# Static detection of files

**AV evasion v1.0**

# Table Of Contents

# Part 1. Structure & data

## 0000. Beginning

It is possible to philosophize long both about possibility of creating undetectable virus, and about detecting absolutely all files, but it is known exactly: there are effective (and not really) detection methods of malware, and no less effective (and even more) techniques of anti-detection. That's what we'll talk about.

I propose the following classification of antivirus detection:

1. static;
2. dynamic;
3. non-contact =)

In this text we'll discuss the first point. By the way, the same detection methods (these and others) can be used both for first and second points, but in other form.

Under **static detection** we'll understand methods applied to detect malware and produced without actually executing malware code (methods used during check of files on disk (local etc), traffic ("on the fly") et al.).

Let's take a closer look at how avers can catch files by **analyzing their structure and data, and possible variants for circumventing these detections**.

The **idea** is that files should be "**correct**" in terms of anti-virus. That is, files created/infected with our code should be similar to usual applications as much as possible by their structure/code/actions (for example, *calc.exe* from *Windows*). The result: knowledge/experience/something else + a beautiful "**clean**" file internally and externally. To achieve this goal it is sufficient to take and examine each piece of at least one of any file - all other are studied/made similarly (PE-format is always at hand).

So, we'll focus on test1.exe, created by me in *Visual C++ 6*. This is a usual Win32-application that contains all the most necessary (without *bound/delay import etc*). And further, we'll consider only those fields/structures of a file, which can cause any difficulties/doubts at their filling. Other fields usually have the same values, as corresponding fields in test1.exe.

All files/sources attached to the text, were tested on *x86*: *Win XP (SP2/SP3), 7, Vista; and on x64: Win 7*.

# 0001. Structure

Let’s start with an overview of the general layout of the PE-file. Here it is:

| |
|---|
| IMAGE_DOS_HEADER (0x5A4D==“MZ”) |
| DOS Stub + Rich Signature |
| Signature (0x4550 == “PE\0\0”) |
| IMAGE_FILE_HEADER |
| IMAGE_OPTIONAL_HEADER (with IMAGE_DATA_DIRECTORY) |
| IMAGE_SECTION_HEADER (.text) |
| IMAGE_SECTION_HEADER (.rdata) |
| IMAGE_SECTION_HEADER (.data) |
| IMAGE_SECTION_HEADER (.rsrc) |
| … |
| Section .text |
| Section .rdata |
| Section .data |
| Section .rsrc |
| … |
| …other info… |
| |

Table 0001.0000 – general layout of the PE-file

Let's go ahead for more details.

# 0010. IMAGE_DOS_HEADER

It is located in the beginning of the file.

## DWORD e_lfanew;

Offset of the structure **IMAGE_NT_HEADERS** in bytes from the beginning of the file.

For this field we use the most frequent values in the range **[0xC0..0xF8]**. Must be multiple of **8**.

# 0011. Rich Signature

Suppose we have a trojan, which is triggered by av (let's call it **fuckingav**). During various experiments we find out that detection is tied to several locations in a file: some bullshit in the code, import and...something is wrong with rich-signature (r-s). And if detection by the last sign is eliminated, the file becomes clean. After successive experiments with r-s we understand that **fuckingav** checks it in some tricky way, and replacing one byte with another (random) won't work. It's not an option to copy r-s every time from other files, too. But, as it is said, on each artful ass there will be a dick with the screw, and therefore we'll generate r-s as it looks in a file compiled with *VC++*. For this purpose, this will help us, and also read on here and here (algorithms from this reference didn't changed and checked some values, so this code has been rewritten and upgraded by me).

*R-s contains information about versions of compiler/linker, and possibly, about some compiling/linking flags/options*. It doesn't store identifiers of the hardware etc, so, if the program will be assembled on different computers with same software (same versions, service packs, dlls etc), r-s of those files will coincide.

