FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Unpacking Framework for Packed Malicious Executables

Gaspar Furtado

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: José Manuel De Magalhães Cruz

Second Supervisor: Jürgen Eckel

June 19, 2013

Unpacking Framework for Packed Malicious Executables

Gaspar Furtado

Mestrado Integrado em Engenharia Informática e Computação

June 19, 2013

Abstract

Malware is a growing concern in the modern connected and machine-dependent world. A common approach to fighting malware is early detection. This is the approach used by most antivirus products. On the other side, malware authors try to keep their software undetected as long as possible in order to achieve their goals. One technique used for this is the use of packers. The ease of use and the protections against detection and analysis that packers provide have made packing malware very popular. An unavoidable fact is that the large majority of malware is packed. The varying complexity of packers from simple compressors to extremely advanced virtual machines have forced the IT security industry to address the problem seriously. The reduced effectiveness of detection on packed binaries is a known problem that the industry tries to solve using different techniques. Static unpacking provides an extremely efficient way of addressing the problem of packed executables. This approach relies on reversing the changes done by the packer to the binary, without executing it. The goal of this project was to implement a static unpacking framework that would allow the unpacking of packed executables. The occurrence of a multitude of different packer families and versions meant that such a tool should allow the incremental addition of support for different packers. Due to the lack of information on the different packers, implementing an unpacker usually involves reverse engineering of its functionality. This was the main challenge of this project.

A static unpacking framework was implemented. Support for four different packers was added the framework: ACProtect 1.41, ASPack 2.12, FSG 2.0, and UPX.

The ultimate purpose of the developed framework was to be integrated into a malware scanning engine, so that unpacking of packed executables could be performed before scanning. This integration was performed and the implemented framework was thoroughly tested with malware samples identified as packed by PEiD signatures. From this testing, the success of the implemented unpackers varied from 86% to 0,015%. These success rates are tightly linked with the quality and precision of the PEiD signatures used to select the testing sample sets and therefore have a relative significance. In total, over 4,5 million samples were tested. There was a significant performance improvement provided by the framework when compared to the dynamic unpacking solution implemented in the company's malware scanner. On average, time was almost halved and memory usage was reduced in most cases.

The implemented solution confirmed the advantages and pitfalls of static unpacking. The performance improvement provided translates into increased user satisfaction, given the real-time, continuous operation of an antivirus product.

Given the amount of different packer versions and families, it is clear that developing a true generic static unpacking framework with support for all packers is a mammoth task. The modular architecture of the developed framework allows for incremental build-up of support for all the different packers, making it a versatile tool and, perhaps, "generic".

Resumo

Na sociedade moderna, interligada e dependente da informática, as aplicações maliciosas (*malware*) são uma preocupação crescente. Uma abordagem comum ao combate do *malware* é a sua deteção precoce. Esta é a abordagem empregue em produtos de antivirus. Por outro lado, os autores de aplicações maliciosas tentam que o seu software se mantenha oculto tanto tempo quanto possível para atingir os seus objetivos. Uma técnica usada para evitar a deteção de malware por produtos de antivirus é o uso de *packers*. A facilidade de utilização e as proteções contra deteção e análise que os *packers* oferecem fizeram do empacotamento (*packing*) de malware uma abordagem muito popular. É incontornável o facto de que a grande maioria do malware é protegido por *packers*. A popularidade e complexidade diversa dos *packers*, desde a simples compressão até máquinas virtuais extremamente avançadas forçou a indústria de segurança infromática a encarar seriamente o problema. A reduzida eficácia da deteção de executáveis protegidos por *packers* é um problema conhecido que a indústria tenta resolver aplicando diferentes abordagens. O desempacotamento estático é uma abordagem extremamente eficiente ao problema de executávies empacotados. Esta abordagem baseia-se em reverter as alterações feitas ao executável, sem a execução do mesmo: desempacotamento estático. O objetivo deste projeto foi o desenvolvimento de uma *framework* para o desempacotamento estático que permitisse o *unpacking* de executáveis empacotados. Devido à existência de uma multitude de diferentes famílias e versões de *packers*, a ferramenta a desenvolver deveria permitir a inclusão incremental de suporte para diferentes *packers*. A falta de informação sobre os diferentes *packers* obriga a que o desenvolvimento de um *unpacker* (desempacotador) para um determinado *packer* envolva a desconstrução da sua funcionalidade. Esta desconstrução foi o principal desafio deste projeto.

No decurso do projeto, foi incorporado suporte para quatro packers diferentes na ferramenta desenvolvida: ACProtect 1.41, ASPack 2.12, FSG 2.0, e UPX.

O objetivo final da ferramenta desenvolvida é a sua integração num *scanner* de *malware* para que os executáveis *packed* possam ser *unpacked* antes de serem analisados. Esta intergração foi feita e a solução implementada foi testada exaustivamente com amostras de *malware* identificadas como empacotadas por assinaturas do programa PEiD. Os testes realizados revelaram que o sucesso do desempacotamento dos diversos packers variou de 0,015% a 86%. Estas taxas de sucesso estão intimamente ligadas à precisão das identificações feitas pelas assinaturas do programa PEiD e têm portanto um significado relativo. No total, mais 4,5 milhões de amostras foram testadas. Houve um aumento de desempenho no uso da ferramenta desenvolvida, quando comparado com a solução de desempacotamento dinâmico previamente implementada no *scanner* de *malware*.

A solução implementada confirmou as vantagens e desvantages da abordágem estática ao desempacotamento de executávies. Com a quantidade de differentes *packers* existentes, é nítido que o desenvolvimento de uma ferramenta estática de desempacotamento de executáveuis verdadeiramente genérica é uma tarefa herculeana. O desenho modular da solução implementada permite uma acumulação incremental de suporte para todos os diferentes *packers*, tornando a ferramenta versátil e, possívelmente, "genérica".

Acknowledgements

I would like to express my gratitude to my supervisor at IKARUS, Jürgen Eckel, for giving me the unique opportunity to work on this project. His contributions and guidance allowed for the success of the project. A special appreciation goes to my supervisor at FEUP, José Manuel De Magalhães Cruz, whose suggestions and reviews greatly improved for the overall quality of this document.
Furthermore, I would like to thank the staff of IKARUS that welcomed me into the company and offered invaluable help in the duration of the project.
Lastly, I would like to express my appreciation to all my family, colleges and friends who reviewed this report.

Gaspar Furtado

O Quinto Império

Triste de quem vive em casa,
Contente com o seu lar,
Sem que um sonho, no erguer de asa,
Faça até mais rubra a brasa
Da lareira a abandonar!

Triste de quem é feliz!
Vive porque a vida dura.
Nada na alma lhe diz
Mais que a lição da raiz-
Ter por vida a sepultura.

Eras sobre eras se somem
No tempo que em eras vem.
Ser descontente é ser homem.
Que as forças cegas se domem
Pela visão que a alma tem!

E assim, passados os quatro
Tempos do ser que sonhou,
A terra será teatro
Do dia claro, que no atro
Da erma noite começou.

Grécia, Roma, Cristandade,
Europa - os quatro se vão
Para onde vai toda idade.
Quem vem viver a verdade
Que morreu D. Sebastião?

Fernando Pessoa, in Mensagem

Contents

List of Figures

List of Tables

Abbreviations

API	Application Programming Interface
ASLR	Address Space Layout Randomization
BYTE	A 8-bit field of data
COFF	Common Object File Format
CPU	Central Processing Unit
DLL	Dynamic Link Library
DWORD	A 32-bit field of data
ELF	Executable and Linkable Format
FSG	Fast, Small, Good
IA	Intel Architecture
IAT	Import Address Table
INT	Import Name Table
IT	Information Technology
OEP	Original Entry Point
OS	Operating System
PE	Portable Executable
RVA	Relative Virtual Address
UPX	Ultimate Packer for eXecutables
VM	Virtual Machine
WORD	A 16-bit field of data

Chapter 1

Introduction

This chapter introduces the project context and motivation. The project's goals and challenges are briefly described. Finally, a summary of each of the following chapters is done.

1.1 Context

The project described in this document was done at IKARUS Security Software, GmbH, to be integrated into the company's scanning engine. IKARUS is an Austrian antivirus company with products ranging from endpoint protection to cloud security. It has over 25 years of experience in the IT security industry and is connected to several other IT security institutions throughout the world sharing malware information. As an antivirus company, the analysis of malware is a key activity of IKARUS, both for the unique identification of the malware family and the investigation of its vulnerability exploits and effects on an attacked system. The company's approach to malware analysis has been mostly a dynamic approach, meaning the monitoring of the execution of the program and the analysis of its behaviour. This approach has many advantages but also some disadvantages, in particular, the fact that some packers make the analysis impossible with a purely dynamic approach. So, a decision was made to try to complement the dynamic analysis with a static one, by which the malicious program is not executed and its (inactive) code is searched for identification (signature matching), included libraries and exported APIs, and deduction of general expected behaviour of the program when it runs.

The malware problem is a growing concern in the modern, computer-dependent world. The battle against malware involves, naturally, being able to detect it. As with any battle, in this case between malware authors and the IT security industry, the strategies used by each side adapt and evolve together. Because no one wants the other side to know their strategies and tools, a central activity performed by both sides is reverse engineering. In the case of malware, the strategies used to detect malware forced the malware authors to invent ways of avoiding detection. From this constant battle, packers emerge as a solution to evade detection. Naturally, the security industry fights back with an array of techniques. One of these techniques is unpacking.

1.2 Project

The unpacking of executables is a technique used by the IT security industry to counter the packing done by malware authors to evade detection. Unpacking is an important task when analysing malware and can be achieved by different means: static and dynamic (see chapter 3). This project aims to create a framework for static unpacking of executables so that they can be analysed and detected properly. The term "unpacking" might be misleading as in many cases compression is just a small part of the result of the operations of the so called runtime packers: deobfuscation, virtual machine code translation and decryption are quite common operations that have to be addressed too (see chapter 3 and 4).

1.3 Motivation and Objectives

The fact that packers commonly employ techniques specifically aimed at avoiding detection by antivirus products makes them an important problem to address for companies in the business. Unpacking enables the detection to work by reversing the changes done by packers. The fact that the company has traditionally tackled the problem mostly with a dynamic approach, meant that some of the advantages of a static approach were missing. The advantages of a static approach to unpacking, such as performance made it very appealing for the company to pursue.

The goal of this project was to develop a generic framework for static unpacking of runtime packed executables used by malware. Given the continuous appearance of new packers and variations within families of packers, the produced framework should be implemented in a modular way, allowing for the incremental addition of support for different packers. The purpose of this framework is to be integrated into the company's scanning engine to allow for static unpacking.

In order to add support for a packer in the framework, an unpacker must be developed for it. In most of the cases this means reverse-engineering the packer (see chapter 4). The first packers to be supported by the tool described will be selected based on statistical usage information collected from the company's malware database. This is to say that the order in which the packers should be integrated in the tool will be dictated by their "popularity" of malicious usage.

1.4 Document Structure

Asides from this introductory chapter, this document contains 5 more chapters. Chapter 2 explains the concepts involved in this project. In chapter 3, a description of the challenges facing the project can be found. Some related work is discussed and the project is defined in detail. Chapter 4 contains a detailed description of the implemented solution as well as the methodology applied. In this chapter, a detailed analysis for each of the packers supported by the developed framework is also done. In chapter 5, the integration of the framework in the company's scanning engine is described. The testing environment is explained and the results from these tests are discussed and analysed. Finally, in chapter 6, an analysis is done to the success of the project and further work considered.

Chapter 2

Fundamental Concepts

This chapter introduces and briefly explains the concepts involved in the domain of this project. These explanations are a collection of information from different sources and intend to introduce the reader to the concepts needed to fully understand the problem. For further reading, please refer to: [Cora, Pie02a, Pie02b, Ost02, Gop06, NHS, Kat, Fur, RSI12a, RSI12b, SH12, Eil05, Revb].

2.1 Malware

The word *malware* comes from the combination of the two words *malicious* and *software*. This type of software has a malicious objective such as performing operations without the owner's consent[Böh08].

2.1.1 Malware Analysis

When a new malicious program is discovered, it is analysed to identify its objective, the systems it exploits and how, and to extract different features that uniquely identify it. Knowing the systems malware exploits and how it achieves that is important information so that software vendors are able to correct those vulnerabilities. The mapping of the features that identify the malicious program serves the purpose of detecting it and possibly prevent its consequences if action is taken quickly enough.

In this field, there are two kinds of analysis, static and dynamic, which will be discussed here.

2.1.1.1 Static Analysis

Static analysis is usually the first step when analysing a piece of malware. It is all the analysis that does not run the malicious program. It includes identification of the malware with file hashes, string search (within the program), file header analysis, among others. Disassembly of malware without running it is also a static analysis technique that can give analysts an in-depth view of the program functionality. This approach has of course its drawbacks, the main one being the time it takes to manually scan through a program's instructions and infer about its functionality and the vulnerabilities it exploits.

2.1.1.2 Dynamic Analysis

Dynamic analysis is usually the second step in the analysis, when a static analysis has reached a dead-end. It encompasses every analysis that is done after launching the malware. Mostly it consists in monitoring the execution of the program and analysing its behaviour. This behavioural analysis is another component of malware detection, as specific behaviours can identify the malware in question. This analysis usually employs a set of tools such as debuggers and virtual-machines. There is an obvious big drawback to this approach: the malware is executed and therefore compromises the security of the system it is being analysed in. The use of virtual-machines is an attempt to overcome this issue by only compromising the virtual-machine, keeping the host safe.

It is easy to understand that not running the malware is a good thing. Especially if we do not know what are its objectives, behaviour and effects on a system. This simple observation makes the static analysis much more appealing. In a perfect world, this is true. However, in the real world, we must deal with budgets and time constraints and the time it takes to dissect a complex malware purely using static analysis is just impractical. Therefore a combination of the two different approaches is regularly used to analyse malware.

2.1.2 Malware Detection

Detectors, generically called antivirus software are used to protect systems against malware. Antivirus software employs different techniques to identify a malicious program from signature identification to behavioural analysis. Of course the analysis, whatever it might be, will always take some time and resources. Therefore, a play between the deepness of the analysis and reasonable time and resource constraints is done. The operation of this kind of software follows two approaches: on-demand and real-time scanning and protection. This means that on an on-demand operation the software may enjoy more relaxed constraints of time and resources and therefore perform a deeper analysis than when operating in real-time. However, in real-time, is should not slow down "too much" the system it is protecting.

