRECTORY {
        DWORD VirtualAddress;
        DWORD Size;
} IMAGE_DATA_DIRECTORY,*PIMAGE_DATA_DIRECTORY;
```

Some of the elements of this array are, for example, the Import Table, the Resource Table and the Relocation Table. Explaining the purpose and structure of single every element would go beyond the scope and is not necessary for the understanding of this paper.

The section table, which holds an entry for every section of the file, has no entry within this array; its location is always determined as “the first byte behind the header”.

The table is an array again, and every single entry is a struct:

```
typedef struct _IMAGE_SECTION_HEADER {
        BYTE Name[IMAGE_SIZEOF_SHORT_NAME];
        union {
                DWORD PhysicalAddress;
                DWORD VirtualSize;
        } Misc;
        DWORD VirtualAddress;
        DWORD SizeOfRawData;
        DWORD PointerToRawData;
        DWORD PointerToRelocations;
        DWORD PointerToLinenumbers;
        WORD NumberOfRelocations;
        WORD NumberOfLinenumbers;
        DWORD Characteristics;
} IMAGE_SECTION_HEADER,*PIMAGE_SECTION_HEADER
```

These are the field´s sizes and meanings:

4	VirtualSize	The total size of the section when loaded into memory. If this value is greater than SizeOfRawData, the section is zero-padded. This field is valid only for executable images and should be set to zero for object files.
4	VirtualAddress	For executable images, the address of the first byte of the section relative to the image base when the section is loaded into memory. For object files, this field is the address of the first byte before relocation is applied; for simplicity, compilers should set this to zero. Otherwise, it is an arbitrary value that is subtracted from offsets during relocation.
4	SizeOfRawData	The size of the section (for object files) or the size of the initialized data on disk (for image files). For executable images, this must be a multiple of FileAlignment from the optional header. If this is less than VirtualSize, the remainder of the section is zero-filled. Because the SizeOfRawData field is rounded but the VirtualSize field is not, it is possible for SizeOfRawData to be greater than VirtualSize as well. When a section contains only uninitialized data, this field should be zero.
4	PointerToRawData	The file pointer to the first page of the section within the COFF file. For executable images, this must be a multiple of FileAlignment from the optional header. For object files, the value should be aligned on a 4-byte boundary for best performance. When a section contains only uninitialized data, this field should be zero.
4	PointerToRelocations	The file pointer to the beginning of relocation entries for the section. This is set to zero for executable images or if there are no relocations.
4	PointerToLinenumbers	The file pointer to the beginning of line-number entries for the section. This is set to zero if there are no COFF line numbers. This value should be zero for an image because COFF debugging information is deprecated.
2	NumberOfRelocations	The number of relocation entries for the section. This is set to zero for executable images.
2	NumberOfLinenumbers	The number of line-number entries for the section. This value should be zero for an image because COFF debugging information is deprecated.
4	Characteristics	The flags that describe the characteristics of the section.

The number of sections within a PE file can vary, as well as their names and purposes. However, there must be at least one section, and some section names are reserved for special purposes, as specified within the *Microsoft Portable Executable and Common Object File Format Specification* (see bibliography).

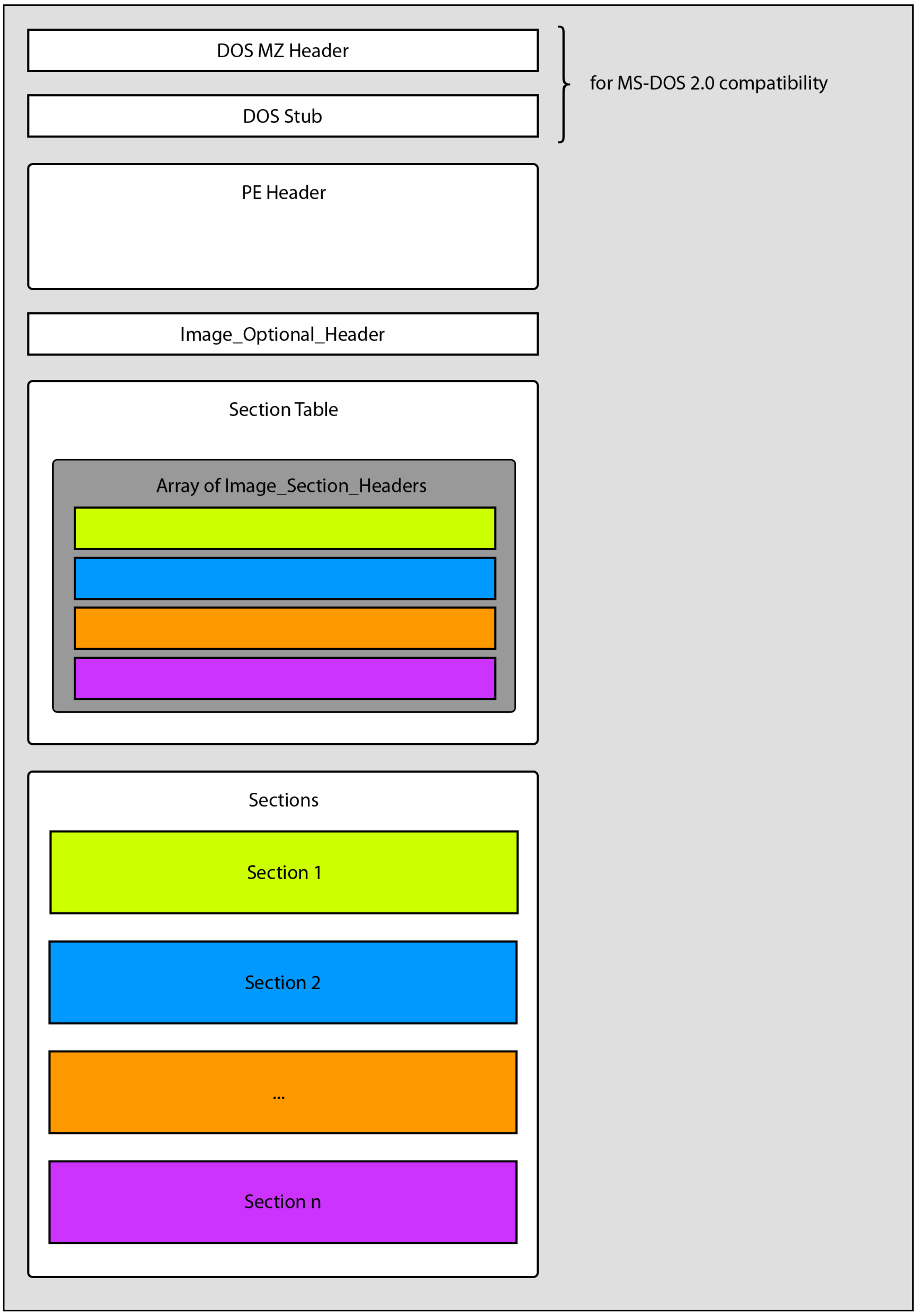


Image 2: The PE file structure

3. General virus analysis procedure

At first, the suspicious file needs to be scanned by a virus scanner to see if there is already an entry within the respective virus database.

If the file is not known already, the second step is to determine whether the file is packed or not. If it is, it can either be packed by a known or a custom- made packer. Known packers can in most cases be unpacked automatically, while other ones have to be unpacked manually.

The original entry point has to be found, the imports have to be rebuilt and the file has to be dumped. In case of self- extracting archives/installers, the malicious file needs to be extracted first.

After the extraction or unpacking of the virus, the actual analysis begins.

The static part of the analysis should be done first, using tools like a hex viewer and a disassembler to get a general overview over the file´s structure, used API functions, text strings, compiler information and so on.

The dynamic analysis with monitoring tools and debuggers takes more time than the static one. Also, most virus authors use anti- debugging- tricks to make the analysis more difficult and to slow it down.

Due to the fact that in most cases, the virus code has to be executed several times, the opportunity to create so- called “snapshots” of the current system state within virtual machines is very helpful during the analysis process.
This way, a previous state can be re-stored quickly, and the virus code can be executed as often as necessary.

After finishing the analysis, the sample has to be named.

While almost every AV company uses its own naming scheme (which sometimes causes confusion), the CARO Malware Naming Scheme is a common naming standard towards which most of them are oriented.

The common version of their virus naming convention was created in 1991 at a meeting of the *Computer Anti-Virus Researchers Organization* (CARO).

It looks like this:

```
Family_Name.Group_Name.Major_Variant.Minor_Variant[:Modifier]
```

There is a guideline on how to choose the single name components (see the linklist for more information). Still, there is no official database with the “CARO names” of viruses.