Graphically the r-s looks like:

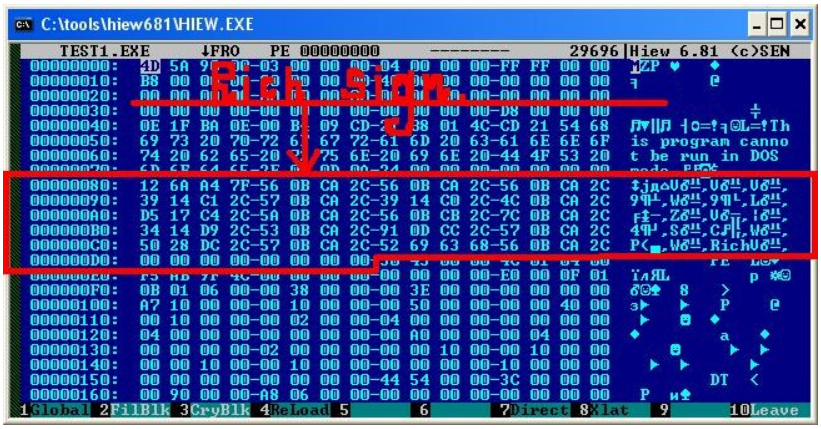


Figure 0011.0000 - studying r-s (test1.exe) with a hex-editor - hiew

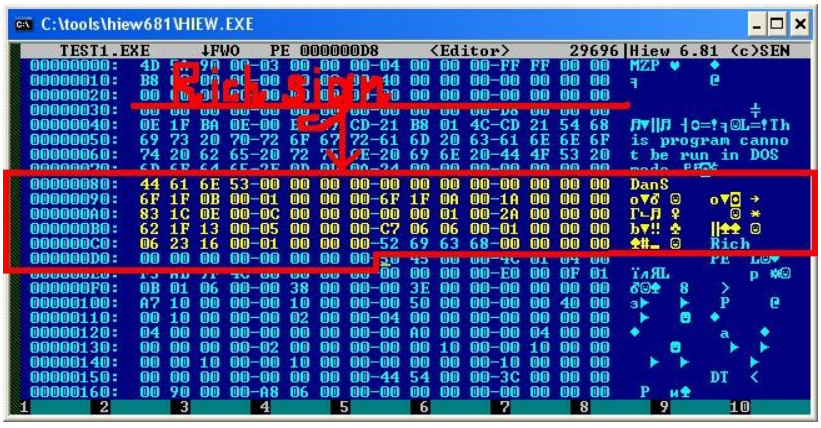


Figure 0011.0011 – decrypted r-s

Initially there are bytes (at offset 0x80) 0x12 0x6A 0xA4 0x7F. This is a string 'DanS' XORed with some number (let’s call it **xmask**).

Further successively follows 3 **xmask**s (0x56 0x0B 0xCA 0x2C). The mask is a 32-bit number calculated by some algorithm.

Then there are data (let’s call it **xdata**) XORed with **xmask** (in a offsets range [0x90..0xC7]).

For each r-s there can be its own set of data and quantity.

After the data there is string "Rich", **xmask** and zeros (in this example it’s 8 zeros, but it can be both more and less. The number of zeros is multiple of 8).

In some cases, absence of r-s can improve the situation in relation to detections... perhaps until the next antivirus database update.

But why do we need this, if there are working algorithms for generating "right" r-s =). Btw, ms always creates it.

Algorithms to check r-s integrity, substitutions of its data, and also its generation from scratch are here: frs_asm.rar & frs_C.rar.

# 0100. IMAGE_FILE_HEADER

Contains the most general information about file and located immediately after the **signature 0x4550** (“PE\0\0”).

## WORD NumberOfSections;

Number of sections in a file.

If you create an **exe**, it is better to create **4** sections (.text, .rdata, .data, .rsrc). Others are added only when necessary (eg, .tls, .reloc). For **dll** it is important to include relocs (.reloc).

If we infect a file, then do it carefully. For example, the victim (a usual file, without the overlay etc) has 3 sections (.text, .rdata, .data). You can create **+1** resource section, put the icon and create another rsrc that will keep the body of our beast (**xcode -** our cheerful code =)). Thus, we have an infected file, and created a completely normal section.

By the way, there was a case when a clean file was detected by some antivirus, and after infection of this file, "signature" disappeared. VX will save the world ).

## DWORD TimeDateStamp;

Time when **PE-file** has been built. This field stores the number of seconds since January 1, 1970 until the time of file creation.