In order to detect malware, several different techniques are used. They span from simple hashes over a file to complex heuristic approaches. The first approach to malware detection is signature-based or pattern-based detection. In this approach, a signature is a fingerprint that identifies the malware file or several files from a malware family. It has the obvious drawback of only being useful for known malware (or relatives, i.e. software with shared features). This approach is very efficient in terms of time and resources usage and is therefore popular amongst antivirus vendors that usually combine it with other detection solutions. The simplest way to uniquely identify a file that has been classified as malware is by the file hash. Hashes of known malware files can be stored and used to compare with files a detector is faced with. This approach is naturally very flawed. A simple bit change in the file, change that might not affect it in any way, makes a file hash different than the original one. Given this limitation, antivirus products rely on feature detection. These features can be many things, but ideally describe the malicious parts of a file that are unique in the way that they clearly have malicious intention. The simplest detections can be bit patterns or bit regular expressions, other more complex ones might be the way the execution of a file behaves. Given the sheer amount of malware circulating and new malware created every day it is not always possible to extract features that represent a malicious action accurately. Therefore, features that are shared between similar malware files are commonly selected. This feature selection must take into that the selected features should not occur in "normal", benign files. This is a hard problem to solve. Malware analysts usually describe a "good" feature as one able to identify many malware samples and no normal files. Naturally, in order to identify many different files, a feature must be small enough to tolerate changes around its definition. Of course the smaller the feature, the more files will be detected with it, making small features particularly prone to false positives. Because of all of this, malware detection is a constant adjustment of balance between identifying as many malicious files as possible and excluding benign ones.

Since the features extracted from static analysis are limited and do not fully describe the executable's *behaviour*, simulation or emulation are commonly applied. This approach allows the files to be executed in a controlled environment and all its behaviour analysed and logged. There are, however, many problems with these simulations. Let us consider Windows PE executables. In order to properly implement an environment in which these samples can execute, all the dependencies for the sample must be met. This means implementing all the APIs the sample uses as well as all the data structures on which it depends. This is an extremely laborious and delicate task. The alternative is to use a sandbox, a modified Windows environment which allows the interception and analysis of all the activity within it. All dynamic approaches have to address a fundamental problem: when to stop. This is not trivial. Naturally, any kind of simulation will incur in a significant performance penalty. All the dependencies that must be implemented in it will consume significant amounts of memory, and translating instructions will also contribute to a time penalty.

In Figure 2.1 we can see the typical malware analysis and detection process from an antivirus company's point of view, when a new sample is obtained. The whole process is automated because of the amount of malware generated on a daily basis. Naturally, we start by obtaining the new malware sample. There are many ways to do this, but this is out of the scope of this document. Having the malware sample, a series of analysis processes are performed on it. These processes extract as much features as possible. Starting from the file hash, all possible features are statically extracted from the file. After that, the file is executed in a controlled environment and all its behavioural features are extracted: API calls, memory layout, etc. All extracted features are stored. These features are then refined by going through a *control group*, which is a set of *benign* files. Refining the features allows them to be less prone to false-positives. Finally, these features are added to the

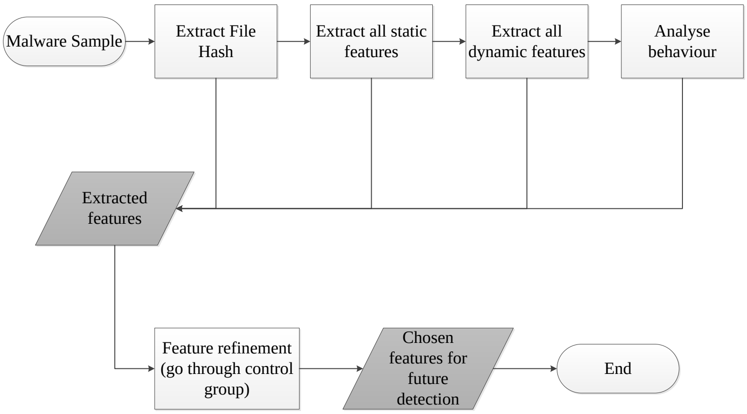

Malware Sample
Extract File Hash
Extract all static features
Extract all dynamic features
Analyse behaviour
Extracted features
Feature refinement (go through control group)
Chosen features for future detection
End

Figure 2.1: Typical malware workflow

"signature database" that is periodically pushed to antivirus clients.

Given a file and a signature database, an antivirus client usually performs the same process described, with the added task of trying to match the extracted features to the ones present in the signature database. Of course, because of time and resources constraints, this analysis might be less thorough in real-time than on-demand scanning.

Malware authors are naturally aware of the efforts of antivirus and security companies to detect their programs and prevent their effectiveness. So, they solve this problem using an array of techniques to thwart detection. They aim their efforts to evade detection at all the phases of the analysis and detection process. A very common approach is to try to evade static signature detection. A way to evade pattern detection is to use self modifying code. This approach converts the original code into some other representation and is converted back into runnable code at runtime. This approach can have many levels of complexity from simple code conversion where the intermediate representation is constant to metamorphic code where the intermediate representation changes with every new build of the file. Metamorphic code makes it extremely hard to detect malware using only signature-based detection since every version of the malware is fundamentally different. It is however not immune to behavioural heuristic detection, since its behaviour stays intact. Another way to try to avoid detection is to buy time, making the feature extraction and analysis as hard as possible by employing anti-reversing and anti-analysis techniques. These are discussed in more detail in section 2.4. All these evasion techniques are employed in packers. These are discussed in chapter 3.

2.2 The PE Format

The Portable Executable (PE) Format is a file format for executables used in Microsoft Windows OS's[Corb]. Portable because it was designed to be compatible with all versions of Windows and all supported CPUs. It is derived from the earlier Common Object File Format (COFF)[Cora] from the OpenVMS [Com] server OS in the beginning of the 1990's. Although it can be used with different purposes, this format is exactly the same for both the cases of stand-alone executables (.EXE) and dynamic libraries (.DLL). Only a bit in the PE Header indicates if the file should be used as an EXE or a DLL. Furthermore, the file extension has absolutely no impact on the file's behaviour. Common extensions for the PE format are .cpl, .exe, .dll, .ocx, .sys, and more.

Although the PE format can be used for several different CPU architectures, we will be focusing solely on the common 32bit type, since most packers only support these.

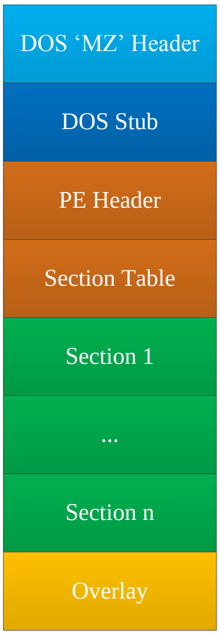


Figure 2.2: Typical PE Layout

The typical layout of a PE file is shown in Figure 2.2, being the top the beginning of the file. One of the interesting characteristics of the PE format is that the data structures in the PE file on-disk are exactly the same as in memory at the beginning of execution. This means that if we need to find some data on the PE file, we can do it with the file on-disk, with no need to load it in memory. This also means that loading a PE is simply a matter of mapping certain regions of the file into memory. However, loading is not simply mapping the file into memory. The offsets and locations of a loaded PE may, and nearly always do, differ from the file on-disk. An explanation of the loading process is done below in section 2.2.10. PE files loaded by the Windows loader in memory are called *modules*. The address at which the module is loaded, that is, the address of the first BYTE of the module is called the *HMODULE*. A module contains all the code, data and resources needed by a process to run.

An explanation of the PE format and relevant related concepts is done in the following subsections.

2.2.1 DOS Header and Stub

Every PE file starts with a small MS-DOS[Corc] executable, composed by the DOS ’MZ’ header and the DOS Stub shown in Figure 2.2. This is to allow the file to be executed in MS-DOS. The small program prints out something like "This program cannot be run in DOS mode." and closes gracefully. In the header of this small executable we have a structure called *IMAGE_DOS_HEADER*. Of interest from this structure, are the fields *e_magic* and *e_lfanew*. The *e_lfanew* field contains the offset of the new PE header within the file. This PE header is the one that is in fact important for the executable to be run in Windows environments. The *e_magic* field needs to have the first two bytes set to ASCII - ’MZ’, which are the initials of Mark Zbikowsky, one of the architects of MS-DOS.

2.2.2 PE Header

This structure, officially named *IMAGE_NT_HEADERS*, and its substructures store all the information needed by the Windows loader to load and run the module.

```
typedef struct _IMAGE_NT_HEADERS {
  DWORD                 Signature;
  IMAGE_FILE_HEADER     FileHeader;
  IMAGE_OPTIONAL_HEADER OptionalHeader;
} IMAGE_NT_HEADERS, *PIMAGE_NT_HEADERS;
```

A PE file must have the signature value of ASCII ’PE00’ to be valid. The following two structures, *FileHeader* and *OptionalHeader* store the actual information. Despite the name *OptionalHeader*, it is required for a PE to be loaded and run.

```
typedef struct _IMAGE_FILE_HEADER {
  WORD  Machine;
  WORD  NumberOfSections;   // Number of sections in file
  DWORD TimeDateStamp;
  DWORD PointerToSymbolTable;
  DWORD NumberOfSymbols;
  WORD  SizeOfOptionalHeader;
  WORD  Characteristics;    // File characteristics
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

From the *IMAGE_FILE_HEADER* structure, *NumberOfSections*, *SizeOfOptionalHeader* and *Characteristics* are of particular interest for the problem at hand.

- The *NumberOfSections* field is self-explanatory and has the number of sections present in the PE file.

- The *SizeOfOptionalHeader* stores the size of the *IMAGE_OPTIONAL_HEADER* structure and is essential for correct parsing of this structure.

- The *Characteristics* field stores flags about the behaviour of the PE file. This is, for instance, if it is a DLL or not.

```
typedef struct _IMAGE_OPTIONAL_HEADER {
  WORD                 Magic;
  BYTE                 MajorLinkerVersion;
  BYTE                 MinorLinkerVersion;
  DWORD                SizeOfCode;
  DWORD                SizeOfInitializedData;
  DWORD                SizeOfUninitializedData;
  DWORD                AddressOfEntryPoint;   // RVA of entry point of program
  DWORD                BaseOfCode;
  DWORD                BaseOfData;
  DWORD                ImageBase;             // Preferred load address
  DWORD                SectionAlignment;      // Section alignment in-memory
  DWORD                FileAlignment;         // Section alignment in file on disk
  WORD                 MajorOperatingSystemVersion;
  WORD                 MinorOperatingSystemVersion;
  WORD                 MajorImageVersion;
  WORD                 MinorImageVersion;
  WORD                 MajorSubsystemVersion;
  WORD                 MinorSubsystemVersion;
  DWORD                Win32VersionValue;
  DWORD                SizeOfImage;           // Size of image in-memory
  DWORD                SizeOfHeaders;
  DWORD                CheckSum;
  WORD                 Subsystem;
  WORD                 DllCharacteristics;
  DWORD                SizeOfStackReserve;    // Used in loading for stack
      allocation
  DWORD                SizeOfStackCommit;     // Used in loading for stack
      allocation
  DWORD                SizeOfHeapReserve;     // Used in loading for heap
      allocation
  DWORD                SizeOfHeapCommit;      // Used in loading for heap
      allocation
  DWORD                LoaderFlags;
  DWORD                NumberOfRvaAndSizes;
  IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
                                              // Data directories RVAs and sizes
} IMAGE_OPTIONAL_HEADER, *PIMAGE_OPTIONAL_HEADER;
```

Looking at the *IMAGE_OPTIONAL_HEADER* structure, we can see that it is here that most of the information for the loading is present. The fields and structures of particular interest are the ones that have comments.

- *AddressOfEntryPoint* stores the Relative Virtual Address (RVA) of the entry point of the program. When packing an executable, packers always change this value.
- *ImageBase* is the preferred load address of the PE file. The loader tries to load the module in this address. If it is not free, it will be loaded elsewhere.
- *SectionAlignment* is a value to which the sections should be aligned in-memory. This will be explained in detail later in 2.2.6 Sections.
- *FileAlignment* is a value to which the sections are aligned to on the file on disk. This will be explained in detail later in 2.2.6 Sections.
- *SizeOfImage* is the size the PE file should take in-memory when loaded. The loader uses this value to allocate memory for loading.
- The *DataDirectory* is an array that is the address book of all the important locations within the executable. Data structures such as imports, exports, resources and base relocations are some examples of data stored at these locations. Each element of the array contains the RVA and the size of a data directory.

2.2.3 PE Section Table

Immediately after the *IMAGE_NT_HEADERS*, we find the section table. This table is an array of *IMAGE_SECTION_HEADER* structures. The number of elements of this array is given by the *NumberOfSections* field in the *IMAGE_FILE_HEADER* structure.

```
typedef struct _IMAGE_SECTION_HEADER {
  BYTE  Name[IMAGE_SIZEOF_SHORT_NAME];  //section name, 8 bytes long
  union {
    DWORD PhysicalAddress;
    DWORD VirtualSize;      //virtual size of the section
  } Misc;
  DWORD VirtualAddress;     //RVA of section start
  DWORD SizeOfRawData;      //size of section on-disk
  DWORD PointerToRawData;   //FO of section on-disk
  DWORD PointerToRelocations;
  DWORD PointerToLinenumbers;
  WORD  NumberOfRelocations;
  WORD  NumberOfLinenumbers;
  DWORD Characteristics;    //flags indicating characteristics
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

The *IMAGE_SECTION_HEADER* structure stores all the information about a section. Commented in the structure declaration above are the most relevant fields.

- The *Name* field is a human-readabale 8-byte long string. There is no terminating null character and the string is exactly 8 bytes long. This field is only for easy human identification of what is contained in the section, it has no impact on loading whatsoever.
- *VirtualSize* is the size of the section in-memory, in bytes.
- *VirtualAddress* is the RVA of the section start in-memory.
- *SizeOfRawData* is the size of the section on-disk, in bytes.
- *PointerToRawData* is the FO (File Offset) of the section start, on disk.
- The *Characteristics* field is a set of flags that determine the characteristics of the section. It includes information about what is included in the section (code, initialized data, uninitialized data, etc.) and what should be the memory flag of the section in-memory (read-only, read-write, executable, etc.).