Within the scope of this paper, some aliases of the used samples are listed at the beginnings of the respective subchapters.

4. Appending Viruses

4.1 Overview

As the name says, an appending virus appends itself behind the original file.
The virus code is written to the end of the last section or a new section is created, as the analysis of *Evyl* will show.

There are different variants of how an appending virus code can gain control; it can either happen by a jump or a combined push/ret instruction which is placed right in front of the host file´s code and which leads to the beginning of the malicious code, or by manipulating the AddressOfEntryPoint within the file´s PE header in a way that the code execution starts with the viruse´s main function.
Later on, there is a jump or push/ret back to the original code, which then proceeds in a normal way.

For virus scanners, the difficulty of finding appending file infectors is in most cases higher than detecting overwriting or prepending viruses and may increase a lot if the virus is polymorphic (or even metamorphic).

Disinfection is mostly no problem, unless the original code was damaged during the infection process.

4.2 Example: Evyl

Some aliases: Win32.Evyl.b, W32/Evyl.b.intd, W95.Evyl,Win32.Evyl.3151, W32/Evyl-B

4.2.1 Description

Evyl is an appending virus. It was not packed and first detected in 2001.

Besides the infection of files which is described below, the virus contains a payload:
By creating and changing some registry entries, the refresh rate of the infected computer´s monitor is increased to 160 Hz. One can imagine that this caused some monitors to power off.

4.2.2 Infection Technique

Evyl scans the drives C:\, D:\ and E:\, as well as all subfolders, for files to infect.
Only files with an ".exe" extension and an MZ and PE header will be infected. They are opened with *ReadFile*, the position from which to read is defined by *SetFilePointer*.

One byte of the *TimeDateStamp* is compared to a static value to avoid double infections (it is changed to this value after the infection).

Evyl appends itself behind the hostfile´s code, creating a new section named "virus32**". It creates a new import table within this section and adjusts the import table address within the header as well: The new address of the import table is the beginning of the virus32** section + 710h.

The *AddressOfEntryPoint* within the header is set to the beginning of the viruse´s main function. The hostfile´s *checksum* is set to 0; the virus author might have thought that this would look less obvious than leaving the newly created one.

After loading three dlls (*LoadLibrary*) and retrieving some functions needed by the virus (using *GetProcAddress*) whose addresses are then stored within the new import table.

Due to the fact that the virus changed the Import Table Address within the header to the address of its own import table, the addresses of the API functions needed by the hostfile cannot be resolved automatically by the PE loader by their names or ordinals.

Therefore, *Evyl* has to assume the role of the PE loader:
Using the address of the “old” import table which was stored within the virus code, the dlls noted within the host´s import directory table are loaded using *LoadLibrary*. Every single entry within the IAT is replaced by the respective VA, using the import lookup table and *GetProcAddress*.

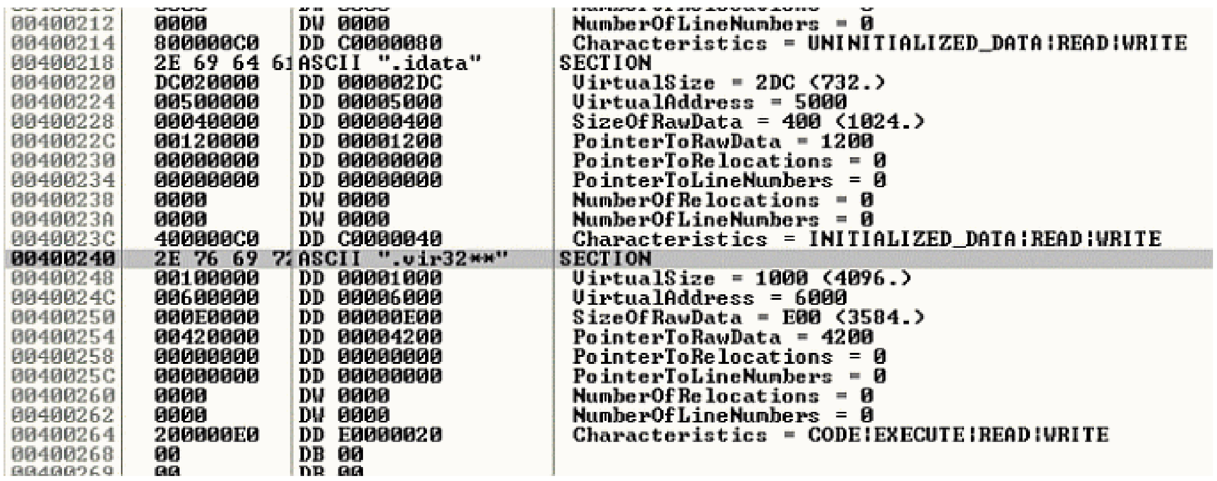
CreateThread is called to create an own thread for the rest of the virus code (which is the payload function à see 5.2.1), followed by a JMP to the file´s OEP. This way, the viruse´s code can be executed parallel to the infected file´s original code.

4.2.3 Disinfection

Infected files can easily be spotted by searching for the "virus32**" string within the header´s *section table* (see image 3). The file´s original entry point can be found within the first part of the virus code, as well as the offset to the import table.

The corresponding fields in the header have to be restored and the virus code section must be deleted; its size is always the same and can be retrieved from the header.

The added/changed registry values must be re- changed manually.



```
00400212  0000         DW 0000            NumberOfLineNumbers = 0
00400214  800000C0     DD C0000080        Characteristics = UNINITIALIZED_DATA|READ|WRITE
00400218  2E 69 64 61  ASCII ".idata"    SECTION
00400220  DC020000     DD 000002DC        VirtualSize = 2DC (732.)
00400224  00500000     DD 00005000        VirtualAddress = 5000
00400228  00040000     DD 00000400        SizeOfRawData = 400 (1024.)
0040022C  00120000     DD 00001200        PointerToRawData = 1200
00400230  00000000     DD 00000000        PointerToRelocations = 0
00400234  00000000     DD 00000000        PointerToLineNumbers = 0
00400238  0000         DW 0000            NumberOfRelocations = 0
0040023A  0000         DW 0000            NumberOfLineNumbers = 0
0040023C  400000C0     DD C0000040        Characteristics = INITIALIZED_DATA|READ|WRITE
00400240  2E 76 69 72  ASCII ".vir32**"  SECTION
00400248  00100000     DD 00001000        VirtualSize = 1000 (4096.)
0040024C  00600000     DD 00006000        VirtualAddress = 6000
00400250  000E0000     DD 00000E00        SizeOfRawData = E00 (3584.)
00400254  00420000     DD 00004200        PointerToRawData = 4200
00400258  00000000     DD 00000000        PointerToRelocations = 0
0040025C  00000000     DD 00000000        PointerToLineNumbers = 0
00400260  0000         DW 0000            NumberOfRelocations = 0
00400262  0000         DW 0000            NumberOfLineNumbers = 0
00400264  200000E0     DD E0000020        Characteristics = CODE|EXECUTE|READ|WRITE
00400268  00           DB 00
00400269  00           DB 00
```

*Image 3: The .vir32** section created by Evyl*

5. Prepending viruses

5.1 Overview

A prepending virus writes itself in front of a host file, so that the virus code is executed first. After the execution of the virus code, the control is passed to the original program again.

Prepending viruses are not too hard to create but yet very effective, because they are executed at all events.

In former times, when .COM files were more popular, prepending viruses were even easier to implement, because the COM file format does not consist of a header, but only of code and data. Nowadays, a prepending virus needs its own DOS and PE header as an independent program and thus cannot easily re- pass control to the original program. That is why it is necessary to work with temporary files to pass control to the host program after the malicious code´s execution.

A common method to do this is to create a file which holds the host program´s code only. This file is most likely to be found within the same directory as the infected one, with a different name and/or extension. It is executed by the virus and deleted as soon as the user quits the program.

Another interesting method is shown within the following subchapter using the example of *Whboy*, where the malicious code is completely removed from the host file on hard disk after execution.

In most cases, prepending viruses are rather easy to spot by virus scanners due to their obvious presence at the beginning of a file; also, they increase the file size.
There is a good chance to clean infected files: If the viruse´s size is known, a certain number of bytes can be just removed from the file. Additionally, it is important to check the “new” first bytes of the disinfected file for a correct file signature, depending on the file format (such

as the “MZ” of the DOS header). The last bytes of the virus code can be checked for known byte sequences as well to avoid deleting parts of the host.