You can use function **gmtime** from standard **C** library (**time.h**), changing time from seconds into a readable form:

```
DWORD TimeDateStamp = 0x4C9FABF5;
struct tm *xtm = gmtime((const long*)&TimeDateStamp);
printf("day = %d\nmonth = %d\nyear = %d\n", xtm->tm_mday,
xtm->tm_mon, (xtm->tm_year + 1900));
```

Basically, one can generate any number, for example, in a range **[0x40000000..0x4C000000]** (2004-2010).

## WORD Characteristics;

File attributes.

Most characteristics are built of these flags (bitwise "OR"):

```
//Relocation info stripped from file.
#define IMAGE_FILE_RELOCS_STRIPPED                  0x0001

//File is executable(i.e. no unresolved external references).
#define IMAGE_FILE_EXECUTABLE_IMAGE                 0x0002

//Line numbers stripped from file.
#define IMAGE_FILE_LINE_NUMS_STRIPPED               0x0004

//Local symbols stripped from file.
#define IMAGE_FILE_LOCAL_SYMS_STRIPPED              0x0008

//32 bit word machine.
#define IMAGE_FILE_32BIT_MACHINE                    0x0100

//File is a DLL.
#define IMAGE_FILE_DLL                              0x2000
```

For **exe** you can boldly use the value **0x10F** (for **dll** – **0x210E**).

# 0101. IMAGE_OPTIONAL_HEADER

This structure is obligatory and contains important additions to general information in IMAGE_FILE_HEADER.

## BYTE MajorLinkerVersion/MinorLinkerVersion;

Contain a version of the linker that created the file. Numbers must be in decimal form, eg, 2.56, 6.0 etc.

Value of this field can be anything, but, not to provoke **fuckingav**, I suggest using a value of **6.0**.

## DWORD SizeOfCode/SizeOfInitializedData/SizeOfUninitializedData;

Total size of all sections of a code/with initialized data/with uninitialized data.

Values of these fields are calculated as follows:

```
//align formula (see below)
#define ALIGN_UP ((x + (y - 1)) & (~(y - 1)))

//(code_sec.Characteristics & IMAGE_SCN_CNT_CODE) == 1
SizeOfCode += ALIGN_UP(code_sec.Misc.VirtualSize, FileAlignment)

//(init_sec.Characteristics & IMAGE_SCN_CNT_INITIALIZED_DATA) == 1
SizeOfInitializedData += ALIGN_UP(init_sec.Misc.VirtualSize, FileAlignment)

//(uninit_sec.Characteristics & // IMAGE_SCN_CNT_UNINITIALIZED_ DATA) == 1
SizeOfUninitializedData += ALIGN_UP(uninit_sec.Misc.VirtualSize, FileAlignment)
```

Whether we create or infect file, values of these fields (and other too) always should be correctly counted up. Otherwise, once the **fuckingav** will inform: “Fucking shit, here the virus is dancing!!!” - and all aesthetics will equal shit.

## DWORD AddressOfEntryPoint;

Entry Point **RVA**, counted from the beginning of **ImageBase**.

In order to avoid rage of **fuckingav**, it is best to have an entry point (EP) in the range:

```
//EP != 0
(EP >= code_sec.VirtualAddress &&
 EP < (code_sec.VirtualAddress + code_sec.Misc.VirtualSize))
```

If code_sec.`Misc.VirtualSize` == 0, then use code_sec.`SizeOfRawData`.

## DWORD BaseOfCode/BaseOfData;

**RVA** of the code/data section.

```
BaseOfCode = code_sec.VirtualAddress; //0x1000; //SectionAlignment; //!!!!!

// = RVA first data_sec
BaseOfData = data_sec.VirtualAddress;
```

## DWORD ImageBase;

Base address of the file in memory.

The linker sets the default **ImageBase** to **0x00400000.** We will use this value to generate exe-files (for DLL - **0x10000000**). If there will be something else than this value, there is a big chance to get detection during next **fuckingav** update.

## DWORD SectionAlignment/FileAlignment;

Alignment multiplicity of sections in memory/on disk.