2.2.4 Dynamic Link Library

Dynamic Link Libraries, DLLs, are essentially PE executables. They provide a mechanism by which a program can use shared data and code. They are files separated from a calling program that are loaded (see section 2.2.10 for a more detail description on loading) at runtime in order for the program to use it. DLLs have explicitly exported functions and internal functions. The exported functions can be accessed and used by other executables, may they be programs or DLLs, but the internal functions can only be used from within the DLL. Functions can be exported by name or by ordinal number. The ordinal number is a WORD sized number uniquely identifying a function within a DLL. When functions are exported by name, it is the function name that uniquely identifies it. When an executable is loaded the needed DLLs that are not loaded yet are loaded into the process's address space, essentially including it in the calling executable. Suppose we have written a program that uses an exported function from an external DLL specified below:

```
God* getCurrentPope();
```

By using this function in our program, we assume it always returns the current Pope. For this to work properly, when the Pope changes, the function must also change the Pope it returns. The developers maintaining the DLL that exports this function can release another version of the DLL, with the updated function, and our program will still work properly with no need to rebuild it. The Windows loader will load the new DLL just as it did the old one and the program will call the function as it did before.

The Windows API is implemented in several DLLs that can be used by programs to interact with the system. These DLLs are different in different versions of Windows. Exported functions might be added, but Microsoft keeps previous exported functions for backward compatibility. However, ordinal numbers can vary from one DLL version to another, making importing by name safer. With this arrangement, programs that were designed with one version of DLL should always work with newer versions of the DLL. However, sometimes, bug-fixes and other updates that fundamentally change the behaviour of an API call that a program was expecting might cause the program not to run properly with newer versions of the DLL. Most of the Windows API DLLs are loaded differently than user DLLs, they are loaded at boot time, most of them in Kernel space, and are readily available for referencing by user executables. This arrangement allows for reduced memory usage since the functionality of a single mapped DLL can be accessed and its contents executed or read by multiple applications, only when memory writes are necessary in the DLL, are those memory pages copied to the process address space.

2.2.5 PE Data Pointers

When dealing with PE files, pointers to data within or outside the PE can be found in three different flavours: Virtual Address, Relative Virtual Address and File Offset (or Raw Offset). In the executable, many locations in-memory must be specified so that when it runs it knows where to find the required data, for example functions and global variables. However, many on-disk locations must also be specified for the loading process and for providing information about the PE. When parsing and navigating a PE file, we must be fully aware of which kind of pointer we are dealing with and treat it accordingly.

- **Virtual Address - VA** is the in-memory address, the address within the module, when the PE is loaded.

- **Relative Virtual Address - RVA** is the virtual address but being zero the beginning of the module, the *HMODULE*. RVA = VA - HMODULE. This value is independent of where the module is loaded and is, therefore, generic since we need only to add the *HMODULE* to get the VA and access the memory location.

- **File or Raw Offset - FO or RO** is as the name suggests, the offset within the file on-disk.

2.2.6 PE Sections

As seen before, the information about each section present in the PE file are specified in the section headers array. A section of a PE represents code or data (or both) of some sort. While code is just runnable code, data can be of many types. It can be read-write program data, such as global variables, API import tables, resources, relocation information, etc. A section has some specified in-memory attributes so that these can be set by the loader when loading. These attributes, amongst others, can be read, write, execute, or a combination of them. Sections also have names that, as stated before, are only intended to provide a human-readable identification of what the section contains. However, this name can be manipulated and can be essentially anything, provided it is 8 bytes long, even an empty one. In most applications, we would be able to see a ".text" or "CODE" named section that usually contain executable code. When dealing with malware however, we are often faced with manipulation of section names and other section header data that are intended to thwart analysis. Compilers usually have a standard set of sections they generate. The Microsoft compiler, for example, prefixes the section names with a dot (.) and creates at least two sections: ".text" and ".data". In the compilation process, the compiler usually defines the sections and places them in the generated .obj files. The linker then takes all the .obj files together with any included static libraries and merges all the sections with the same purposes into the final PE. Sections have two alignment values that are defined in the *IMAGE_OPTIONAL_HEADER* structure: *SectionAlignment* and *FileAlignment*. Sections always start at an offset that is a multiple of the alignment, might it be in-memory or on-disk. The purpose of these alignments is to keep the start of each section aligned with the start of an atomic block of data. Both disks and memory have atomic blocks of data. In disks it is usually the disk sector size, in memory it is usually page size. Since memory pages are atomic, it doesn't matter if a data block is smaller than the page size, one page will be assigned. The reason there are two values is that the default page size in Windows is usually 4KB (1000h) and aligning the file to this value would waste a lot of disk space. But disks also have atomic data blocks so the sections are usually aligned to multiples of the disk sector size (usually 512 bytes, 200h) to optimize the reading of the section from disk when loading. The fact that linkers can merge sections can be used to optimize memory usage when loaded since each section uses at least one memory page.

2.2.7 Imports Section

The Imports Section, although not a section as defined above, is a data directory where information about the imported DLLs and APIs is stored.
When code or data from another DLL is used, it is imported. When the program is compiled and linked, the information about the DLL names and the function names used is stored in this structure. This information allows the loader to map the addresses of all the required APIs referenced by the executable. The basis of this section is a structure called *IMAGE_IMPORT_DESCRIPTOR*. The *DataDirectory* entry for imports in the *OptionalHeader* points to an array of these structures. The end of the array is determined by one of these structures filled with zeroes.

In Figure 2.3 an overview of the structures that compose the imports section and their relationships can be seen. There is an *IMAGE_IMPORT_DESCRIPTOR* structure for every imported executable, usually DLLs. For the purpose of parsing the imported APIs, the fields from this structure that are relevant are the *Name1* and *FirstThunk* as seen in Figure 2.3. *Name1* is a DWORD with the RVA of a null-terminated string that contains the name of the imported DLL. *FirstThunk* points to an array of DWORDs (terminated by a zeroed DWORD) called the Import Address Table (from here forward called IAT). When the file lies on disk, this is one of two things: either the ordinal number of the imported API in the DLL, or the RVA of a *IMAGE_IMPORT_BY_NAME* structure. In order to distinguish between both cases, the high bit of the DWORD is used. If it is

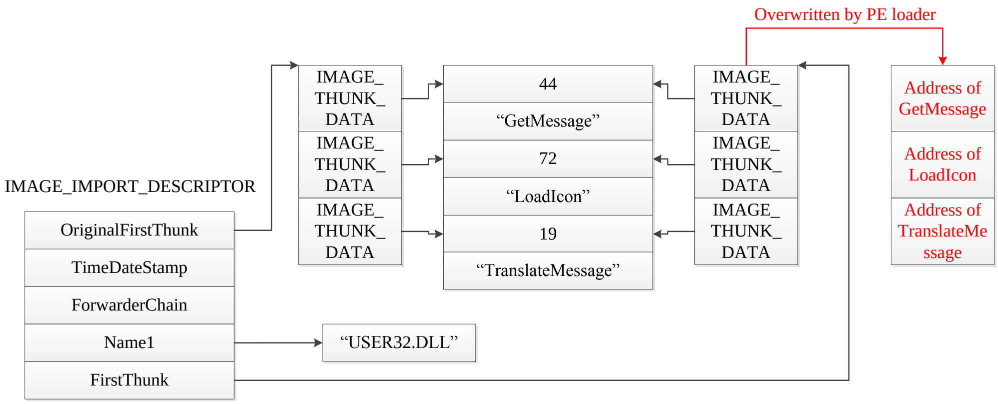


Figure 2.3: Complete overview of the Imports Section

set, then the lower 31 bits are treated as an ordinal, if not, it is treated as an RVA pointing to the referred structure. This structure is basically a WORD followed by a null-terminated string with the name of the imported function. The WORD is used by the loader to estimate the position of the API in the DLL and be quicker at finding it. Once an API address has been found in memory by the loader, it is placed in the correct place of the IAT. The executable will use this table to call the imported functions.

It can be seen in Figure 2.3 that there is yet another table with apparently the same data as the IAT, pointed by *OriginalFirstThunk*. This table is the Import Name Table (INT). The only difference between this table and the IAT is that the INT never gets overwritten. The purpose of this table is to keep the original information about the imports when the executable has been bound (Binding lies out of the scope of this document, refer to [Pie02b]) but the stored API addresses in the IAT are stale and need to be fixed. Furthermore, the INT does not need to be present for the PE to load, although without it, the PE cannot be bound.

2.2.8 Resources Section

The resources section, although usually placed in a section of its own (Microsoft compiler and linker usually places it in a section called ".rsrc"), can be found within other sections. It is an area in PE that stores resources such as icons, bitmaps and dialogs. Its location and size is stored in the *DataDirectory* in the *IMAGE_OPTIONAL_HEADER*. It is organized in a way similar to a filesystem with directory and leaf nodes. This allows objects of the same type to be put together for better organization and faster loading.

2.2.9 Overlay

Although not officially part of the PE format, the overlay is sometimes found in PEs. The overlay refers to any data appended beyond the PE image, i.e. any data appended to the PE file after the end of the last section. This information is discarded by the Windows loader when creating the module. Common examples of overlay are certificates and debug information. It can be simply a certificate or have several regions with different data.

2.2.10 PE Loading

The PE loading is a central part of PE behaviour. The way the PE is structured is precisely to allow for correct loading and execution in the environments it was designed to run in. Loading is a multiple step complex process. In the Microsoft Windows OS, there is a routine that is responsible for loading PE files into memory and creating a process to run them. This is called the Windows loader. It must said that the Windows loader varies slightly from one Windows version to another. The loader starts to work when an executable is executed (eg. by double clicking it in the Windows Explorer). It first tries to read the DOS, PE, and Section Headers. These structures, as seen before, contain information about how to load the PE into memory. One value found in these structures is the preferred loading address, *ImageBase*. This is the address in which the loader tries to map the

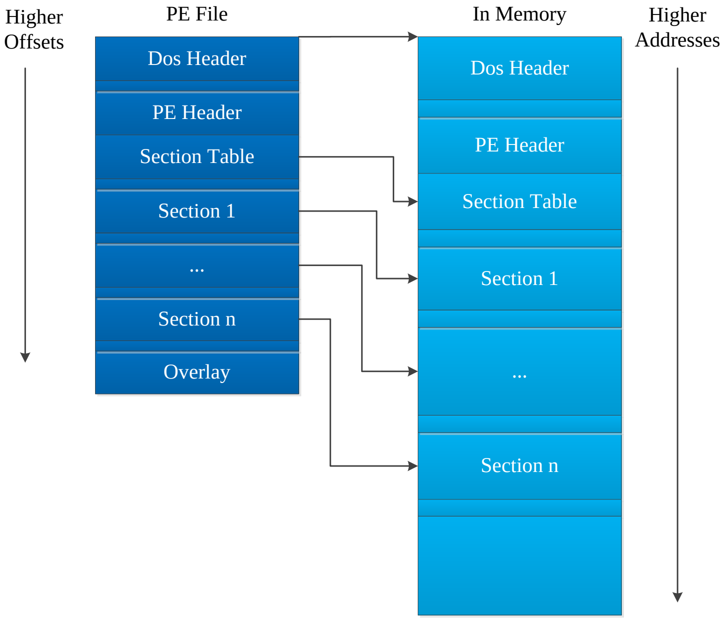


Figure 2.4: PE Loading

PE file into memory. Another field is the size the PE should take in memory, *SizeOfImage*. If the preferred loading address is not available, either because another module is already loaded there or because there is not enough contiguous available space to store the loaded PE, the loader chooses another location and allocates the area for the PE. The loader then goes through all the Section Headers and loads each section into memory in its correct place. In this section loading process, the loader copies the section located in the file at the offset specified by *PointerToRawData*, with size *SizeOfRawData*, into the memory location specified by *HMODULE+VirtualAddress*. The loader applies padding (zeroes memory) to the difference between both values if the *VirtualSize* value is larger than the value of *SizeOfRawData*. It then applies padding until the next address that is a multiple of *SectionAlignment*. As seen before, the overlay is not loaded. It is common to have some hardcoded virtual address (VA) within an executables' code (for function calls, global variable access, etc). This of course causes a problem when the PE is not loaded at its preferred load address: the referred VAs are invalid. For this reason, once all the sections have been loaded, if the PE was not loaded at its preferred address, the loader performs relocation fix-ups, or base relocations (if the file supports it). The PE file has a *DataSection* for relocation data. This data is basically a list of locations where VA are used in the PE. With this information, the loader can just add the difference between the *ImageBase* and the *HMODULE* to all the VAs at the locations pointed by the relocation data. By doing this, all the VAs are valid once again. The loader then goes on to guarantee all the dependencies of PE. The dependencies that are specified in the imports section are parsed and each of the DLLs that are not loaded yet are loaded using the *LoadLibraryA* API from Kernel32.DLL. Then, each of the APIs imported from each of the DLLs are mapped using the *GetProcAddress* API from Kernel32.DLL. If any one of the imported DLLs or APIs is not found, the loader exits displaying an error message. If all dependencies are met, the loader uses the information in the PE header to initialize the stack and heap for the executable. An initial thread is created and the process is started at the entry point. From here on the loader's job has ended and the executable runs its code.

2.3 Reversing

Reverse engineering is the process of extracting knowledge or design blueprints from anything man-made[Eil05]. It is usually used to obtain missing information when this information is owned by someone who doesn't want to share it or it may be lost or destroyed. Software reverse engineering is commonly called reversing. The purpose of reversing might be to develop a competing product or to better understand undocumented programs. The sheer complexity of present programs make it many times not worth it to reverse and implement a competing product because it would just take too much effort, more effort than just implementing it from scratch. Reversing requires a set of skills such as code breaking, puzzle solving, programming and logical analysis. In IT Security, reversing is used extensively on both sides of the fence: both by malware developers and those fighting them. It is also very popular with crackers that aim to remove from software copy and other protections. Malware developers extensively use reversing to locate vulnerabilities in OS's and other software that can be exploited. On the other side, developers of antivirus products and security researchers use reversing techniques to dissect malware samples in order to identify the vulnerabilities they exploit, their purpose, the damage they could cause, how it can be removed, etc. In cryptography, reverse engineering has been used to try and crack the encryption, either to test its effectiveness or to extract the data that was encrypted. Traditionally, cryptography has been based on secrecy, where the encryption and decryption methods were only shared between the people that were meant to communicate privately. Historically, cracking encryption has been based on either obtaining the decryption algorithms, trying to find flaws in the algorithm that can be exploited to extract the key or message, or trying all possible combinations in order to get the key or message. An example of such usage of these techniques is in military history, in which one side tries to decrypt the other side's encrypted messages in order to get an advantage in the battlefield.