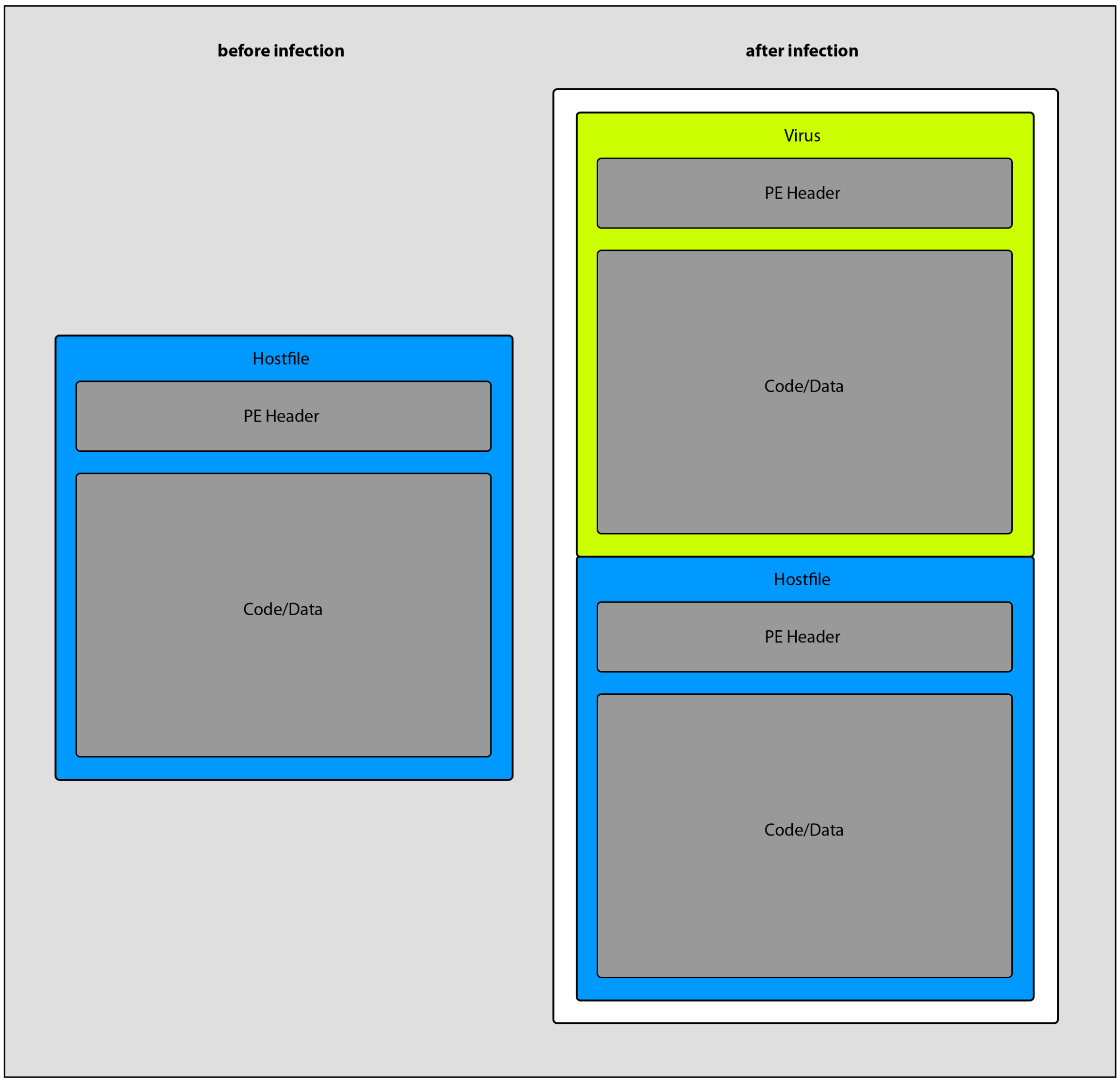


Image 4: Prepending infection technique

5.2 Example: Whboy

Some aliases: TR/Rootkit.Gen, Win32.HLLP.Whboy.115, Worm.Win32.Fujack.da, Virus.Win32.Viking

5.2.1 Description

Whboy first appeared in the wild[2] in 2007. It was packed with UPX and written in Delphi.

Besides its file infection capabilities, the virus also owns some characteristics of a worm, spreading through the local network under the names “Setup.exe” and “Cool_GameSetup.exe”.
In this context, it also tries to get access to the admin$ share of remote computers, using its own (initially encrypted) password list.

The virus contains too many functionalities to describe them all within the scope of this paper, such as creating a huge number of registry entries for several purposes as well as creating/writing some additional files and connecting to 2 URLs to download more malware.

One important functionality to mention is the creation of a file named JM.sys within the drivers folder , which itself is a trojan. The registry entry referring to the driver of the Dmusic Service of Windows is changed in a way that instead of the original driver, JM.sys will be run. Its purpose is to detect and shut down anti- viruses.

Whboy copies itself into the drivers folder under the name TXP1atform.exe.

5.2.2 Infection Technique

Whboy is a prepending file infector which increases the infected file´s size by ca. 78 kb (size of the packed virus code). It infects files with .scr and .exe extension (.scr stands for “Screensaver”, but in fact, those files are in PE file format, too).

Additionally, the virus tries to infect htm(l) files by inserting some content it downloads before (by the time this analysis was done, the respective content was already removed from the internet).

Whboy contains an encrypted list of files, which gets decrypted during the execution. The files in this list (mostly system files) will not be infected; additionally, running processes are skipped, in order that the virus does not catch attention immediately.

If a file to infect (“a.exe” in the following) was found, it is mapped into the viruse´s address space using the APIs *VirtualAlloc* to allocate memory, *CreateFileMappin*g to create a file mapping object and, finally, *MapViewOfFile* to map the created object.

CopyFile is used to replace “a.exe” by the virus.

There is an interesting detail about this infector: *Whboy* extracts the host file´s icon (*ExtractIcon*) from its mapped view and stores it within %temp%.

“a.exe” (which is now the virus) is opened in write mode (*CreateFile* with value *FILE_SHARE_WRITE*), the file pointer is set to the position of the icon data (*SetFilePointer*), which now gets overwritten with the host file´s icon.

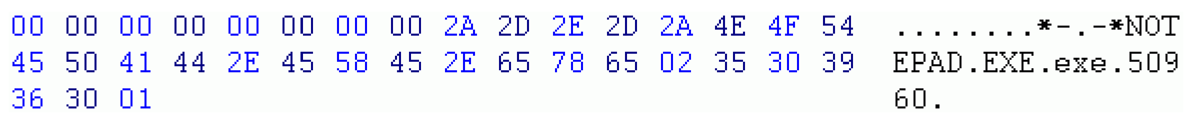
Then the file pointer is set to the end of “a.exe”” and the original code of the hostfile is appended in portions of 80 bytes from a buffer (*WriteFile*). Now the handle to the file is closed, and the infection is done.

If an infected file (“a.exe) is launched, the virus code will be executed; the virus then creates a batch file that looks like this:

```
:try1
del "C:\Documents and Settings\admin\Desktop\notepad.exe"
if exist "C:\Documents and Settings\admin\Desktop\notepad.exe" goto try1
ren "C:\Documents and Settings\admin\Desktop\notepad.exe.exe" "notepad.exe"
if exist "C:\Documents and Settings\admin\Desktop\notepad.exe.exe" goto try2
"C:\Documents and Settings\admin\Desktop\notepad.exe"  //execute notepad.exe
:try2
del %0 //delete the batch file
```

As soon as the infected file is ended, it will be deleted. The newly created “clean” file will be renamed and executed. The batch file is deleted as well.

There is a kind of signature *Whboy* appends to the end of infected files which looks like this: **-.-*filename.exe.exe.number* (see Image 5).
The appended number is the host file´s size in bytes; it serves as a check for correctness for the virus when creating the clean file with the double .exe extension.



```
00 00 00 00 00 00 00 00 2A 2D 2E 2D 2A 4E 4F 54  ........*-.-*NOT
45 50 41 44 2E 45 58 45 2E 65 78 65 02 35 30 39  EPAD.EXE.exe.509
36 30 01                                         60.
```

Image 5: Signature created by Win32.HLLP.Whboy.115 at the end of an infected file

5.2.3 Disinfection

The following steps do not only refer to the disinfection of one file, but to the disinfection of the whole system.