The alignment formula:

```
//for addresses of sections (physical and virtual)
#define ALIGN_DOWN(x, y)     (x & (~(y - 1)))

//for the sizes of sections
#define ALIGN_UP(x, y)       ((x + (y - 1)) & (~(y - 1)))
```

, where **x** – aligned value, **y** – align factor.

Usually SectionAlignment = **0x1000**, and FileAlignment = **0x200** or **0x1000**.

## DWORD SizeOfImage;

Overall size of the loaded application in memory, starting from **ImageBase** till the end of the last section, and aligned on SectionAlignment value. It’s calculated as follows:

```
SizeOfImage = last_sec.VirtualAddress + ALIGN_UP(last_sec.Misc.VirtualSize,
IMAGE_OPTIONAL_HEADER.SectionAlignment);
```

### DWORD SizeOfHeaders;

Total size of all headers, aligned in a file on FileAlignment (in memory on SectionAlignment). Often equal to **0x400**.

### DWORD CheckSum;

The checksum of the image file.

For usual executable files the checksum isn't verified, i.e. can be any. If it’s equal to zero, then it can be any too. For all system dlls – it must be correct.

To get the checksum of the executable file, you must call **CheckSumMappedFile** with appropriate parameters. This function is available from **Imagehlp.dll** library.

If we create or infect a file, in which **CheckSum** = 0, then we leave value equal to zero in this field.

### IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];

An array of structures of type **IMAGE_DATA_DIRECTORY**, the number is stored in field **NumberOfRvaAndSizes** (usually = **0x10**).

By default, *VC++ (6)* creates in "normal" application 3 directories: *import*, *resource*, **IAT** (at least for *Win32 Application*). Therefore, these 3 directories also should be in our to be created file (+tls, relocs if necessary).

## 0110. IMAGE_SECTION_HEADER

An array of structures of type IMAGE_SECTION_HEADER, the number is stored in NumberOfSection field.

### BYTE Name[8];

8-byte ASCII-name of section. Better to use standard names: .text, .rdata, .data, .rsrc, .tls, .reloc.

Use this information when you create your own files and when infect files (if you chose a method of adding a new section, choose section name wisely).

## DWORD VirtualSize;

Virtual size of section, when file is being loaded into memory, virtual size is aligned up on SectionAlignment. And in a file VirtualSize < SizeOfRawData for all sections. Exception is .data section (then vice-versa).

## DWORD VirtualAddress/ PointerToRawData;

Contain RVA address of the section start in memory and section offset relative to beginning of a file. Usually first_sec.VirtualAddress = **0x1000**, first_sec.PointerToRawData = **0x400** or **0x1000**. Values of addresses of sections are aligned down on SectionAlignment/FileAlignment.

## DWORD SizeOfRawData;

The physical size of section aligned up on FileAlignment.

## DWORD Characteristics;

Section attributes.

Often, attributes are built of the following flags (bitwise "OR"):

```
// Section contains code.
#define IMAGE_SCN_CNT_CODE                              0x00000020

// Section contains initialized data.
#define IMAGE_SCN_CNT_INITIALIZED_DATA                  0x00000040

// Section contains uninitialized data.
#define IMAGE_SCN_CNT_UNINITIALIZED_DATA                0x00000080

// Section can be discarded.
#define IMAGE_SCN_MEM_DISCARDABLE                       0x02000000

// Section is executable.
#define IMAGE_SCN_MEM_EXECUTE                           0x20000000

// Section is readable.
#define IMAGE_SCN_MEM_READ                              0x40000000

// Section is writeable.
#define IMAGE_SCN_MEM_WRITE                             0x80000000
```

For "usual" files:

```
.text.Characteristics   =     0x60000020;
.rdata.Characteristics  =     0x40000040;
.data.Characteristics   =     0xC0000040;
.rsrc.Characteristics   =     0x40000040;
```

As an example, I decided to check test1.exe at *virustotal.com*, before and after changing the attributes of code section, and here is what I got:

```
.text.Characteristics = 0x60000020;
.text.Characteristics = 0xE0000020;
```

So, there will be always predators ready to grab an "**anomaly**" =)

## 0111. Import

**Import** - a mechanism that allows to use function and/or variables from modules which are distinct from the given module.