For reversing and malware analysis, a set of tools are employed. These tools, that have different characteristics and purposes, are explained in the following subsections.

2.3.1 Disassembly

When a program is compiled, its human-readable text representation, the source-code, is converted into machine-readable binary code. This machine-readable binary code is just a stream of numbers that are interpreted by the CPU as instructions. Each platform (CPU type) has a different instruction and register set and their mapping. The machine code has a direct human-readable representation that is generically called assembly language.

In essence, disassembly is taking the stream of numbers and converting them into assembly language. A disassembler must support the platform the binary it's analysing was compiled for. Disassemblers usually support several platforms. The core problem of disassemblers is the distinction between code and data. As far as the pure disassembly process is concerned, the function of a disassembler is to convert binary data into the text representation of the assembly language, and program data such as hardcoded constants is also just binary data, meaning the disassembler will treat it as code if not properly configured. For performance reasons, compilers commonly insert data into code sections. To try and solve this problem, disassemblers try to follow the code flow and only mark reachable code as code, being the rest treated as raw data. This approach used together with user intervention can properly distinguish the code from data. As will be discussed later in section 2.4, there are a few techniques that can be employed to confuse the disassembler. Because of the ability to extract some knowledge of the function and objective of a software sample, disassembly is extensively used in malware analysis and is commonly a component of detection.

2.3.2 Debugging

Anyone who has done some programming has come across debugging. Debuggers are primarily used by programmers as a tool to find and correct errors in their programs. In this scenario, debuggers usually display the source-code, but the reality is that they are working on the low-level assembly code and there is an additional layer that can map the execution in the low-level code to a position in the high level source-code. These low-level workings can make debuggers a powerful reversing tool. The ability to show a disassembled view of the running code as well as all the context in which it is running are of particular interest to anyone attempting a reversing task. For reversers, a good debugger should have a powerful disassembler that allows a clear view of the code and a way to show the cross-references of where the branching instructions go and where they can be called from. The basis of a debugger are breakpoints. There are two main types of breakpoints: software and hardware breakpoints. Software breakpoints are instructions introduced into the running code that transfer control to the the debugger when they execute, and the debugger pauses execution. This allows the user to have a look at the state of the execution and step through the instructions if needed. Hardware breakpoints are a special CPU feature that allows the processor to pause execution and hand over control to the debugger. Hardware breakpoints are especially useful because it is possible to set breakpoints in memory regions of interest and pause execution when these regions are accessed (read or written). Using these breakpoints we can easily find code sections that manipulate these memory regions. This is particularly useful when dealing with packers as will be explained in detail in chapter 3. A view of the registers and memory is extremely important, and a good debugger will provide some interpretation of what is stored there (pointer or data) while still allowing a raw view of its contents. Being able to see the context of the running process is extremely helpful when debugging. This context may be the loaded modules, the running threads, a stack and CPU registers view for each thread, or call stacks. Due to the powerful insight of a software sample a debugger can provide, they remain the most useful and direct way of analysing and reversing software.

2.3.3 Decompiling

Decompiling is the software reverse engineering Holy Grail. It attempts to completely reverse the compilation process and produce a high-level source-code-like representation of the executable being analysed. The problem is that the compilation process removes some of the information from the program, so, some high-level information is completely lost: structure declarations, variable names, variable types, comments. Furthermore, compilers attempt to optimize the program as much as possible, and in doing so, fundamentally change the order and characteristics of the instructions. This means that although functionally correct, the binary code is extremely hard to interpret. The fact remains that the binary code executes as expected in the CPU. The CPU, however, does not really need to *understand* what it is doing. All this, means is that although it is possible to decompile a binary, this can be done only to a certain point. The result will be functionally identical to the original but will be inevitably a lot less readable. It must be said that although these considerations are true for native x86 binaries, other platform binaries such as Java bytecodes or .NET executables retain a lot of information from the high-level source-code and are therefore easier to decompile and obtain decent results.

2.4 Anti-Reversing

Anti-reverse-engineering, or anti-reversing, is the name given to all the techniques employed to increase the time and level of skill required to reverse engineer a piece of software. Anti-reversing makes a lot of sense not only to protect intellectual property but also for malware authors. In practice, people are faced with time and budget constraints and making a piece of software much harder to reverse might force people to give up because of these constraints. Even for the hobby reversers that have no budget constraint, where their motivation is the drive behind the reversing, efforts to frustrate them might result in them giving up the challenge. However, no matter how hard you try, you cannot make an application that cannot be reversed: it still needs to run properly. Ultimately, if it runs, it can be reversed. Another consideration to have is that all anti-reversing techniques come at a cost, may it be in space or in time. Depending on the approach, software protected by anti-reversing techniques will inevitably have a poorer performance than its unprotected counterpart. Whether this performance penalty is acceptable or not is up to the people who develop the application. For malware authors, it is definitely worth it, since performance has usually a lower priority than evading detection. Making security analysts take more time to identify and analyse their software is, naturally, a high priority because that means more time in the wild and more infections and potentially more profit. Anti-reversing is extensively used in copy protection technologies and digital rights management, for obvious reasons. In this section, some anti-reversing techniques are briefly explained. Generally speaking, it is possible to divide anti-reversing approaches into two categories: reducing readability or actively checking and messing with the reversing process.

As far as reducing readability goes, the use of obfuscation and encryption is commonly applied. Obfuscation is a technique of anti-reversing that aims to make code as unreadable as possible for a human being, but still run correctly. A simple example of obfuscation is embedding in the code many useless operations, causing confusion when reading. As previously stated, bytecode-based compiled programs like Java or .NET contain a lot of symbolic information makes them extremely easy to reverse. Due to this fact, many obfuscators have been developed for these platforms in order to reduce the readability of the compiled program. Code encryption is a common way of preventing static analysis by making the code essentially unreadable and is accomplished by encrypting the code after compilation and embedding a decryption routine that runs before the program itself. On the other side, there are those techniques that actively interfere with the reversing process. Anti-reversing techniques have long been observed in malware and are implemented in many packers. A few of them are described in the following sections.

2.4.1 Anti-Disassembly

As the name suggests, anti-disassembly is a technique used to make disassembly harder. In addition to making disassembly harder and, consequently, harder for a reverser to understand, these techniques are commonly effective against automated detection from antivirus products since these usually rely on some disassembly techniques. Code tailored to confuse disassemblers is routinely used by malware. The main approach at anti-disassembly is to mess with the code flow. One example of such a technique is using calls as unconditional jumps, also called return pointer abuse, and is achieved by calling a function that manipulates the stack and modifies the return pointer, pointing it to the new offset, the actual target. Disassemblers cannot follow these jumps, making them possibly mark these target regions as data. Another approach is the use of two conditional jumps back to back, or conditional jumps with constant conditions, that force the disassembler to disassemble an area that will not be executed. In this area, multi-byte instructions are placed to confuse the disassembler, making the resulting disassembly wrong. A similar technique is to make a byte part of two instructions, disassemblers cannot correctly parse it since they only allow a byte to belong to one instruction. This is an extremely powerful technique that cannot even be solved with human intervention. It is however hard to implement correctly.

2.4.2 Anti-Debugging

Anti-debugging is a technique used by malware and other protected software to detect when it is under the control of a debugger or to interfere with the debugging process. Once the software detects it is being debugged, it might alter its behaviour or terminate execution, adding to the time and effort required to analyse it. There are many anti-debugging tricks that could fill a book, we will be focusing only on a few common ones.

Using the Windows API and checking Windows structures can be used in many different ways to detect the presence of a debugger. The simplest and most direct detection mechanism is calling the *IsDebuggerPresent* Windows API. This is however of little effectiveness since it is very easy to identify and bypass when debugging. It is also possible to detect a debugger by its behaviour. As we saw in section 2.3.2, a software debugger inserts special instructions (specifically, the "int 3" instruction) in the code. An application can check its code for these instructions and determine the presence of a debugger. Another different approach has to do with time. In the use of a debugger, it is very common to set breakpoints and single-step over the code. This makes the execution slower. Because of this, timing checks can be used to infer the presence of a debugger. It is possible to measure the time taken between two key instructions, and, if it took more than a certain threshold, it is assumed the application is being debugged. Another mechanism used by malware to make analysis more cumbersome is exploiting the PE header. This mechanism is very powerful in crashing many analysis tools, debuggers, disassemblers or PE explorers. The technique relies on the fact that Microsoft does not follow the COFF specification too strictly and performs many undocumented checks on the PE Header and file when loading it with the PE loader. Manipulating the PE header in specific ways can cause analysis tools to crash but still be correctly loaded by the Windows Loader.

2.4.3 Anti-VM

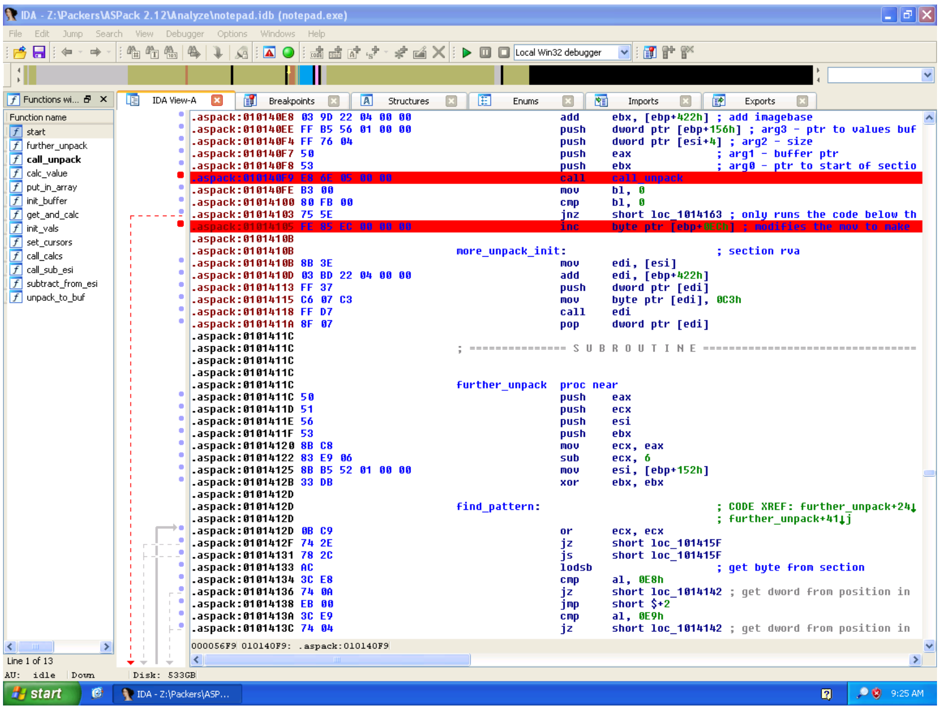
Since many malware analysts use virtual machines for analysis, authors try to embed in their software virtual machine detection mechanisms. This relies on the assumption that if it is running in a virtual machine, it is being analysed. This is of course not always true. The use of virtual machines has become more popular in recent years, making the anti-virtual-machine technique potentially counter productive for malware authors by eliminating a large group of potential targets. Since historically the most popular virtual machine software was VMWare, malware tends to try to detect its presence. The most basic detection is commonly done by looking at the data such as manufacturer, name or serial number of the different devices such as display adapter, hard-drive or mouse. Because these devices are emulated in software virtual machines such as VMWare they usually have dummy names that can be identified. There are of course many other techniques for detecting virtual machines environments; those are, however, out of the scope of this document.

2.5 Tools

A short description of the most important and relevant tools used in the project is done in the next subsections.