- Remove TXP1atform.exe from the drivers folder
- Remove JM.sys
- Delete all the created registry values and reset the changed ones
- Select a specific, unique byte sequence within the packed virus code to scan for in all exe files on all drives (maybe specify a second one to make sure you really found the virus code). If found, delete the viruscode (to be sure, check its ending for the string “1235C”, which is the offset to the viruse´s icon data saved as string, and delete everything including the “C”)
- Scan all drives for htm(l) files and search for inserted content; remove it
- Empty the temp folder
- Search for all known additional files that could have been downloaded and remove them

6. Cavity Viruses

6.1 Overview

Every single struct within the section table of a PE file has a *PointerToRawData* field which points to the beginning of “used” space within a section (see 2.3). The end of the data (as specified in *SizeOfRawData*) does not have to be equal to the end of the section. This means that there may be unused space at the end of a section.

A cavity virus makes use of this circumstance by writing parts of itself within these spaces as well as into other unused bytes within the file, trying to prevent an infected file´s size from increasing and thus making it more difficult to determine the infection.

Some cavity viruses, do not only make use of free space within sections, but also overwrite the relocation section with parts of their code. This is possible without harming the functionality of the host program, because nowadays, the informations stored in “.reloc” are seldom if never needed[3].

Cavity viruses are generally not easy to spot. The filesize does only increase if there are not enough free bytes available, and the whole file has to be scanned. During the infection process, every single piece of virus code, scattered within the hostfile, has to be found and removed without damaging the hostfile. If a cavity virus owns an encryption engine, the difficulty even increases.

6.2 Example: Elkern v1.1

Some aliases: WORM/Klez.E,Win32.Klez.4219,Virus.Win32.Elkern.b
(**Some aliases of this Klez version:** WORM/Klez.E, Win32.HLLM.Klez.1, Email-Worm.Win32.Klez.k)

6.2.1 Description

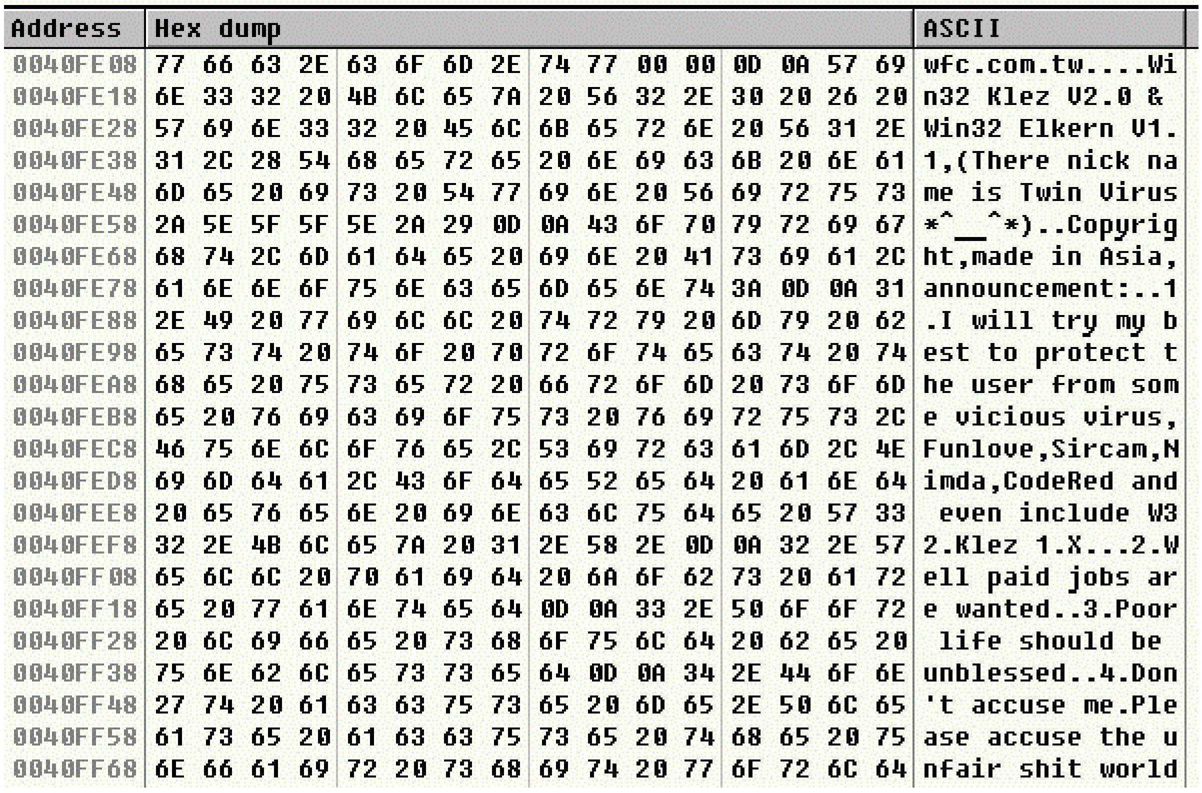
Elkern is dropped by a (famous) mail worm named Klez. The mailworm first appeared in 2001; it exploited a security hole in both Outlook and Outlook Express which enabled it to execute automatically from an e-mail-attachment. It spread very fast and efficiently. There are three versions of the virus and about twelve versions of the mailworm.

Klez carries Elkern in encrypted form within its body; on execution, Elkern´s code is decrypted, and the virus is dropped as “Wqk.exe” into the System folder (Win 9x) or as “Wqk.dll” into the system32 folder (Win 2000/XP).

Under Win XP, the created dll is supposed to run within the context of Klez (using *CreateRemoteThread and LoadLibrary*). Elkern uses 32- bit checksums for the API functions it needs to call, but only compares 16 bits; that is why, on a call to *IsDebuggerPresent*, the dll crashes.

Wqk.exe is started by a call to *CreateProcess*.
For both the dll and the exe file, registry entries are created to make them start up with the system.

Elkern contains the following payload: *GetSystemTime* is called and afterwards, there is a comparison to 0D (13 decimal). If the day is the 13th of the month, comparisons to 3 (March) and 9 (September) follow. If one of them is equal to the current month, all files on all hard disks (found out by calling *GetDriveTypeA* with the letters from A- Z as parameters) are overwritten with Zeroes.



```
Address   Hex dump                                         ASCII
0040FE08  77 66 63 2E 63 6F 6D 2E 74 77 00 00 0D 0A 57 69  wfc.com.tw....Wi
0040FE18  6E 33 32 20 4B 6C 65 7A 20 56 32 2E 30 20 26 20  n32 Klez V2.0 & 
0040FE28  57 69 6E 33 32 20 45 6C 6B 65 72 6E 20 56 31 2E  Win32 Elkern V1.
0040FE38  31 2C 28 54 68 65 72 65 20 6E 69 63 6B 20 6E 61  1,(There nick na
0040FE48  6D 65 20 69 73 20 54 77 69 6E 20 56 69 72 75 73  me is Twin Virus
0040FE58  2A 5E 5F 5F 5E 2A 29 0D 0A 43 6F 70 79 72 69 67  *^__^*)..Copyrig
0040FE68  68 74 2C 6D 61 64 65 20 69 6E 20 41 73 69 61 2C  ht,made in Asia,
0040FE78  61 6E 6E 6F 75 6E 63 65 6D 65 6E 74 3A 0D 0A 31  announcement:..1
0040FE88  2E 49 20 77 69 6C 6C 20 74 72 79 20 6D 79 20 62  .I will try my b
0040FE98  65 73 74 20 74 6F 20 70 72 6F 74 65 63 74 20 74  est to protect t
0040FEA8  68 65 20 75 73 65 72 20 66 72 6F 6D 20 73 6F 6D  he user from som
0040FEB8  65 20 76 69 63 69 6F 75 73 20 76 69 72 75 73 2C  e vicious virus,
0040FEC8  46 75 6E 6C 6F 76 65 2C 53 69 72 63 61 6D 2C 4E  Funlove,Sircam,N
0040FED8  69 6D 64 61 2C 43 6F 64 65 52 65 64 20 61 6E 64  imda,CodeRed and
0040FEE8  20 65 76 65 6E 20 69 6E 63 6C 75 64 65 20 57 33   even include W3
0040FEF8  32 2E 4B 6C 65 7A 20 31 2E 58 2E 0D 0A 32 2E 57  2.Klez 1.X...2.W
0040FF08  65 6C 6C 20 70 61 69 64 20 6A 6F 62 73 20 61 72  ell paid jobs ar
0040FF18  65 20 77 61 6E 74 65 64 0D 0A 33 2E 50 6F 6F 72  e wanted..3.Poor
0040FF28  20 6C 69 66 65 20 73 68 6F 75 6C 64 20 62 65 20   life should be 
0040FF38  75 6E 62 6C 65 73 73 65 64 0D 0A 34 2E 44 6F 6E  unblessed..4.Don
0040FF48  27 74 20 61 63 63 75 73 65 20 6D 65 2E 50 6C 65  't accuse me.Ple
0040FF58  61 73 65 20 61 63 63 75 73 65 20 74 68 65 20 75  ase accuse the u
0040FF68  6E 66 61 69 72 20 73 68 69 74 20 77 6F 72 6C 64  nfair shit world
```

Image 6: Text found within the .data section of Klez

6.2.2 Infection Technique

The file to be infected is mapped within the viruse´s memory space using *CreateFileMapping/MapViewOfFile.*

The file´s original entrypoint is looked up within its header to figure out in which section the code execution starts (comparisons between the entrypoint and every section´s VA (+*SizeOfRawData*)).