For file to work correctly in many versions of Windows and not arouse suspicion of various AV, presence of the import in a file is a must.

Import Table begins with an array of structures of type **IMAGE_IMPORT_DESCRIPTOR**.

For "usual" files by default some fields of import are filled with such values, that the system loader during file start would use the standard import mechanism. Therefore, I suggest to use below mentioned values.

### DWORD OriginalFirstThunk;

RVA of array of double words. Each element of this array is the union **IMAGE_THUNK_DATA32** and corresponds to one function that is imported by PE-file.

### DWORD TimeDateStamp;

Timestamp when the file was created. Since we are using standard import mechanism, we write here **zero (0).**

### DWORD ForwarderChain;

This field is related with functions forwarding. We write here **0**.

## DWORD Name;

ASCII-name of imported dll.

## DWORD FirstThunk;

RVA of array of double words IMAGE_THUNK_DATA32.

FirstThunk is **Import Address Table** (**IAT)**.

## IMAGE_THUNK_DATA32

Structure is a **dword**, which corresponds to a single imported function.

## DWORD AddressOfData;

If function is imported by name, then this field contains the RVA on **IMAGE_IMPORT_BY_NAME** structure, in which there is a name of the necessary function. If import occurs under number (ordinal), the highest bit of double word is set to 1.

## IMAGE_IMPORT_BY_NAME

Contains information about import functions.

## WORD Hint;

This **word** was created for use by the system loader, that loader could find quickly a function in the export table. And contains the index of function in array of exported functions, index is equal to ordinal. Naturally, the same function have different ordinals in different versions of windows, so there is no universal hint for all systems.

I suggest to calculate hint as follows:

1. get a random number and save it in a variable cor_hint;
2. find the desired function in the export table of the library (dll) and save it's ordinal in a variable hint;
3. add to hint value cor_hint;

```
// this value will be calculated once and is added to every new hint;
WORD cor_hint = rand()%0x100;

// for each hint we find the appropriate ordinal and add value cor_hint;
WORD hint = *AddrOfNameOrd + cor_hint;
```

Thus, hint will be random, but calculated on the basis of a corresponding ordinal.

**BYTE Name[?];**

ASCII-name of imported function.

Now let's talk about an order of creation of import and other details.

Having carefully considered our test1.exe (and similar files), we see the following:

1. all import is in .rdata section (import is at address, multiple of 4);
2. at the beginning of the .rdata section there is IAT (after which may lie a variety of data ... and can not lie);
3. followed by the import table;
4. followed by arrays of dwords IMAGE_THUNK_DATA32. RVA of these arrays are saved in fields OriginalFirstThunk (lookup-table);
5. after, follows arrays of structures IMAGE_IMPORT_BY_NAME and names of dlls, in such an order: first comes an array of structures IMAGE_IMPORT_BY_NAME, and after – dll name – it imports the functions whose names are in this array. Then again goes an array of structures and dll name etc. And the number of such pairs, of course, equal to amount of imported libraries in this file.

Libraries are added in import in the manner of their connection to the project. Therefore (yet) there are no rules: first can go KERNEL32.dll, and completely other dll.

WinApis can be placed in any order too (sorting is needed only for export).

As for the paired APIs: of all the tests carried out by me, did not reveal any case of detection on given sign (at least, it wasn’t key). Though, perhaps, for some functions, and under certain circumstances AVs can catch. For example, in test1.exe there is LoadLibraryA, but there isn't FreeLibrary; WriteFile is present, but CreateFile is absent (though this winapi is not necessary, as with WriteFile, it is possible to write not only to a file, but also in console =)).

Now about strings.

Names of libraries are often such: KERNEL32.dll, USER32.dll, GDI32.dll etc. (name is uppercase, and “dll” is lowercase).