2.5.1 IDA Pro

The Interactive Disassembler Professional, IDA Pro[SA], is an extremely powerful disassembler distributed by Hex-Rays. This application supports many CPU architectures such as IA-32, IA-64, AMD64 and many others. It also supports many executable file types from the PE and Linux ELF to XBE, the Microsoft XBOX executable. Besides being a very powerful disassembler, it also allows debugging with many different debuggers, local or remote. Furthermore, it is possible to manipulate and interact with the disassembly process in very useful ways such as function definition and naming, variable naming and insertions of comments. It has an extremely useful feature that allows the user to visualize a graph of the code flow, with code blocks connected by arrows that represent jumps, may they be conditional, unconditional or calls. Another neat feature is the ability to recognize and identify external library API calls, allowing the user to clearly see them. It also supports plugins which allow the user to develop a tool that might aid him/her and integrate it into the application, making it even more powerful. The ability to save the progress of an analysis to later resume it is another advantage of this program. The combination of these features within the application makes it a favourite of many malware analysts and reverse engineers.



```
.aspack:010140E8 03 9D 22 04 00 00                add     ebx, [ebp+422h] ; add imagebase
.aspack:010140EE FF B5 56 01 00 00                push    dword ptr [ebp+156h] ; arg3 - ptr to values buf
.aspack:010140F4 FF 76 04                         push    dword ptr [esi+4] ; arg2 - size
.aspack:010140F7 50                               push    eax             ; arg1 - buffer ptr
.aspack:010140F8 53                               push    ebx             ; arg0 - ptr to start of sectio
.aspack:010140F9 E8 6E 05 00 00                   call    call_unpack
.aspack:010140FE B3 00                            mov     bl, 0
.aspack:01014100 80 FB 00                         cmp     bl, 0
.aspack:01014103 75 5E                            jnz     short loc_1014163 ; only runs the code below th
.aspack:01014105 FE 85 EC 00 00 00                inc     byte ptr [ebp+0ECh] ; modifies the mov to make
.aspack:0101410B
.aspack:0101410B                        more_unpack_init:                 ; section rva
.aspack:0101410B 8B 3E                            mov     edi, [esi]
.aspack:0101410D 03 BD 22 04 00 00                add     edi, [ebp+422h]
.aspack:01014113 FF 37                            push    dword ptr [edi]
.aspack:01014115 C6 07 C3                         mov     byte ptr [edi], 0C3h
.aspack:01014118 FF D7                            call    edi
.aspack:0101411A 8F 07                            pop     dword ptr [edi]
.aspack:0101411C
.aspack:0101411C                        ; ============== S U B R O U T I N E ================================
.aspack:0101411C
.aspack:0101411C
.aspack:0101411C                        further_unpack  proc near
.aspack:0101411C 50                               push    eax
.aspack:0101411D 51                               push    ecx
.aspack:0101411E 56                               push    esi
.aspack:0101411F 53                               push    ebx
.aspack:01014120 8B C8                            mov     ecx, eax
.aspack:01014122 83 E9 06                         sub     ecx, 6
.aspack:01014125 8B B5 52 01 00 00                mov     esi, [ebp+152h]
.aspack:0101412B 33 DB                            xor     ebx, ebx
.aspack:0101412D
.aspack:0101412D                        find_pattern:                     ; CODE XREF: further_unpack+24↓
.aspack:0101412D                                                          ; further_unpack+41↓j
.aspack:0101412D 0B C9                            or      ecx, ecx
.aspack:0101412F 74 2E                            jz      short loc_101415F
.aspack:01014131 78 2C                            js      short loc_101415F
.aspack:01014133 AC                               lodsb                   ; get byte from section
.aspack:01014134 3C E8                            cmp     al, 0E8h
.aspack:01014136 74 0A                            jz      short loc_1014142 ; get dword from position in
.aspack:01014138 EB 00                            jmp     short $+2
.aspack:0101413A 3C E9                            cmp     al, 0E9h
.aspack:0101413C 74 04                            jz      short loc_1014142 ; get dword from position in
000056F9 010140F9: .aspack:010140F9
```

Figure 2.5: Screenshot of IDA Pro

2.5.2 Titan Engine

The TitanEngine[Reva] framework was developed at ReversingLabs as a complete framework for unpacking. It has embedded within a debugger, a disassembler, a memory dumper, an import fixer and a file realigner. The more than 400 functions and a well defined API allows the programmer to use the TitanEngine for many reversing tasks, especially unpacking. It has powerful functions for dynamic unpacking and supporting features for static unpacking. Support for both x86 and x64 PE files make it capable of creating all types of unpackers. The fact that it is open-source gives the programmer the freedom to modify and easily integrate it into existing solutions as needed. As a framework developed for the purpose of unpacking, it made the perfect candidate for use in this project.

2.6 Summary

All the concepts described in this chapter will be used in one way or another throughout the document. Armed with the knowledge about the concepts involved in this project it is possible to grasp the challenges faced and fully understand the rest of the document.

Chapter 3

The Packer Problem in Detection

This chapter will explain in detail the packer problem. The impact of packers in detection and analysis is also explained so that the considerations and implications of the problem and its solutions are well understood. The project's ultimate objective is defined and the adopted solution is briefly described.

3.1 Packers

Runtime packers, or packers, have been around since the early computers. Given a program, a packer is the generic name given to programs that envelop the original program either to reduce its size or to thwart its detection or analysis. They were first developed as a way to save disk space when disk space was a very limited and valued resource. The found solution was to compress the original executable and embed a routine that would uncompress it at runtime. This compression, however, does not alter the functionality of the original program. When loading the resulting packed program, an unpacking routine restores the original program into memory. The fact that the packed binary *looks* fundamentally different than the original one (on disk at least) was noticed by malware authors that saw in it an opportunity to evade detection. Although packers are used both by malware and legitimate software, the descriptions will be focusing only on malware.
The simple concept of compression eventually evolved into encryption. Analogous to compression, the original binary can be encrypted with some key and then decrypt it at runtime. These simple packers are commonly called cryptors. The variation of the key allowed for multiple different *looking* files from the same executable. This allowed for further evasion because even if one file encrypted with one key got detected and blacklisted, it would still be different from its relatives that used different encryption keys. Eventually, some anti-reverse-engineering and anti-analysis protections were incorporated into packers. A fundamental problem remained: when the executable is running its contents are fully exposed. To try to solve this problem, a technique called "shifting decode frame" was developed. This implementation does not unpack the whole

code. It only unpacks one block of code at a time and overwrites the last one after moving on to the next. With this approach, only parts of the code are unpacked in memory at any point in time. Eventually full-blown virtual machines appeared. In this solution, a virtual CPU is implemented in the unpacking stub. When the code is packed it is translated into the instructions for the virtual CPU. When executed, each instruction is executed by the virtual CPU and no understandable code is present in memory at any point in time. Furthermore, in some implementations, the instruction set for the virtual CPU vary even within the same packer, being almost unique to each packed binary. Naturally, a combination of these and other techniques can be implemented in a packer.

The evolution of packers and protectors is intertwined with the evolution of malware detection. It is the stereotypical game of cat and mouse where the security industry tries to detect and the malware authors try to avoid detection. As the detection technologies become more effective, the evasion techniques evolve. As these evasion techniques evolved, it ultimately became practical for malware authors to gather them. With this gathering, the modern packer, the *protector* came to be. It is simply more efficient to develop your malware without having to worry about detection and protect it afterwards by packing it. Packers are easy to use, and the availability of many different packers mean that exactly the same malware can be packed with many different packers obtaining many different signatures, therefore evading signature-based detection. In many cases, a use of several layers of packing, done with different packers in each layer or not, has been observed. By varying the packers used and the order in which they are used, it is possible to get a significant increase in binaries with different signatures for the same original binary.

There are many different implementations of packers. Some focus more on reducing the size of the executable, some focus more on anti-reverse-engineering. However, they generally function on the same basic principle: encode the original binary, store that data in the new binary and add an unpacking stub (and point the entry point to it) that allows the original program into run. A simple illustration of the packing and unpacking can bee seen in Figure 3.1.

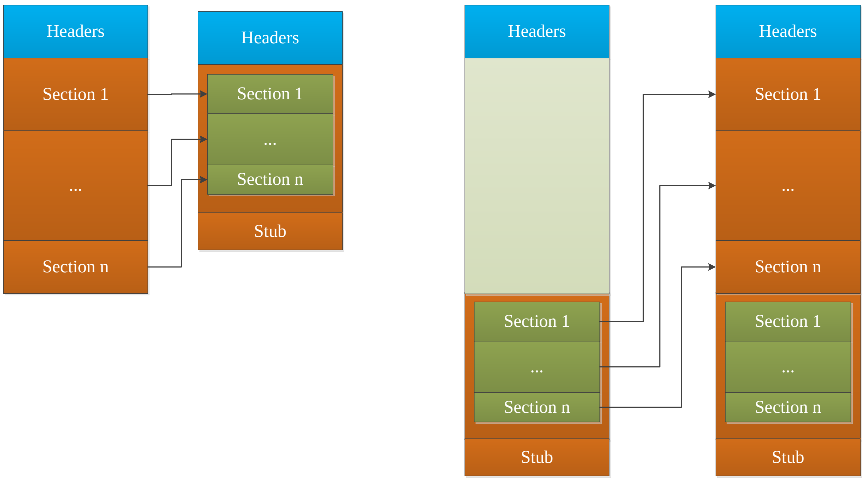


Figure 3.1: Left: packing; Right: unpacking in runtime

Although the layout in Figure 3.1 can be considered a generic representation of packers, many different layouts are observed. However, in most of the packed binaries, the last section contains the unpacking stub. Usually, the unpacking stub performs a series of steps that are common to most packers. The first step is commonly a decryption of the internal data of the unpacking stub. This is followed by the unpacking of the sections. After this, since many packers obscure the DLL imports, the unpacking stub takes care of importing all the necessary DLLs and APIs. Finally, the unpacking stub passes control to the original code in the original entry point (OEP). This final step is called the OEP Jump. After this step, the original code runs normally, as it was originally designed.

3.1.1 Packer Identification

There are several heuristics that can be applied to decide if a file is packed or not. As with detection, the simplest way is signature-based, where the unpacking stub is analysed statically and a signature is produced and compared with a signature database. This is very fast and effective for known packers already in the signature database. Such an approach is used by PEiD[Sna], a very popular analysis tool. Although this method is the best approach to correctly identify which specific packer was used to pack a binary being analysed, it suffers from the same problems of signature-based detection. Furthermore, many of the signatures used by PEiD are very prone to false-positives, ultimately incorrectly identifying packers. Another more "model-based" approach relies on the entropy analysis of a file. Here, with a trained heuristic implementation, a binary with more entropy than a specific value is considered to be packed[LH07]. By itself, this approach does not identify the packer used. Finally, a behaviour-based approach, therefore dynamic, looks for tell-tale signs of the phased execution of packed binaries. Looking for the signs of the OEP jump is one such approach.

3.2 Unpacking

Unpacking is the reverse operation of packing. It is taking a packed binary and restoring it to its original, unpacked, state. This process can be divided into two main phases: unpacking and rebuilding. In this context, unpacking is the extraction of the original, underlying code and data present in a packed binary. Rebuilding is to produce a runnable binary with the code and data obtained by unpacking. This two-phase division allows for an unpacking solution to be tailored to the specific needs of each situation. As far as detection of payload goes, the rebuilding phase is up to some point unnecessary. In the context of this project, these two phases were treated separately. Rebuilding the executable can be problematic. There are many challenges to rebuilding a working unpacked executable from a packed binary. Furthermore, depending on the packer used, it might not be possible to perfectly restore the original, unpacked binary. This might be, for example, because at the time of packing, some sections of code were replaced or redirected. This kind of fundamental change to the code makes it impossible to revert to the original state. Another issue when rebuilding is the import resolution. Many packers obscure the imports of the packed binary to try to obscure the packed executable's behaviour. This means that the unpacking stub imports the dependencies itself. In order to rebuild a working executable, these imports must be extracted and patched into the unpacked file in the proper format. There are also the obvious fixes that must be done to the executable header to accommodate the changes that the unpacking procedure made.

3.2.1 Unpacking in Perspective

The ultimate objective of an antivirus company's products is to protect a system against malware. This is traditionally done by detection and containment. Due to the fact the the most used operating system remains Microsoft Windows, it is natural that most of the malware is targeted at systems with this OS. This means that vast amounts of malware are of the PE file format (see section 2.2). As described in chapter 2, the detection of malware is faced with many challenges. One of these challenges is the use of packers. The detection of the underlying payload of packed malware is commonly done resorting to several techniques. Static signature detection is of little use in these cases, asides from detecting the packer that is used. Alone, static signature-based detection cannot detect the underlying malware payload. To address this, unpacking must be done, revealing the original code as much as possible.

3.2.2 Dynamic Unpacking

For dynamic unpacking, simulation or emulation is commonly applied. This approach allows the files to be executed in a controlled environment. Because the first stage of packers is to unpack the underlying payload, allowing it to execute will allow it to unpack itself. Once the sample is unpacked, it is possible to take a snapshot of the module and scan for signatures on the unpacked code. These dynamic approaches have to address a fundamental problem: when to stop. In the case of unpacking, it should stop when the execution is passed to the original unpacked code: the OEP jump. Detecting when this happens is a hard problem and has been addressed in research (see sections 3.2.2.1, 3.2.2.2, 3.2.2.3 and 3.2.2.4). Another problem is getting past anti-analysis techniques. A simple loop long enough to outlast the simulation will result in an analysis on a still packed module. There are many ways of a sample detecting if it is being run in an emulator, sandbox, or VM. As usual, these techniques are effective up to the point when they are detected by the simulation environment developers that try to circumvent them. Even ignoring the fact that it is possible to detect the simulation environment, the issue of performance remains. An advantage of this approach is the fact that the packer used is of no concern, allowing it to deal with unknown packers and making it generic. However, this solution does not work for all packers. As discussed in section 3.1, there are packers that never fully unpack the code making this approach insufficient. That, together with a detection that evaluates if a binary is packed or not can allow the simulation only to be run when an executable is packed, minimizing the time and memory penalty in global operation. As discussed in section 3.1.1, the detection of a packed binary can be made with different approaches. From signature-based identification to randomness or entropy analysis of the binary.

In the next few paragraphs, some attempts at dynamic, automatic, generic unpacking solutions are analysed.

3.2.2.1 PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware [RHD+06]

This unpacking solution presents a model that uses both dynamic and static approaches to unpack malware. The model relies on a primary static analysis of the packed binary. This extracts the visible code of the unpacker stub. It then runs the malware in an emulator and at each instruction compares the code in memory with the code obtained from the static analysis. This comparison is done after each instruction is executed to assert if any new code is present in memory. If new code is detected, it is assumed it is unpacked code. This hybrid combination avoids the heavy maintenance required with a purely static approach but, as discussed before, cannot unpack the more complex packers. As an unpacker, it does not evaluate if the code is malicious or not. At the time of its development, the most similar approach was the plugin for IDA Pro[SA] called Universal PE Unpacker. Because this approach runs instruction by instruction, it is slow. The tests performed took an average of just under 20 minutes to unpack. There is also the possibility of analysing non-code regions. Due to the dynamic component of the solution, it is vulnerable to anti-debugging and anti-emulation techniques.

3.2.2.2 Renovo: A Hidden Code Extractor for Packed Executables [KPY07]

This solution comes after PolyUnpack and tried to address some of the issues that unpacker had. It is a fully dynamic approach that monitors memory writes. It runs the malware in an emulated environment. It is built on top of a TEMU[SBY+08], a dynamic analysis component of BitBlaze which is a binary analysis tool. It monitors the execution of the program and looks for a tail jump that might indicate the end of the unpacking stub. To complement this detection technique, it marks all written memory pages as dirty and monitors them for execution. Once one is executed, this might mean the unpacking stub has ended and the execution has begun on the unpacked code. This approach can penetrate into several layers of packing by recursively going through the program. It uses a defined time-out to stop this recursion since the last layer cannot be decided. Like PolyUnpack, it operates after every instruction, being therefore relatively slow. But due to the fact that is does not need to compare code at each iteration, it needs only to see where the program counter points to and if it is a dirty position, making it faster than PolyUnpack[RHD+06].