If the right section and entrypoint were found, the 2 bytes right before it are checked for the string “WQ”.
If the string is not there, it means that the file was not infected yet, and the infection begins.

Starting from the first section, the virus checks whether there is free space at the end of a section by comparing the value of *SizeOfRawData* to the value of *VirtualSize*.

If VirtualSize is smaller, the number of free bytes within the section is calculated.

If there is not enough space for the first 278h bytes of virus code (in fact, it must be 27dh: 5 additional bytes for the call to the decryption function), the next section will be checked (and the next one and so on). If there is no data at all within a section, it won´t be used for storing virus code either.

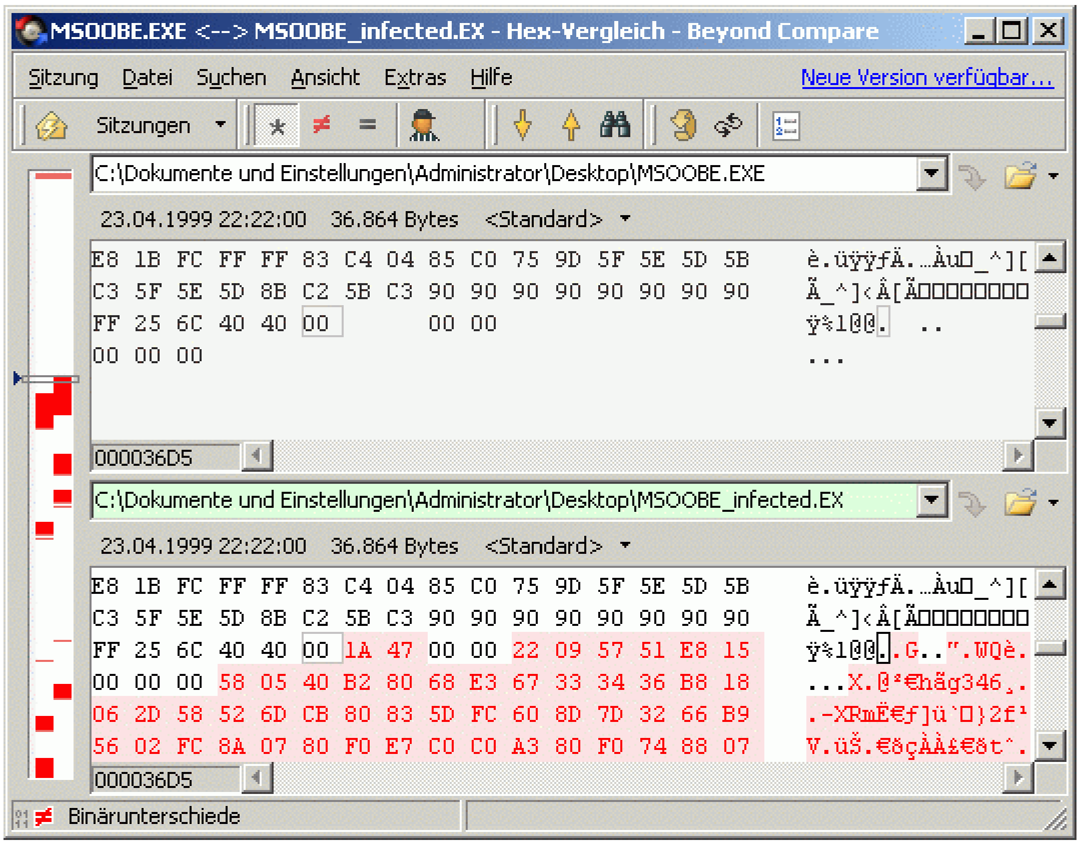
In case that none of the sections provide enough space, the file size will increase, because the 27dh bytes will be appended behind the end of data within the last section.

8 additional bytes are reserved in front of every piece of virus code which will hold the virtual address to the next piece of virus code and the size of the current piece; that is necessary for the “self-re-assembling” of the viruscode when executing the infected file.

The offset to the original entry point of the infected file is stored within the virus code; when running the infected file, the value will be added to the file´s imagebase and there will be a jump to the file´s OEP after the virus code´s execution.

The new entrypoint is the location of the first piece of virus code + 8 bytes.
The section which holds the new entrypoint is set to RWE[4].

The virus code gets encrypted before it is placed between the sections and the appendant decryption routine is re-newed within the code as well.
The checksum is recalculated and the timestamp is preserved.


MSOOBE.EXE <--> MSOOBE_infected.EX - Hex-Vergleich - Beyond Compare
Sitzung Datei Suchen Ansicht Extras Hilfe
Neue Version verfügbar...
Sitzungen
Binärunterschiede

```
C:\Dokumente und Einstellungen\Administrator\Desktop\MSOOBE.EXE
23.04.1999 22:22:00   36.864 Bytes   <Standard>

E8 1B FC FF FF 83 C4 04 85 C0 75 9D 5F 5E 5D 5B    è.üÿÿƒÄ.…ÀuD_^][
C3 5F 5E 5D 8B C2 5B C3 90 90 90 90 90 90 90 90    Ã_^]‹Â[ÃDDDDDDDD
FF 25 6C 40 40 00       00 00                      ÿ%l@@.  ..
00 00 00                                           ...
000036D5

C:\Dokumente und Einstellungen\Administrator\Desktop\MSOOBE_infected.EX
23.04.1999 22:22:00   36.864 Bytes   <Standard>

E8 1B FC FF FF 83 C4 04 85 C0 75 9D 5F 5E 5D 5B    è.üÿÿƒÄ.…ÀuD_^][
C3 5F 5E 5D 8B C2 5B C3 90 90 90 90 90 90 90 90    Ã_^]‹Â[ÃDDDDDDDD
FF 25 6C 40 40 00 1A 47 00 00 22 09 57 51 E8 15    ÿ%l@@..G..".WQè.
00 00 00 58 05 40 B2 80 68 E3 67 33 34 36 B8 18    ...X.@²€hãg346¸.
06 2D 58 52 6D CB 80 83 5D FC 60 8D 7D 32 66 B9    .-XRmË€ƒ]ü`D}2f¹
56 02 FC 8A 07 80 F0 E7 C0 C0 A3 80 F0 74 88 07    V.üŠ.€ðçÀÀ£€ðtˆ.
000036D5
```

Image 7: Beginning of the virus code with “WQ” signature at the end of a section

6.2.3 Disinfection

The following is a step- by- step explanation of how to disinfect a file infected by Elkern:

- Go to the entrypoint of the suspicious file and check whether the 2 bytes before it are 51 57 (“WQ”) and if at the entry point itself, there is a call (which start with E8). If both conditions are true, the file is probably infected
- Execute the code until the end of the decryption routine (the first 42 bytes from the entrypoint)
- Look up & store the program´s OEP;starting at the instruction “POP EBP” within the function, go 381 bytes further and read the next 2 bytes (they contain the offset to the OEP)
- Check the 2 Bytes before WQ to find out the first virus code piece´s length
- To find the next part, add the 4 bytes at (“WQ”-6) to the imagebase; the address you land at contains the length of this 2nd piece of code, the 4 bytes right in front contain the next offset and so on...
- Repeat this until there is no more offset to be found; now you have the offsets to all the pieces of virus code and their length, and you can delete those pieces
- Replace the OEP (*AddressOfEntryPoint* field) and correct the other changed header fields (*BaseOfCode, SizeOfData* ...); recalculate the checksum.

7. EPO viruses

7.1 Overview

EPO stands for "Entry Point Obscuring". It describes a type of virus which manipulates the hostfile code in a way that the virus code is executed.