Also I will notice, that names of DLLs and functions (and possibly other strings in the .rdata), most likely aligned on an even number of bytes:

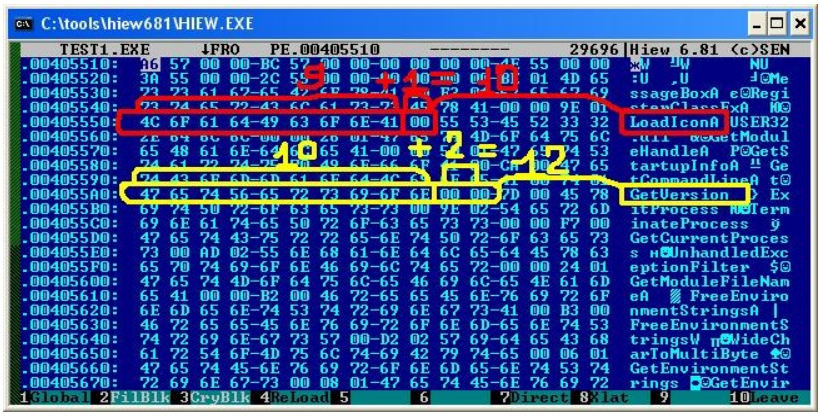


Figure 0111.0000 – alignment of dll names and functions on an even number of bytes

The figure shows examples of two strings - names of functions from various libraries, which are aligned by zeros. When you create fake import, use only frequently used DLLs and functions. Here is a big plus - absence of "anomalous" strings and even if file is detected by import, then most likely, it is not a main cause, so import can stay “the same” for long time (if necessary, filter it). Sure, there is another way - to use “rare” dll / functions. Minus – detection exactly by them. But this is also advantage: the possibility of easy cleaning (it is enough to replace detected function by other “clean” function).

Aver’s have a "black list" of api/dll, which in their view, are mainly used in malware. And also, perhaps, on the approach is the "white list", more severe =). What is not in it - is evil. Such unhealthy situation must be considered.

To gather statistics of winapis/libraries I wrote an utility stait_C.rar, which can help us in this amusing affair.

## 1000. Resources

Practice shows that resources affect **fuckingav** well. Therefore, we’ll look at what it is, and what is better to add to our file.

Resources Table corresponds to resource section (**.rsrc**). It is important to know that resources - is a binary sorted tree, and in windows used only the first 3 levels (Type, Name, Language). Remember that! Since this info will facilitate coding with resources.

Usually RT_ICON, RT_GROUP_ICON and some other crap is enough (strings, dialogs, bitmap etc). And in such crap you can store important encrypted data (code). RT_RCDATA – be careful with this one, store only garbage in it, nothing more (**fuckingav** makes special checks of the contents of the given resource).

To work with the resources we need: BeginUpdateResource, UpdateResource, EndUpdateResource, EnumResourceNames, EnumResourceTypes etc. (by the way, it is possible to infect files with these functions, instead of using CreateFile/CreateFileMapping/MapViewOfFile/UnmapViewOfFile).

## 1001. Sections

Section - the continuous area of memory with the same attributes. This section briefly discussed only the most important of them (tls & relocs maybe next time).

### .text

In this section is placed the program code (we will talk separately about it).

### .rdata

Here we store import, and after it, we fill section with zeros for alignment (ALIGN_UP(SizeOfRawData, FileAlignment)).

### .data

In this section we store various data: random strings, numbers etc.

### .rsrc

Resources of the program (see "**Resources**" section).

Sections in a file and in a memory should go in the order in which they are listed here. Otherwise the "anomaly". Also, it is important to watch the sizes of the sections and the ratio of sizes.

Although the collected statistics on files showed that there are no any rules (there is files with absolutely opposite proportions of sections), you can do so:

1. if new file is created, we take as the standard - sizes of sections and their percentage from other "clean" files of required sizes (for example, if we are going to generate file with the size of = 80kb, then we first find any file whose size is also equal to 80kb, and built with VC + + (6). And then, we use the sizes of its sections and their proportions in creating our sections);
2. but if we infect a file, then we try to stick to the proportions of its sections. That is, for example, if the file had 15%/12%/70% of code/data/resources before infection, after infection

it is desirable to maintain such a parity (but not necessarily, if you remove other detection flags);

3. as for the storage of encrypted code for the 1-st and 2-nd points, then we put it into one of the sections (or overlay) or, alternatively, break code into pieces and throwing them into sections with preservation of proportions. Entropy of the resulting file (it's sections) must be normal (see below);
4. test everything carefully;
5. often there are files, which (.text_sec.SizeOfRawData > .rdata_sec.SizeOfRawData) && (.text_sec.SizeOfRawData > .data_sec.SizeOfRawData).