3.2.2.3 OmniUnpack: Fast, Generic, and Safe Unpacking of Malware [MCJ07]

This solution is, a purely dynamic one. Unlike other solutions before it, it is attached to an anti-virus engine, ClamAV[Inc], to detect if the packed payload is malware. It is based on OllyBone[Ste] which is a plugin for the debugger OllyDbg[Yus]. It uses the technique of tracking memory writes and monitoring if they are executed to find the end of the unpacking phase. Furthermore, to detect the end of all the unpacking layers, it monitors dangerous system calls like creation or setting of registry keys. This is based on the assumption that the unpacking stub only loads the packed code into memory and link the required libraries. This is a good assumption since a packer does not affect the system, it merely unpacks a code that might. It is implemented as a kernel driver for Microsoft Windows XP, which is more resilient to anti-debugging techniques. Once the code is unpacked, it is fed to the ClamAV anti-virus software for analysis. Another difference from other related systems such as Renovo[KPY07] is the granularity at which the execution is tracked: at the memory-page level. This makes the tracking much faster. The solution also employs a technique to try to figure out if the malware is packed or not based on the simple premise: if there are no memory-pages written and then executed, it is not packed. Tests showed that it performed faster and with better results than PolyUnpack[RHD+06].

3.2.2.4 A Study of the Packer Problem and Its Solutions [GFC08]

This very complete study done by Symantec Research Laboratories, not only overviews the packer problem but also proposes a solution. It discusses the problems and challenges posed by packers to anti-virus engine detection mechanisms which have been presented in chapter 1. It talks about a tool Symantec uses called SymPac which is exactly what project outlined in this report aims to achieve as will be seen in the next chapter. (The fact that Symantec already possesses this tool shifts the aim of this study in a more dynamic approach.) The study defines some traits an unpacker should have. An unpacker should be: (1) Effective: able to restore the original form of the binary before packing; (2) Generic: cover as many packers and binaries as possible; (3) Safe: not leave any undesired side-effects on the system it operates in; (4) Portable: run on multiple operative systems. It talks about the different advantages and disadvantages of the different approaches to detection and unpacking of packed binaries. They propose and implement a solution called Justin (Just-In-Time AV scanning), a purely dynamic approach. This solution implements the same memory-page tracking mechanism as OmniUnpack[MCJ07]. It implements a couple more detection mechanisms such as checking the stack pointer's position (if it is in the same point of the start of the run). It also avoids tracking memory-pages that might contain unpacker stub code. Another technique used is to check if the command line arguments supplied with the packed binary's execution are moved to the stack. The combination of these techniques makes the detection of the end of the unpacking phase more precise.

Malware detection has to be automatic, with virtually no human intervention, the sheer number of malware circulating and the number of entry points to a system leaves no other option. It is obvious that a dynamic approach to unpacking with human intervention to identify the OEP jump allows for an effective unpacking strategy. For detection however, this strategy cannot be applied and automated approaches for OEP jump detection have their limitations. Time and space constraints of the malware detection itself make the use of signature-based detection extremely appealing due to the very good speed and time efficiencies. These considerations and the issue of anti-simulation make static unpacking a promising approach.

3.2.3 Static Unpacking

Static unpacking is the unpacking of an executable without running it. This means a procedure must be applied to the binary in order to unpack it. These procedures can be implemented in a very efficient way by removing all the superfluous operations not related to the unpacking of the binary itself. These superfluous operations can be all the anti-analysis implemented by the packer, the decryption of the unpacking procedure itself, or anything else that is not strictly necessary for the extraction of the unpacked code and data. Because of this, once implemented, static unpackers are usually the fastest way of unpacking a binary. This a big advantage when it comes to real-time detection. An obvious problem with this approach is the fact that a static unpacker only works for the packer it was designed for. Luckily, there are many known packers that are commonly used by malware authors to protect their payload. In the context of the antivirus industry, the project's goal was to develop a generic unpacking framework for runtime packers, used by malicious software. Given the continuous appearance of new packers and variations within families of packers, the produced framework should be implemented in a modular way, allowing for the incremental addition of support for different packers. This allows the framework to be *generic* to some extent. The framework should allow the pure unpacking of a binary as well as its full reconstruction. Focusing on the unpacking itself, a static unpacker can be developed from two different approaches. Either by reversing the unpacking stub and implementing a functionally equivalent procedure, or obtaining the packing procedure and devise a procedure to run it in reverse. Regardless of the approach, the unpacking phase follows a generic process described in section 3.1. The problem was addressed by clearly dividing the two phases: unpacking and rebuilding. The adopted solution was to have a framework that only unpacked the binary, feeding this data to another framework that rebuilds the executable. This approach allows the extraction of unpacked data as needed: if no runnable executable is needed, the data obtained from the first framework is enough. The implementation of the solution is described in detail in chapter 4.

3.3 Conclusions

It is clear that malware detection must address the packer problem. Given the many approaches to unpacking described in this chapter, it is obvious that no one solution is the silver bullet. They all have their advantages and pitfalls. Considering the project's goals, static unpacking was the solution to pursue.

Chapter 4

Implementation

This chapter describes the implementation of the solution for the unpacking framework in detail. It presents the general architecture of the solution and the work process design and application. A description of development process of each unpacker and its integration in the framework is done in depth.

4.1 Framework Design

Taking into account the goals of the project, the developed framework should be implemented in a modular way in order to accommodate the introduction of new packers. With this in mind, and the fact that the purpose of the unpacking framework is to be integrated into the company's products, the use of third-party libraries is undesirable. Given the workings of the company's scanning engine, the framework should be able to, given a file, unpack it into memory and return a pointer to the buffer with the unpacked file. However, to rebuild the executable, the use of TitanEngine[Reva] is employed. To solve this issue, a decision was made to divide the framework into two libraries: one for the unpacking itself and one for rebuilding the executable. These libraries were called *unpacklib* and *unpackrebuildlib*, respectively. With this separation, it is still possible to take advantage of the unpacking in the company's products and use the rebuilding internally for malware analysis. In order to allow for the modularity needed, a class inheritance architecture was adopted. This allowed the methods and data used by all unpackers to be implemented in the base class and re-implemented in derived classes only if needed. This architecture was adopted for both libraries. A data flow was defined between for the global solution.

In order to test the development, a unit testing application was implemented. This application compares the outputs of *unpacklib* and *unpackrebuildlib* to data known to be correct. It also automatically generates this comparison data for the first time an unpacker is implemented.

In figure 4.1 the data flow between the different components involved in the solution can be seen.

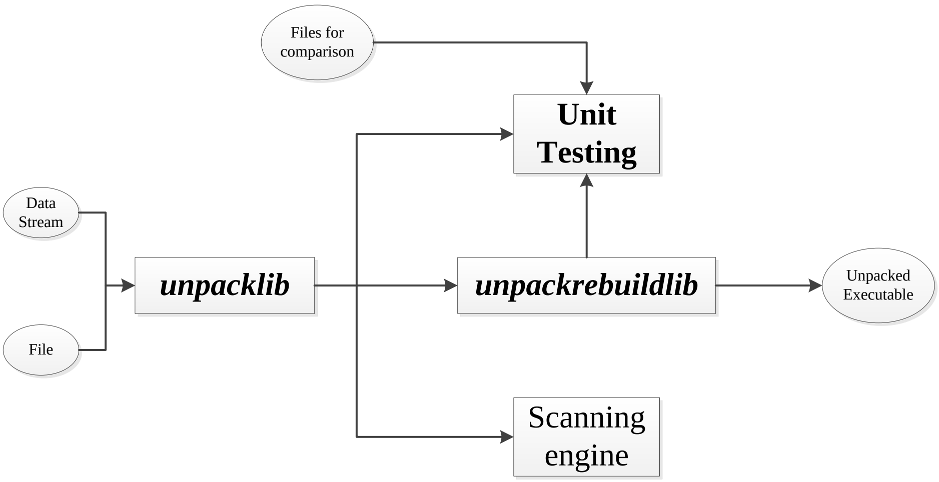


Figure 4.1: General overview of the whole solution

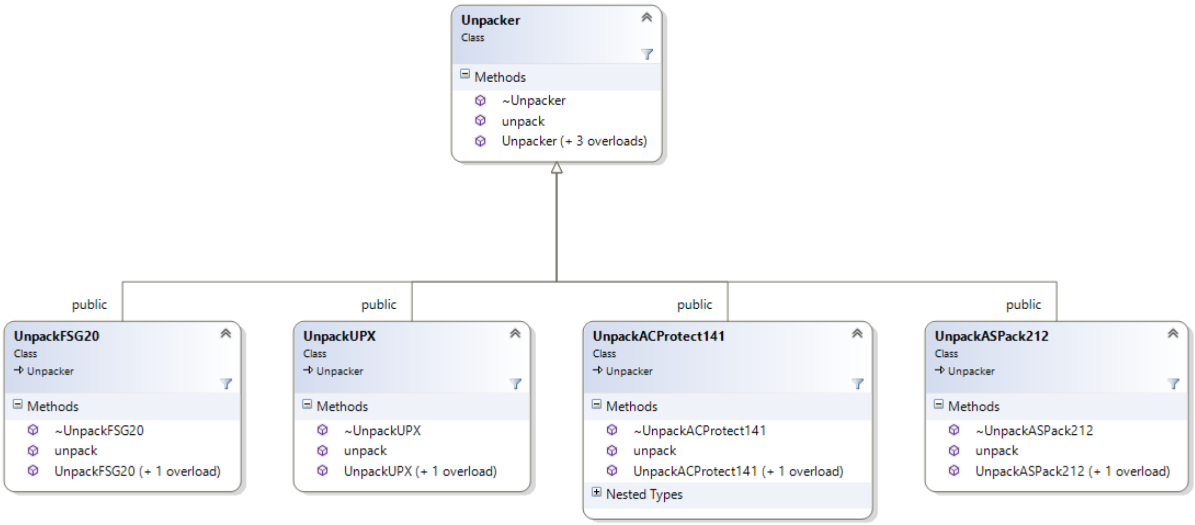


Figure 4.2: Architecture of *unpacklib*

In figure 4.2 the general architecture of *unpacklib* is shown. All the gathered information about the file (such as the OEP and IAT addresses, sections information, etc.) is stored in the base class. This information is essential not only for the rebuilding process but also for analysis. Comparatively, the architecture of *unpackrebuildlib* is shown in figure 4.3. These figures show the implementation of the four unpackers developed in the duration of this project. Subsequent packers can be added by implementing a class derived from the base classes and implementing the

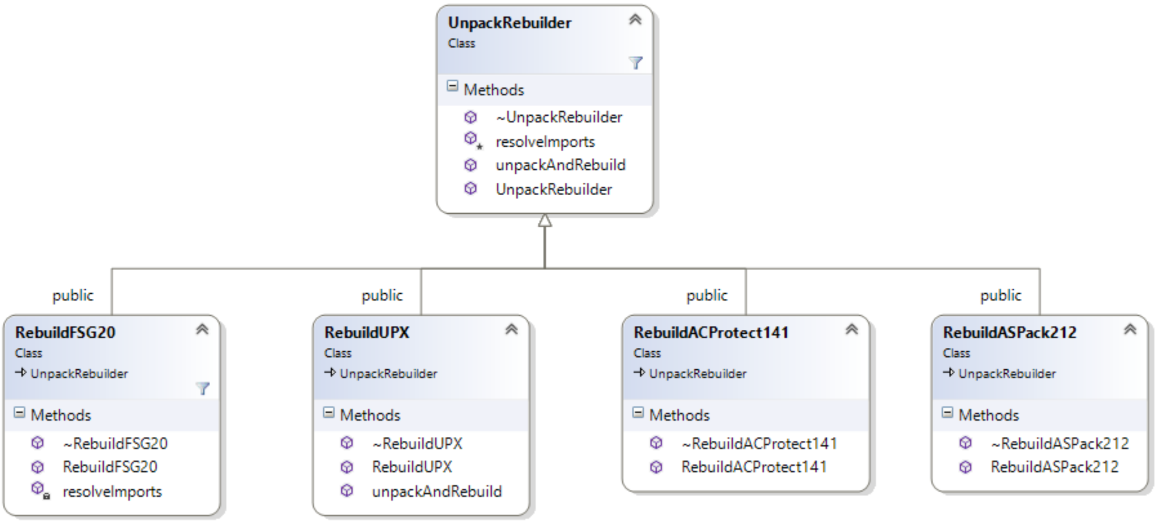


Figure 4.3: Architecture of *unpackrebuildlib*

abstract methods unpack() and unpackAndRebuild().

4.2 Methodology

In order to tackle each of the packers, a systematic approach was defined:

1. Research for anything that can be used, i.e. some unpacking code that can be adapted. If such an open-source code can be found, jump to 7.
2. Research the packer.
3. Obtain the packer and pack a known executable: *notepad.exe*, *calc.exe*, *putty.exe*. If the packer cannot be obtained, use samples in the malware database that are packed with it.
4. Analyse the structure of the file.
5. Disassemble and analyse the unpacking stub.
6. Debug the unpacking stub to get a good understanding of the unpacking procedure and define a way to extract the code.
7. Implement unpacking in *unpacklib*.
8. Implement rebuilding in *unpackrebuildlib*.
9. Test and produce some unit tests.

The use of several different files packed with the same packer allows for the distinction between the constant and variable features of the packer.

4.2.1 Packer Prioritizing

To choose the order by which the packers would be worked on, an analysis was made on the statistical usage of packers in the company's malware database. Each malware sample in the company's malware database has a PEiD signature associated with it. The samples were grouped by PEiD signature for analysis. This analysis produced an ordered list of the most used packers. The top 50 were singled out to be worked on. It must be said that in this list were also entries that were not packers but simply compilers. These were removed. The resulting list is can be seen in Table 4.1.