For example, a call to an API function or a function belonging to the host could be overwritten with a call to the virus code. Another (possibly less successful) method would be to insert a jump to the virus code randomly.

The malicious code is then executed, and during the execution, the formerly patched instruction is restored. Then the control is passed to the host again, which runs in the regular way.

The virus code itself can either be appended to a file or overwrite a certain section, as the *CTX Phage Virus* in the following example does. Those two methods are the most common ones, although one could also think of a combination of, for example, cavity and EPO technique.

EPO viruses are difficult to develop; they need a routine to scan for a suitable instruction to modify the code, and there is a chance that the instruction they chose is not executed at all when running the host.

On the other hand, EPO viruses are also difficult to detect. Additionally, most of them own an encryption/decryption engine. When disinfecting a file, one does not only have to remove the virus code; the manipulated instruction has to be found and changed back as well, otherwise, the disinfected program could crash.

7.2 Example: CTX Phage

Some aliases: W32/CTX,Win32.CTX.6889,Virus.Win32.CTX.6886

7.2.1 Description

CTX Phage was created in 1999. It was written in assembly language by GriYo, a member of the so-called *29A group* as part of the *Simbiosis Project* and spread together with a worm known as *Cholera*.
It contains its own polymorphic engine which makes it difficult to detect.

The virus contains the following payload:

After retrieving the current date using *GetLocalTime*, month, date and hour are compared to the viruse´s creation date. If an infected file is executed exactly 6 months after its infection (it must be the same hour as well), the desktop colours are inverted. The payload routine runs in a loop as long as it is still the same hour and the file infection described in the next part only takes place if the loop is not taken (anymore).

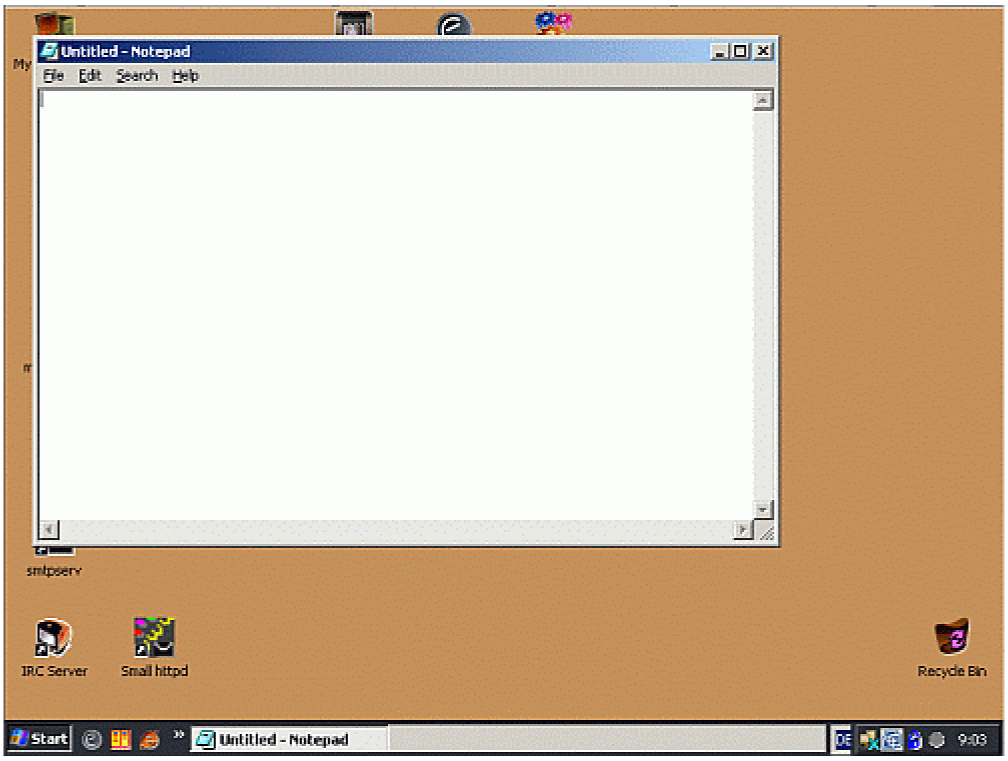


Image 8: Desktop colour invertion on Win XP caused by an infected Notepad.exe

7.2.2 Infection Technique

As well as *Evyl* (see 4.2), *CTX Phage* is an appending virus; but in this case, the entry point of the infected file is not manipulated.
Instead, a call to a library function somewhere within the code is patched by the virus in a way that the viruse´s decryption function is called.

The virus code is executed and decrypted and later on, the patched function call will be restored as soon as the host program regains control (*GetCurrentProcess/WriteProcessMemory*). The former function call can be retrieved from the virus code, where it was stored during the infection.

CTX Phage searches for the address of the *GetProcAddress* function of kernel32.dll; this is done by calculating a checksum which is compared to another one stored within the virus code.

Now, checksums are calculated for every function of kernel32, using the function names within the export name table.If one of those sums is equal to an entry within the viruse´s own checksum table, *GetProcAddress* is used to retrieve the function´s address.
This is done for 23 API functions (*CreateFileA*, *CreateFileMappingA*, *FindCloseChangeNotification*, *FindClose*, *FindFirstFileA*, *FindNextFileA*, *FreeLibrary*...).

VirtualAlloc is used to allocate new memory within the infected program´s address space. After copying the complete virus code, there´s a JMP right into it (shortly after this, the patched call will be restored).

After passing the payload routine, the file infection starts:
The current directory is the first one that will be searched for files to infect (*GetCurrentDirectory*); later on, the windows directory and system directory are affected as well (*GetWindowsDirectory, GetSystemDirectory*).
In each directory, no more than 5 files will be infected.
Only files with an *.exe extension can be infected. Some conditions within the WIN32_FIND_DATA structure[5] (whose information was formerly retrieved by calling *FindFirstFile/FindNextFile*) of the found files are checked:

- An AND operation of the file attributes with 814 must return 0
- The filesize must be smaller than 4GB
- The size of the file modulo 65h has to be 0. [6]

If those conditions match, the filename is checksummed as well and compared to another list of checksums, to avoid infecting certain programs.

Also, there is a call to *SfcIsFileProtected* to avoid infecting files that are protected by the *system file checker*.

Now, once the file got mapped into memory (*MapViewOfFile*), there are more checks, like for the DOS and the PE header, the value of the IMAGE_FILE_MACHINE (*Machine*) field etc.

Then, the copy of the virus code is either written right into the .reloc section, overwriting its previous content and filling the rest of it with zeroes, or appended to whatever section is the last one.

Image 8 shows the infection without overwriting .reloc, which is the more frequent case.

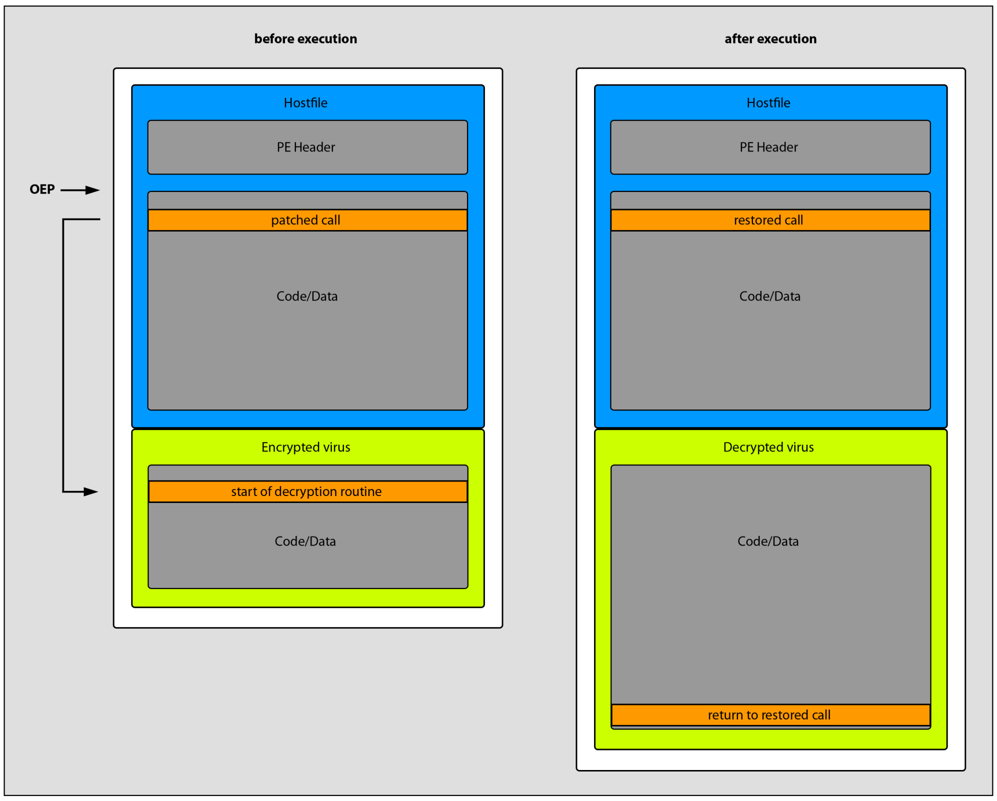


Image 8: EPO Technique of CTX Phage

Now the virus detects an API call to patch: Starting at the OEP of the hostfile, every byte is compared to E8 (CALL..) and FF15 (CALL DWORD...) and then the virus examines whether it is a call to the import table or not (it looks up the Import Table Address + its size within the header before and determines whether the call goes to an address in- between).