## 1010. Entropy

In some situations it is useful to programmatically find out the file is packed or not. And on the basis of the result continue any calculations. It has a direct relationship on antiviruses.

Suppose a file is checked for malware. The antivirus heuristics will determine, whether the file is packed/encrypted. If the resulting percentage/ratio is less than the specified limit, then the analysis is completed. Otherwise, it determines with what the file is compressed/encrypted. If it is known with what – it shows the name of the packer/cryptor. If not, it'll write that it's some XCryptVirus. That's heuristics on simple viruses. And such checks can be for anything…before we rejoice that we just created a virus, but it's already fucked up. To prevent this shit, we in an any way should be at least on a step ahead.

There are several options to detect a packed file programmatically. Here are some of them:

1. the first instruction at the entry point is pushad;
2. non-standard section names;
3. attributes of section of a code are exposed on write (perhaps any other section, not originally intended for writing);
4. the presence of the overlay;
5. high entropy (file/parts of it);
6. arithmetical mean;
7. signatures;
8. statistical analysis of bytes/instructions;
9. other;

Naturally, each of these points will not **100%** tell whether a file is packed or not. And only their literate combination for certain types of files can give the most accurate result.

With the correct file structure, most of points will be eliminated at once.

Here let’s dwell on entropy.

So, briefly, information entropy - a measure of randomness of the information. For the first time concept of "**entropy**" and "**information**" linked *C. Shannon* in 1948. Shannon used a bit as the unit of information. Measure of the amount of information Shannon suggested counting function, which he called entropy:

**H(i) = -P(i) * log2(P(i));**

**H = H(1) + H(2) + H(i)…**

H - Shannon's entropy, P(i) - probability of occurrence of the i-th symbol.

More information about entropy here and here.

Unit of measurement of information and entropy depend on the base of the logarithm. We will measure in bits on 1 byte to within the tenth.

Empirically established that "normal" value of the entropy for the PE-files (exe) lies in the range [5.5;6.8]. If it turns out more, then most likely the file is packed.

It is possible to notice also that in the different programs having function of calculation of entropy, value of the entropy for the same file can be different. Why?

In fact, the program uses the same formula of the Shannon, but, as has already been written, can be used by different units of measurement, the result can be output as a percentage or ratio. At calculation it is possible to throw back aligning zeros, the most frequent and most rare bytes (for average values), bind to the file size, calculate the entropy for whole file and/or its separate parts (headers, sections etc). If you calculate the entropy of the sections, you can take a minimum size of the section (from the physical and virtual sizes) etc.

In order to avoid being caught by the entropy, it is obvious we need to improve it. Here, too, I suggest a few ways to do this:

1. Permutation Ciphers. However, if the data is already been compressed, this cipher loses its meaning.
2. Huffman codes.
3. Code dilution by "homogeneous garbage". For example, by some algorithm is inserted in different places of a code of a sequence 0x00, 0xFF or other bytes. Up until the entropy is not “normalized". The same algorithm (or another) should be able to clear our code of garbage.
4. Base64. In short, this coding scheme of an arbitrary set of bytes into a sequence of printable ASCII-characters. In general, the length of the resultant code is increased by about 30%. The basic coding rule is simple - we are deliberately reduce the range of characters (bytes), which we have at the exit. And then the smaller the characters we use, the less the uncertainty of their occurrence, and hence entropy is less. For encoding is used 64 characters (2 in 6th degree). In general, read here. Thus it is possible to encrypt some code and cram it into a resource in

"correct" place =) (RT_STRING). Base64 - a kind of substitution cipher. Nothing prevents us to use the cipher (where the symbol size = 6 bits, then the entropy does not exceed 6.0 by definition) and with its own alphabet. By the way, as a variant: take 3 bytes, each of them to take away 2 bits. And write after these 3 bytes new byte consisting of bits of the previous cut, etc. It turns out that each byte will have only 6 bits “working”. The number of different bytes will be reduced from 256 to 64 variants. And entropy, respectively, reduced (<= 6.0) (+ add several bytes - watch the statistics of bytes).