Table 4.1: Packer Statistical Usage

Usage	Packer	Level	Order
1	Armadillo_v1.71	0	
2	UPX_V2.00-V2.90	1	2 (4.4)
3	UPX_v0.89.6_-_v1.02_/_v1.05_-_v1.22	1	
4	UPX_2.90_[LZMA]	1	
5	Armadillo_v1.xx_-_v2.xx	0	
6	UPX_2.00-3.0X	1	
7	UPX_v0.89.6_-_v1.02_/_v1.05_-v1.24	1	
8	ACProtect_1.41_->_AntiCrack_Software	4	3 (4.5)
9	PolyEnE_0.01+_by_Lennart_Hedlund	-	
10	ASPack_v2.12_	3	4 (4.6)
11	ASProtect_V2.X_DLL_->_Alexey_Solodovnikov	4	
12	ASPack_v2.12	3	
13	ACProtect_1.3x_-_1.4x_DLL_->_Risco_Software	4	
14	Anti007_V1.0-V2.X_->_NsPacK_Private	-	
15	MaskPE_V2.0_->_yzkzero	-	
16	PECompact_V2.X->_Bitsum_Technologies	3	
17	PeCompact_v2.08_->Bitsum_Technologies	3	
18	PeCompact_2.xx_-->_BitSum_Technologies	3	
19	UPX_v0.89.6_-_v1.02_/_v1.05_-_v1.22_DLL	1	
20	PECompact_2.0x_Heuristic_Mode_	3	
21	Upack_v0.39_final_->_Dwing	-	
22	UPX_2.93_-_3.00_[LZMA]	1	
23	Upack_V0.37-V0.39_->_Dwing	-	
24	PECompact_v2.xx	3	
25	FSG_v2.0_->_bart/xt	2	1 (4.3)
26	UPX_v0.89.6_-_v1.02_/_v1.05_-v1.22_(Delphi)	1	
27	Upack_V0.39-V0.399_->_Dwing	-	
28	tElock_0.99_-_1.0_private_->_tE!	-	

After some research, an evaluation was made to the most important packers. A level was assigned to each packer according to the effort required to unpack being 5 extremely complex and 1 very simple, and finally 0, no unpacking needed. The order by which the packer support as added is shown in column "Order".

The fact that some packers appear several times might be due to the fact that they have different versions or different options when packing.

It is clear that Armadillo is the most used packer. However, research showed that this specific version of the packer was simply a wrapper with licensing and anti-analysis features. The payload was not packed, hence the 0. Although the next most used packer was UPX, a decision was made to begin with FSG 2.0. This would allow the author to get familiar with unpacking and reversing. UPX would be worked on after FSG 2.0. Following UPX, unpacking ACProtect 1.41 would be addressed. According to Table 4.1, the next packer to be tackled should be PolyEnE. However, once briefly analysing randomly selected samples from the malware database which supposedly packed with PolyEnE, it was found that they were in fact packed with UPack. Because of this, a decision was made to demote this packer's statistical significance and make the next one ASPack 2.12.

4.3 Unpacking FSG 2.0

In order to get familiar with static unpacking, the unpacking of FSG 2.0 was done. FSG stands for Fast, Small, Good and was developed by "xtreeme team", apparently a Polish team of developers. Its latest version is 2.0 which was the one analysed in this project. This packer presented a good starting point because of its relative simplicity.

4.3.1 Analysis

4.3.1.1 Layout

Work started by packing *notepad.exe* with FSG 2.0. This packed executable was then disassembled and thoroughly analysed.

An important information noticed early on was the fact that the executable did not present the typical layout of a PE file, as shown in Figure 4.4. It only contained two unnamed sections whereas the typical layout is to have three or three named sections: ".text", where the execution code lies; ".data", where data used by the code lies; and ".rsrc", where resources like icons and strings are stored. It was quickly noticed that the unpacking procedure resided just after the real PE header, although the section sizes and size of headers were manipulated to hide it.

It was also observed that the first section was collapsed: it did not take up any space on disk but had space allocated to it in memory. Raw size = 0 and Virtual size > 0. This suggested that this would be the section to which to unpack the data. The second section had some data but a string search yielded little results since the only strings found were about the authors and details of the application, metadata. No strings used in the program itself were found. This obviously suggested that the program was packed.

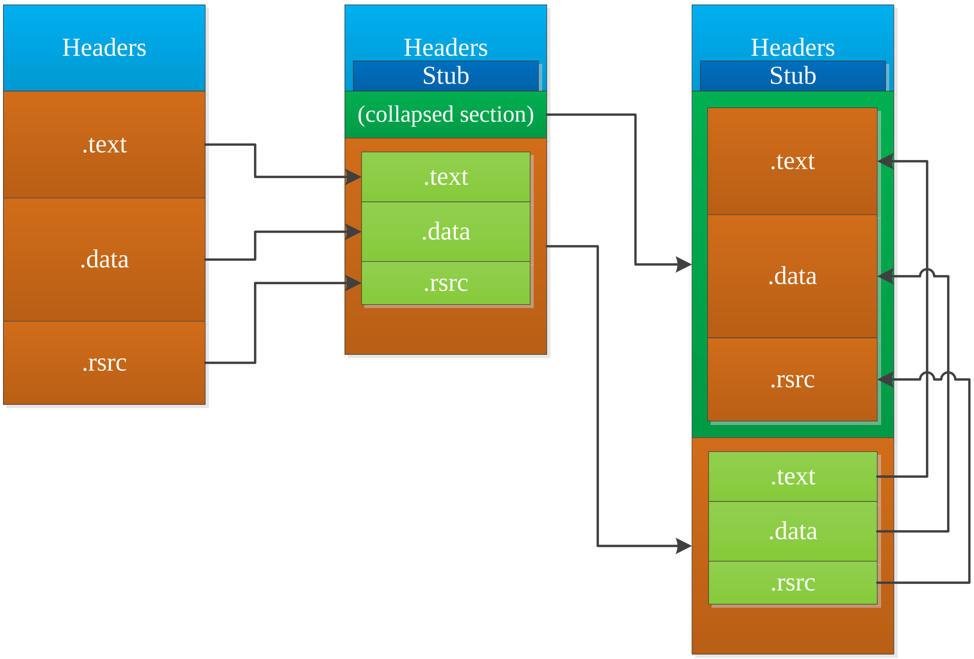


Figure 4.4: Packing and unpacking of FSG 2.0

4.3.1.2 Imports

When analysing the imported DLLs and functions, it is clear that only two functions are imported from Kernel32.dll. These functions were *LoadLibraryA* and *GetProcAddress*. This might seem strange since Microsoft Notepad has a graphical user interface and Gdi32.dll was not imported. However, it is a common technique to hide the program's true imports by importing only *LoadLibraryA* and *GetProcAddress* since by calling these functions it is possible to import all the libraries and functions that are needed.

4.3.1.3 Unpacking

After careful analysis of the unpacking procedure, two different phases were identified: one to unpack and one to resolve the imports.

Unpacking data

Once the unpacking code had been correctly identified, it was necessary to determine if it was generic, that is, if it was the same for every executable packed with FSG 2.0. This was done by packing *putty.exe* and analysing it. It was concluded that it was indeed generic and that the start state values from which it began were the only things that varied from one executable to another. These values were stored in the packed data section. The address at which these values are stored could be extracted from the first instruction (file offset 154h+2) of the unpacking code, which loaded the values into memory. This address, however, was a VA (Virtual Address), as was the OEP address, and there was no relocation information. This meant that FSG 2.0 could not run on systems with ASLR enabled. In Microsoft Windows ASLR came enabled by default only from Windows Vista SP1 onwards.

Resolving imports

Once all the data finished unpacking, it is clear that all the sections present on the original unpacked file have been merged and are now present in the first section. Analysing this section, it is clear that the IAT is now clearly visible and the imports by name can be seen, even though it is not according to PE COFF specification. The part of the unpacking stub that deals with the imports mimics the PE loader in the sense that it loads the libraries and finds the offset of the imported function and places that offset in the table for the application to use. Since the imports are visible after unpacking, they can be extracted to build a complete import table.

OEP jump

At the end of the import resolution, the unpacking stub hands over the control to the original application. This jump is made to an offset that is also stored in the values array in the packed section of the file and it is therefore possible to extract.

4.3.2 Implementation

To unpack statically means the executable should not be run but another process should unpack it by modifying its contents. To this end, an application was developed to do just that.

4.3.2.1 Unpacking

First, the executable file was loaded into a buffer in memory. Once there, the header was parsed using a library the company possesses. The unpacking code, extracted from the disassembled unpacking stub, was translated from x86 assembly into C code. The start state was built by parsing the values in the file at the offsets that have been identified in analysis. The unpacking code was executed and the data was unpacked into another buffer in memory. The next task would be to produce an executable file of the unpacked code.

4.3.2.2 Rebuilding the executable

In order to produce an executable file, the buffer with the unpacked data would not suffice. A header should be added that would enable the Windows loader to load the file and execute it. The import table should also be fixed. Another consideration is that some of the resources for proper loading of the file still remain in the packed data section, such as the author and icon information. To aid in this task, another library was used: TitanEngine[Reva].

Using this library, the first section of the file was inflated to its size in memory (since it was collapsed) and the unpacked data copied into it. A parser was developed for the IAT, and the functions provided by TitanEngine were used to build a valid import directory, according to PE COFF specifications. A new section was added at the end of the file with the new import directory. Through this approach, all the libraries and functions imported by the import resolution phase of the unpacking stub were present in the new import section.

Also with the help of this library, the header was fixed to reflect the new sectional layout, the entry point was set to the OEP, whose address had been parsed when unpacking and the IAT address was pointed to the new correct IAT. This produced an unpacked executable file. Although this file was unpacked and executable, it was not identical to the original file before packing, it still contained the packed section and the section layout information from the original file had been completely lost. This, however, is not a problem since the objective was to unpack and produce an executable file for analysis.

4.4 Unpacking UPX

The Ultimate Packer for eXecutables, UPX is a very popular open-source packer. It has been around since 1996. It was developed and maintained by Markus Oberhumer, László Molnár and John F. Reiser. The fact that it supports unpacking out of the box made the unpacking of UPX extremely simplified. The fact is, UPX is not aimed at protecting code from reverse engineering or analysis: it is aimed at compressing executables. This is the reason it supports unpacking out of the box and is open-source. If it was aimed at protecting against reverse engineering, this would be counter-productive.

Adopting a pragmatic approach, no analysis of the packer was done. Work could start immediately with integration of the available source-code. Starting from the source code of UPX, work began in integrating it into *unpacklib*. All the code concerning packing was removed. A wrapping interface was created to integrate the code into *unpacklib*.

The fact that UPX supports many different executable types, from PE's to ELF's, made it possible to also integrate some unpacking capability for these different executable types and architectures. The integration of the UPX unpacking code meant that virtually all the versions of UPX could now be unpacked.

4.5 Unpacking ACProtect 1.41

Anti Crack Software Protector, ACProtect is a packer designed with protection in mind. Unlike the packers previously dealt with, ACProtect 1.41 implements some anti-debugging and encryption which made the unpacking a lot more challenging. The version analysed, 1.41, is quite old, from around 2005. At the time of writing this document, the developers' website was no longer operational. Information was found to be scarce and scattered. However, it was possible to determine that ACProtect 1.41 is a packer very similar to ASProtect, and that the developers were

Chinese. The protections implemented in ACProtect seem to be unceremoniously copied from ASProtect (see section 4.6). ACProtect 1.41 has a licensing mechanism that allows developers to issue registration keys for their software protected by ACProtect. It features public key encryption algorithms for generation of mentioned registration keys. It can perform code replacement, where some code of the protected application is replaced with calls to functions in the ACProtect code. It also uses a metamorphic engine and several encryption layers, varying every time a file is packed. It features many anti-debugging, anti-disassembly and anti-emulation techniques.

4.5.1 Analysis

4.5.1.1 Layout

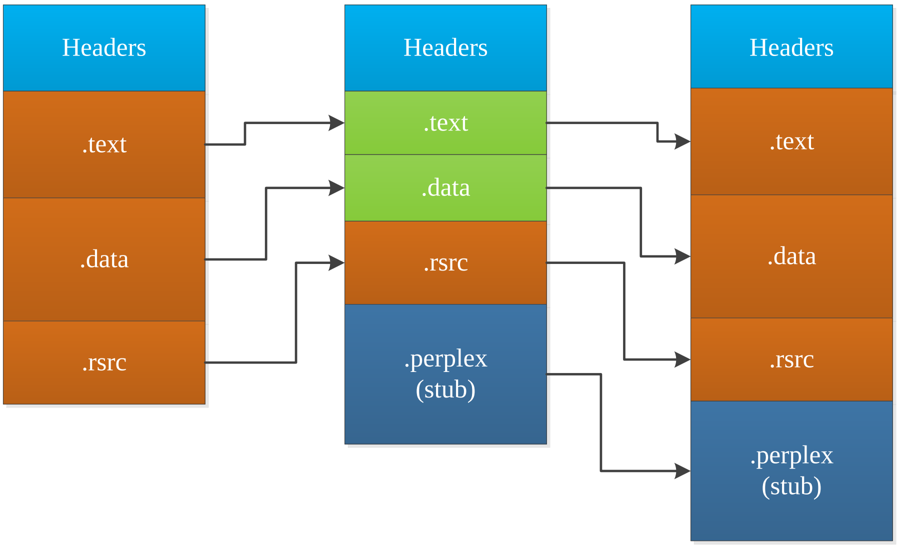


Figure 4.5: Packing and unpacking of ACProtect 1.41

Following the general procedure, work started by obtaining a copy of the packer and packing *notepad.exe*. Comparing the packed and unpacked notepad executables, a notable difference can be seen: the packed executable has one extra section, the last one, named ".perplex". This section probably holds the unpacking stub, since the address of entry point is located within it. Other differences are in the raw sizes of the other original sections that in some cases have shrank. Packing of other executables with this packer did not always result in a smaller binary size due to the appending of an extra section of considerable size in the end.

4.5.1.2 Imports

Once again, the imports give us very little information about the packed executable. In this case there are more explicit imports than the typical *LoadLibraryA* and *GetProcAdddress*. Some research on the packer and the packing tool reveals this packer has the ability to look for an enemy process and kill it, and, due to the licensing mechanism, it is able to display a message box when a trial version is used. This explains the *ExitProcess* import from Kernel32.dll and the *MessageBoxA* from User32.dll.

4.5.1.3 Unpacking

After long and careful analysis of the unpacking, several stages were identified.