It does not just have to be a call to an API function, but it must be a call to kernel32.dll: The direction of the call is compared to the virtual address of kernel32´s text section as well as to its ending address (VA + *SizeOfRawData*).

Now if those conditions are fulfilled, the call can be patched.

Afterwards, the encryption of the virus code takes place and the encrypted bytes are appended to the file.

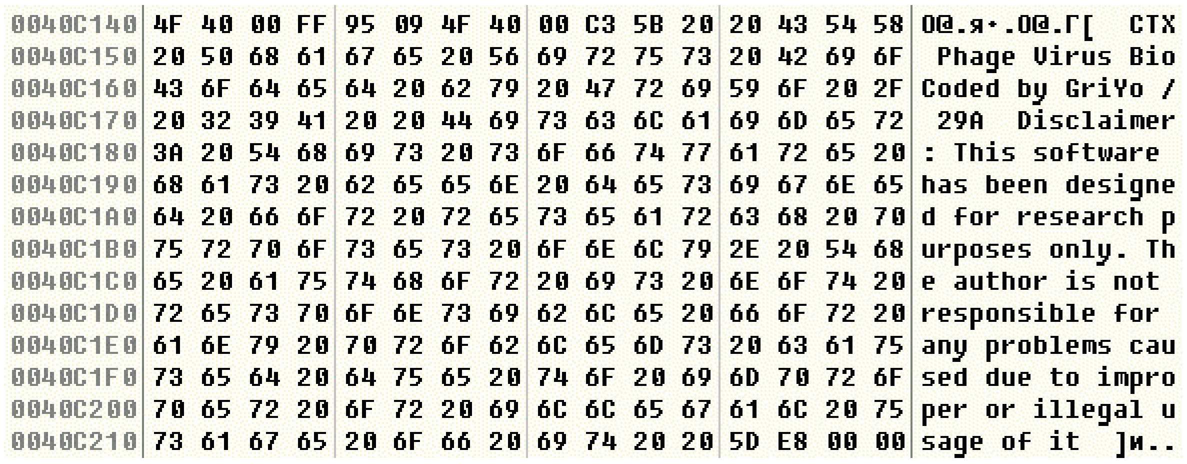
If all the encryption is done, the file is unmapped again; the old filetime (*CreationTime*) is restored from the WIN32_FIND_DATA structure.

If there are no more files to be found or at least 5 files have been infected in every folder, there is a JMP to the restored function call. Now the infected program continues as usual.

7.2.3 Disinfection

The patched call has to be found; starting from the infected program´s entry point, one has to look for calls to the last section, where the virus code is stored.
These calls must start with E8 or FF15, followed by an address within the last section.

If the right call was found, it can be entered and the decryption routine can be run. The virus author´s message within the code, located at the virus code offset + 140h, can help to determine when the decryption is done (see Image 9).



```
0040C140 4F 40 00 FF 95 09 4F 40 00 C3 5B 20 20 43 54 58 O@.я•.O@.Г[  CTX
0040C150 20 50 68 61 67 65 20 56 69 72 75 73 20 42 69 6F  Phage Virus Bio
0040C160 43 6F 64 65 64 20 62 79 20 47 72 69 59 6F 20 2F Coded by GriYo /
0040C170 20 32 39 41 20 20 44 69 73 63 6C 61 69 6D 65 72  29A  Disclaimer
0040C180 3A 20 54 68 69 73 20 73 6F 66 74 77 61 72 65 20 : This software 
0040C190 68 61 73 20 62 65 65 6E 20 64 65 73 69 67 6E 65 has been designe
0040C1A0 64 20 66 6F 72 20 72 65 73 65 61 72 63 68 20 70 d for research p
0040C1B0 75 72 70 6F 73 65 73 20 6F 6E 6C 79 2E 20 54 68 urposes only. Th
0040C1C0 65 20 61 75 74 68 6F 72 20 69 73 20 6E 6F 74 20 e author is not 
0040C1D0 72 65 73 70 6F 6E 73 69 62 6C 65 20 66 6F 72 20 responsible for 
0040C1E0 61 6E 79 20 70 72 6F 62 6C 65 6D 73 20 63 61 75 any problems cau
0040C1F0 73 65 64 20 64 75 65 20 74 6F 20 69 6D 70 72 6F sed due to impro
0040C200 70 65 72 20 6F 72 20 69 6C 6C 65 67 61 6C 20 75 per or illegal u
0040C210 73 61 67 65 20 6F 66 20 69 74 20 20 5D E8 00 00 sage of it  ]и..
```

Image 9: “Disclaimer” within the decrypted code of CTX Phage

The original call which is restored later on is stored at virus code offset + 2c5, so the patched call can now be restored and the virus code can be deleted.
The (possibly) overwritten .reloc section is not necessary for proper execution and thus does not have to be restored.

8. Other file infection techniques: A short overview

There are many more file infection techniques than the ones mentioned on the last few pages; here is an overview over some of them.

8.1 Overwriting viruses

Overwriting is the most simple infection technique. Disinfection is impossible in this case and backups made on a regular basis are the best way to protect oneself from data loss.
There are different ways of inserting code by overwriting:

- Infected files are completely erased and their code is replaced by the virus code. Of course, this kind of infection is very obvious and easy to detect.
- The hostfile is overwritten from the beginning. If the viruscode is not bigger than the hostfile, the filesize will not change. Still, detection is easy.
- Another seldom and rather ineffective technique is random overwriting: The virus code is inserted randomly somewhere within the host and will hardly ever be executed. Finding the inserted code is more difficult and takes longer than in the first two cases, because the whole file must be scanned instead of searching in “known” places, such as the beginning or the end.

8.2 Compressing viruses

This kind of virus uses runtime packers to compress the hostfiles code.
The virus code contains the decompressor and at runtime, the hostfile is decompressed again.

This technique can decrease the possibly grown filesize or prevent the filesize from changing at all, making it harder to detect the infection.

The packers can be custom- made, but often, also common packers like UPX are used.

8.3 Companion viruses

Companion viruses do not need to infect host files; instead, they are executed instead of them.

This can (in a more “traditional” way) happen by creating a COM file (Notepad.com) with the same name as an EXE file (Notepad.exe). If “Notepad” is run via the command line, the COM file will be executed instead of the EXE file.

Nowadays, companions often change the extensions of an executable and create a copy of the virus code under the original filename. After the execution of the viruscode, the original file is renamed to EXE again and the virus code is deleted.

8.4 Dll injection

In this case, the infection happens by a virus creating a dll with the malicious code and inserting it within running processe´s address spaces at runtime. The actual content of a hostfile is not changed.

This can be done by

- Creating Windows hooks. This technique can, for example, be used to monitor keyboard and mouse activities, depending on the HookType used
- Using the *CreateRemoteThread* API and *LoadLibrary* or
- Instead of creating a dll, *WriteProcessMemory* can be used to write the virus code directly into a running processes memory; afterwards, it can be run with *CreateRemoteThread* as well.

Internet connections made by a dll running in the context of some host program can often be successfully hidden from a desktop firewall.

The classification of a malware making use of this technique as a virus, is a matter of argument. Still, the address space of the hostfile gets “infected” by the dll. Disinfection is rather easy: The original executable file that created the dll has to be found and deleted, as well as the dll itself.

9. Résumé

The analysis of viruses, and malware in general, is a fascinating and very complex topic. Sometimes it is comparable to a treasure hunt, whereas one´s way to the “treasure”, respectively to completion, can be very time- consuming and energy- sapping (most of all at the beginning of one´s learning process). On the other hand, getting a satisfying result is a great feeling and motivates one to go on.