5. Dilute the code so that as a result to receive "a correct" code with low entropy and «correct frequencies». Here and here.

As an example, I’ve created a program xentr_C.rar, for calculating the entropy of files (+ a couple of features).

## 1011. File reputation

Recently, many av had started to promote the so-called cloud technology. And yet, in principle, well, just because of these miraculous technologies someone was hit hard paranoid, someone has lost a heap of money, to somebody came *suspicious.insight*. But there is a big pluses...avers knows =).

Next, we consider the specific av - symantec and his brother.

As symantec writes, they had established a "revolutionary" technology (codenamed *Quorum*), which provides data protection based on reputation estimation. A bit more about it here. Having rummaged in various info and having experimented with NIS 2011, I will add the description of the given technology:

1. norton insight – the technology providing protection without sacrificing performance at the expense of an exclusion of check of trusted files. It allows to reduce scanning time.
2. After the launch of this technology it is initially connects to the server Shasta-rrs.symantec.com. This is so-called, their cloud platform (here it is).
3. Via a secure connection (TLS protocol), transmitted/received/checked different data. By data: sending/receiving ~300 bytes. And if to trust the information from a site of the developer - no personal information, note, and also a checked file is not sent to their servers (the file is not scanned). Transmitted only statistical information about the file: **sha256 hash** of the file (possibly) it’s name, size, version (resources), a digital signature (it’s certain data).
4. The resulting hash with special algorithms is compared with a database (stored on Symantec servers), containing information about known trusted files. The result (reputation/trust degree) + file hash is probably stored in an encrypted form in a database on a user’s computer.

5. If the file has been modified, the trust received earlier will be not valid anymore, and the file will be checked again.
6. a) it should be noted that, perhaps in symantec's database is stored and information about running processes/modules/drivers, and various traces of files, data about the OS and the file system; б) presence of a digital signature in a file do not mean that the file can be trusted. In the possible exception to ms =).

Btw, was found a fairly simple way to circumvent *suspicious.insight* (about it) (however, it will not help against Norton - look for something else):

1. take any file with a digital signature, for example, from ms;
2. rip out from file the digital signature (all data from the IMAGE_DIRECTORY_ENTRY_SECURITY directory) and save it to disk;
3. to other file (which we want to protect from suspicious.insight), we cling the stored digital signature a) save the signature as an overlay in the end of file; b) write down the address and the size of the signature in fields VirtualAddress & Size in directories IMAGE_DIRECTORY_ENTRY_SECURITY; c) correct a field IMAGE_OPTIONAL_HEADER.CheckSum).

The resulting corrected file, of course, won't pass check of the digital signature, but for our problem completely approaches:

```
//first recheck test1.exe - it has no fake signature
//after rechecking it is clear that after a while the file is already detected
//=)
//but now something else: push the button «Show all» and see Suspicious.Insight
before_ds   //test1.exe

//then to test1.exe has been added the signature of excel.exe (XP SP3)
// and check the received file
//as can be seen, even other detections have disappeared
after_ds    //si_sux.exe
```

Src attached: fakeds_C.rar.

## 1100. Good triumphs over evil

It should be remembered: that has helped with cleaning of one file, can be unsuitable in cleaning of another, since for different files may be different detection. Therefore, as the saying goes, normally do - normally will be =). And know, that the strength is in simplicity. That's all for now.

### References


1. Antivirus Engines
2. PE format
3. Microsoft's Rich Signature (undocumented)
4. About Rich Signature
5. Fake Rich Signature
6. Information entropy (eng)
7. Base64 (eng)
8. Calculation entropy (+ comments)
9. About entropy
10. Crypting simple instructions
11. Symantec about technology Quorum
12. Symantec Cloud Platform Shasta
13. Symantec Suspicious.Insight


### Sources

1. test1.exe & si_sux.exe
2. frs_asm.rar & frs_C.rar
3. stait_C.rar
4. xentr_C.rar
5. fakeds_C.rar

### Thanks


**izee**, who assisted in the preparation of this text and other subtle moments. And also *greetings to* **izee**, **Dark Prophet**, **fAMINE** and all others **virmakers**.


09.02.2011.

pr0mix/EOF

Virmaking for yourself...art is eternal