Curiosity, a lot of studying, enthusiasm for the topic, endurance and some talent are needed if one wants to become a good malware analyst.

The analyses which were done and described within the scope of this paper were very educational and offered the possibility to become more familiar with some of the common file infection techniques.

They showed that it is important to acquire a steady way of proceeding and structuring that can be applied in slightly modified forms on every new analysis. Without such a (more or less fixed) strategy, there is the danger of losing the thread and getting confused.

Although one encounters some schemes, techniques or usage of library functions again and again, there is always something new and different to discover; this might, of course, decrease after a longer time of working in this field.

Still, the author of this paper will definitely go on studying the captivating field of malware analysis, extend her knowledge and not stop being curious.

10. Bibliography

10.1 Linklist

Computer AntiVirus Researcher's Organization (CARO): http://www.caro.org/

Microsoft Developer Network (MSDN): http://msdn.microsoft.com/en-us/default.aspx

The WildList Organization International: http://www.wildlist.org/

VX heavens: http://vx.netlux.org/

10.2 Literature

Eilam, Eldad: Reversing: Secrets of Reverse Engineering. USA 2005

Marsh, Kyle: Win32 hooks.
Article taken from http://msdn.microsoft.com/en-us/library/ms997537.aspx

Microsoft Portable Executable and Common Object File Format Specification.
Available for download at
http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx

Pietrek, Matt: Peering Inside the PE: A Tour of the Win32 Portable Executable File Format.
Article taken from http://msdn.microsoft.com/en-us/library/ms809762.aspx

Szor, Peter: The art of computer virus research and defense. Amsterdam 2005

Riau, Costin: A Virus by Any Other Name: Virus Naming Practices
http://www.symantec.com/connect/articles/virus-any-other-name-virus-naming-practices

11. Appendix

11.1 Footnotes

[1] The following tables were adopted almost unmodified from the *Microsoft Portable Executable and Common Object File Format Specification* (see bibliography).

[2] Definition by Paul Ducklin adopted from wildlist.org (see bibliography):*"For a virus to be considered In the Wild, it must be spreading as a result of normal day-to-day operations on and between the computers of unsuspecting users."*

[3]The .reloc section holds a table of base relocations, those are needed in case the loader cannot load the file where it was specified by the linker. For more information, see chapter 6.6 of the *Microsoft Portable Executable and Common Object File Format Specification*

[4] read, write and execution permissions

[5] See *http://msdn.microsoft.com/en-us/library/aa365740(VS.85).aspx*

[6] This was a common technique used by the 29A group; after file infection, the file´s size is increased to a value dividable by 65h (101 decimal) to avoid double infections.

11.2 Glossary

Address space: A range of addresses with a common identifier in which every single address can be referred to conclusively (e.g. offsets in relation to a certain imagebase address).
admin$ share: The admin$ share is part of the Administrative share on Windows which is used to share files and folders within a local network. By default, admin$ points to the environment variable %SystemRoot%; on Win XP, it is the \WINDOWS folder. More information can be found here: *http://en.wikipedia.org/wiki/Administrative_share*
File signatures: They are also referred to as "magic numbers" and are used to identify file formats. Examples are the string "MZ" in DOS and PE files or the "PK" in zip files.
Host/hostfile: Both terms are used synonomically in this paper and referred to a file that was (or is going to be) infected by a virus.
Imagebase: The base address at which a loaded file was mapped into memory. For .exe files, it is in most cases 0x00400000 (see also the *ImageBase* field in the PE header).

Import lookup table: A component of the import table (or .idata section) that consist of an array with one element for every function to import. The fields within these elements hold ordinals or *RVA*s to hint/name entries (see *hint/name table*).
Import table: It used to import dlls and consists of the import directory table, the *import lookup table*, the *import address table* and the hint/name table.
Metamorphic virus: A virus that re- writes its own code in a way that its structur (or “grammar”) is changed but the semantical meaning/functionality remains equal
Offset: Used equivalently to the term “*virtual address*” within this paper
Original entry point (OEP): The OEP is the point in a program´s code where it originally starts (before getting compressed/packed or protected by additional software).
Packer: Packers are used to decrease a file´s size, using a respective algorithm.
Payload: Additional malware functionality, besides replication. A typical payload would be overwriting files with garbage code, deleting files, foramtting the hard disk etc.
Polymorphic virus: A polymorphic virus changes the look of its code (without changing its structure; see *metamorphic virus*) to avoid being detected.
Relative virtual address (RVA): An address in memory which is relative to a base address (see also *Virtual address*)
Ring: Defines a processe´s privileges. Ring 0 means the privileged kernel mode, while rings 1-3 refer to the unprivileged user mode with decreasing rights from 1 to 3. à *http://de.wikipedia.org/wiki/Ring_(CPU)*
Struct: A data type in the programming language C which can be used to unite variables of different datatypes within one element (a struct).
System file checker (sfc.exe): A utility that is used to verify the correctness of system files and to restore corruptions if possible. Some files are checked automatically when started; additionally, sfc.exe can be ran via the command line.
Virtual address (VA): A virtual address consists of a base address and an offset (the *relative virtual address*)
Windows hooks: “A *hook is a mechanism by which a function can intercept events (messages, mouse actions, keystrokes) before they reach an application. The function can act on events and, in some cases, modify or discard them.”* (Quote taken from the article “Win32 hooks” by Kyle Marsh; see bibliography)

11.3 API Listing

All API (Application Programming Interface) functions mentioned within this paper are listed below; except from ExtractIcon and SfcIsFileProtected, all of them are part of Kernel32.dll.

An additional “A” or “W” appended to an API name (FindFirstFileA, FindNextFileW) stands for either the ANSI (“A”) or the Unicode version (“W”) in view of the return values.

CopyFile: http://msdn.microsoft.com/en-us/library/aa363851(VS.85).aspx

CreateFile: http://msdn.microsoft.com/en-us/library/aa363858(VS.85).aspx

CreateFileMapping: http://msdn.microsoft.com/en-us/library/aa366537(VS.85).aspx

CreateRemoteThread: http://msdn.microsoft.com/en-us/library/ms682437(VS.85).aspx

CreateThread: http://msdn.microsoft.com/en-us/library/ms682453(VS.85).aspx

ExtractIcon: http://msdn.microsoft.com/en-us/library/ms648068(VS.85).aspx à **shell32.dll**

FindClose: http://msdn.microsoft.com/en-us/library/aa364413(VS.85).aspx

FindCloseChangeNotification: http://msdn.microsoft.com/en-us/library/aa364414(VS.85).aspx

FindFirstFile: http://msdn.microsoft.com/en-us/library/aa364418(VS.85).aspx

FindNextFile: http://msdn.microsoft.com/en-us/library/aa364428(VS.85).aspx

FreeLibrary: http://msdn.microsoft.com/en-us/library/ms683152(VS.85).aspx

GetCurrentDirectory: http://msdn.microsoft.com/en-us/library/aa364934(VS.85).aspx

GetCurrentProcess: http://msdn.microsoft.com/en-us/library/ms683179(VS.85).aspx

GetProcAddress: http://msdn.microsoft.com/en-us/library/ms683212(VS.85).aspx

GetSystemDirectory: http://msdn.microsoft.com/en-us/library/ms724373(VS.85).aspx

GetSystemTime: http://msdn.microsoft.com/en-us/library/ms724390(VS.85).aspx

GetWindowsDirectory: http://msdn.microsoft.com/en-us/library/ms724454(VS.85).aspx

IsDebuggerPresent: http://msdn.microsoft.com/en-us/library/ms680345(VS.85).aspx

LoadLibray: http://msdn.microsoft.com/en-us/library/ms684175(VS.85).aspx

MapViewOfFile: http://msdn.microsoft.com/en-us/library/aa366761(VS.85).aspx

ReadFile: http://msdn.microsoft.com/en-us/library/aa365467(VS.85).aspx

SetFilePointer: http://msdn.microsoft.com/en-us/library/aa365541(VS.85).aspx

SfcIsFileProtected: http://msdn.microsoft.com/en-us/library/aa382536(VS.85).aspx à **SFC.dll**

VirtualAlloc: http://msdn.microsoft.com/en-us/library/aa366887(VS.85).aspx

WriteProcessMemory: http://msdn.microsoft.com/en-us/library/ms681674(VS.85).aspx