Decrypt unpacker

The last section, containing the unpacking stub, proved to be encrypted. This made static analysis practically impossible. A dynamic analysis proved that not only did the stub decrypt itself several times, it also mutated throughout execution. Because of this, it was much harder to identify the unpacking code itself, within all the modifying and stub decrypting code. Furthermore, the decryption of the stub code varied from packed file to packed file but only in some instructions and hardcoded values. This will be discussed in detail below. After some persistence, the unpacking code for the sections was identified. This code proved to be the same in different packed files.

Unpack data

The unpacking is done section by section most of the times, although sometimes it can unpack two contiguous sections at once. However, not all sections are packed and not all the data in a section might be packed. The information about the packed sections, the beginning and size of the packed code is stored in an array of DWORDs in the stub section. This array however, although located at the same offset within the .perplex section in every packed file (at file offset E836h relative to the beginning of the section) is encrypted. The decryption of this array is done at runtime like the rest of the section, may it be code or data, and follows the same general procedure shown in Listing 4.1. It is clear from the amount of variables and variable operations on the DWORDs that the decryption is not straightforward. With the information of the packed sections, the unpacking stub proceeds to unpack each of the sections by first copying the whole section into a dynamically allocated buffer and then unpacking it directly into the section, overwriting it.

Check for debugger and licensing

Once all the sections have been unpacked, the unpacker proceeds to do an array of verifications. These include detection of debugger or analysis through various methods, verifying the file has not been modified by reading it and generating a hash and comparing it to the stored hash, looking for enemy processes and killing them, etc. Some of the debugger detections were:

- IsDebuggerPresent API from Kernel32.dll

- int 3 - instruction causes an interrupt to see if it is intercepted by a debugger
- int 1 - trap to kernel debugger
- Searching all the running processes for known debuggers and analysis tools (ollydbg, etc)

The packer also has some anti-disassembly features. Return pointer abuse and conditional jumps back to back are extensively used (see section 2.4.1). In order to bypass all the debugger detections the IDA Stealth[New] plugin was used and all exceptions were passed to the debugee. If debugging or analysis is detected, the execution will end, either by intentionally crashing or simply terminating. Assuming no debugging is detected, it then proceeds to check if the file is licensed or in trial, if not licensed, it displays a message box saying it is a trial version.

```
...
mov          ecx, some_value           ;not really important for the purpose
                                         ;of decrypting the array
mov          edi, seed_value           ;unknown seed value, different for every
                                         ;packed file, calculated throughout
                                         ;the previous decrypting cycles
loop_start:
mov          eax, dword_from_array[i]  ;get the DWORD from the array
<op1>        eax, edi                  ;the operation varies but is always from
                                         ;the set {sub, xor, add}
<ror/rol>    eax, rot_value            ;operation also varies, but not important
                                         ;since rol and ror are complementary.
                                         ;The value however is unknown and is
                                         ;different in every packed file. It
                                         ;is also calculated throughout the
                                         ;previous decrypting cycles.
<op2>        eax, dword_from_array[i+1] ;gets the next DWORD from the array and
                                         ;performs once again an unknown operation
                                         ;on it from the set {sub, xor, add}
mov          dword_from_array[i], eax  ;puts the decrypted DWORD back in the array
<op3>        edi, step_difference      ;operates on the value that is used in the
                                         ;first operation, again with an unknown
                                         ;operation {sub, xor, add}
loop loop_start
...
```

Listing 4.1: General procedure for decryption of stub

Resolving imports

In order to resolve the imports the procedure uses another array of DWORDs (at file offset E906h, relative to the .perplex section start) that contains the RVA of the original import directory. This is, of course, also encrypted and is decrypted with the generic code seen before. Once it has the decrypted RVA of the import directory it proceeds to decode the API strings that are still encoded, even though the section is already unpacked. This decoding is however extremely simple and proved to be the same in every packed file. It is done byte by byte on each string by doing a rol of 3. The operations done on each string can be seen in Listing 4.2.

```
...
decrypt_byte:
mov    al, string[i]    ;get byte
test   al, al           ;see if the string has ended (byte is zero, since it is a
    null terminated string)
jz     end_loop         ;when string end, break loop
rol    al, 3            ;operate on the byte
mov    string[i], al    ;put the byte back
jmp    decrypt_byte     ;continue loop
end_loop:
...
```

Listing 4.2: General procedure for string decoding

After all API strings are decoded, it uses *GetModuleHandleA* API to get the handles of the already loaded libraries and loads the rest using *LoadLibraryA*, and links the APIs using *GetProcAddress*: a typical approach. However, once each DLL is loaded, it overwrites the string with zeros. The same is done with the APIs.

OEP jump

Once getting past the anti-debugging tricks, some more decryption is done and the OEP jump becomes obvious, with a simple jmp instruction to the correct offset. This offset was stored, unencrypted, at file offset 1714Bh, relative to the .perplex section.

Code replacement

If this option is checked when packing an executable, this packer replaces certain instructions with calls to functions in the ".perplex" section that implement the same functionality, although highly obfuscated. This of course means that the unpacked code is inevitably different than the original code before packing.

4.5.2 Implementation

4.5.2.1 Unpacking

In order to statically unpack executables packed with ACProtect 1.41, it was necessary to address the most obvious problem: the encrypted values. There was, however, a critical detail regarding the layout of the arrays where the values are stored that suggested the problem could be solved: the 10 or so DWORDs before the array were supposed to be zero after decrypting and they were decrypted in the same loop as the values needed. As seen in Listing 4.1, to properly decrypt a DWORD, one must know:

- the DWORD to be decrypted
- the next DWORD
- y = seed_value
- z = rot_value
- a = step_difference
- α = op1
- β = op2
- γ = op3

Since the DWORD and the next DWORD are obtained directly from the array, it is only necessary to find out the other values and operations. It is not necessary to know if the rotation is ror or rol. It can be set to ror and just get a different value for the rot_value but still preserve the functionality.

Let x[0] be the first DWORD in the array that holds some data to be decrypted (i.e. not 0). With this in mind, and knowing that the DWORDs before the beginning of the array are supposed to be 0 after decryption, it is possible to extract the following three equations:

$$\beta(ror(\alpha(x_{-3}, 0\gamma(y,a)), z), x_{-2}) = 0 \tag{4.1}$$

$$\beta(ror(\alpha(x_{-2}, 1\gamma(y,a)), z), x_{-1}) = 0 \tag{4.2}$$

$$\beta(ror(\alpha(x_{-1}, 2\gamma(y,a)), z), x_{0}) = 0 \tag{4.3}$$

From these three equations, the following two equations were extracted:

$$y = \alpha(x_{-3}, rol(\beta^{-1}(0, x_{-2}), z)) \tag{4.4}$$

$$a = \gamma(y, \alpha(x_{-2}, rol(\beta^{-1}(0, x_{-1}), z))) \tag{4.5}$$

Since from these two equations the variables that have more different possible values (DWORDs) are obtained, it is reasonable to find the others by brute-force: each operator only has 3 possibilities and **z** only has 31. With this hybrid "informed-brute-force" approach it was possible to find all the needed values and operators within a very reasonable time-frame (few ms). Once all the values and operators are defined, it is possible to decrypt the rest of the array and get the information needed for unpacking.

Because this packer sometimes unpacks several contiguous sections at once, simply loading the file into memory would not suffice. To address this, a simple PE loader was implemented to mimic the memory allocation of the Windows PE loader. With the executable "loaded" into memory, unpacking could be done. The unpacking code was converted to C and thoroughly tested in both 32 and 64 bit architectures with the values decrypted using this approach. The resulting unpacked sections were identical to the ones unpacked at runtime.

4.5.2.2 Rebuilding the executable

Once all the sections were unpacked, one could proceed to fix the imports. For this, as previously, TitanEngine was used. As mentioned above, the strings with the libraries and API functions are still encoded, so decoding must be done on the strings before. For the decoding of the strings, the string decoding procedure described in "Resolving imports" was used. It was noticed that the import directory was according to PE COFF specifications, and a parser was developed for it. With this, the imports were fixed and the executable rebuilt.

There remained, however, a problem. When packing, if the user chose the code replacement option, the resulting code, even after unpacking is inevitably different than the original one. As far as signature identification goes, this might not be a very big problem since the replaced instructions are sparsely spread, with many large sequences of untouched code. Execution, however, remained an issue. This is because the code with which the instructions were replaced are calls to functions in the stub (.perplex) section. Even if the stub section is present, it is not present in its final, decrypted state, since no static unpacking of this section was done. A decision was made not to pursue unpacking of the .perplex section since the original code was already unpacked and as far as signature identification goes this was enough. Also, all the imports were now visible as well.

4.6 Unpacking ASPack 2.12

ASPack is essentially just a compressor, it has no protection or obfuscation asides from omitting a few API imports in the packed file. It is the simplest packer/protector in a suite that includes ASProtect and the extra protections are therefore implemented in the latter. The suite is developed by Alexey Solodovnikov, and the products are commercially available to legitimate costumers who want to protect their software.

Although this packer had no anti-debugging tricks, it proved a challenge because it is object oriented, written in C++. This is a problem because when translating the code to C, all the pointers must be taken into account. Although this packer only supports the IA-32 architecture, the developed unpacker should also work in 64 bit architecture.

4.6.1 Analysis

4.6.1.1 Layout

Once again, the packer was obtained and *notepad.exe* was packed. A comparison between the section layout of the packed and unpacked file shows that the packed file keeps all the original sections and adds two extra sections at the end. These sections are named ".aspack" and ".adata"

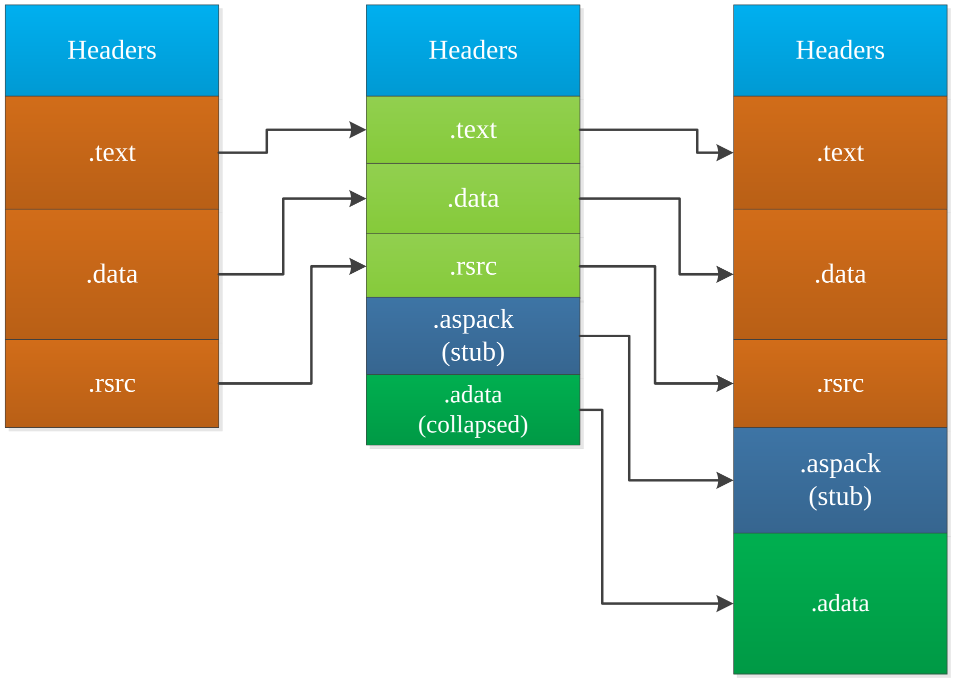


Figure 4.6: Packing and unpacking of ASPack 2.12

and appear in that order. The last section, ".adata" is completely collapsed and has raw size of 0. This suggests the unpacking code resides in the ".aspack" section. It was also noticed that all the original sections had different raw sizes, meaning they have all been packed or touched by the packer in some way.

4.6.1.2 Imports

The libraries imported by the packed file are the same as the original file. However, many API imports are missing. This suggests they are linked at runtime. The import of *LoadLibraryA* and *GetProcAddress* also support this.

4.6.1.3 Unpacking

Unpack data

Unpacking is done section by section. The information about the address of the packed code and its size in each section is stored in an array at offset 57Ch, relative to the beginning of the ".aspack" section, and is not encrypted. For each section, an instance of a class is created and a buffer is allocated to which the code is copied and subsequently unpacked. The unpacking itself is done with the aid of 9 functions, belonging to either one or two classes. These classes have member data, some of the data are pointers to arrays or other structure types. The unpacking is done also with two seed number arrays, one present in the ".aspack" section and another which is dynamically allocated and calculated in runtime.

Resolving imports

The unpacked import directory is according to PE COFF specification. The RVA of the original import directory is located at offset 279h relative to the beginning of the ".aspack" section. The resolving of imports is pretty straightforward. The unpacker calls *GetModuleHandle* with every DLL to see if it is already loaded, if not, loads it. It links all the APIs using *GetProcAddress*.

OEP jump

The OEP is stored at offset 39Bh, relative to the beginning of the ".aspack" section. It is not encrypted. A simple jmp instruction transfers the execution to it.

4.6.2 Implementation

4.6.2.1 Unpacking

After identifying and understanding all the functions used in unpacking, the translation into C code began. In the member structures of this class, all pointers had to be singled out from the rest of the data. This was done by carefully going through the code and seeing where an access was attempted at an address that was stored in one of these structures. These variables were then converted into offsets, relative to a beginning. This approach was chosen instead of changing them to pointers because in this way one could keep the structure intact and just add the pointer to which the offset is relative to when it would be used. Otherwise, let's say an array of DWORDs has data and pointers, when the pointers become QWORDs, the array is broken from that point on. All the pointers were corrected and the information about the packed sections was parsed. With this, the sections could be sequentially unpacked. Testing proved the static unpacked sections were identical to the runtime unpacked ones, both in 64 and 32 bits.

4.6.2.2 Rebuilding an executable

After unpacking all the sections, TitanEngine was used to rebuild the imports. Parsing the import table presented no problem as the layout was according to PE COFF specification and the parser developed previously proved effective. The imports were rebuilt successfully without issue and executable files were produced.

4.7 Conclusions

It is clear from the different packers analysed and integrated in the unpacking framework that packers can differ greatly in their approaches. Each of the packers presented a different challenge and the unpacking strategy had to be adapted.

Chapter 5

Tests and Analysis

In this chapter, the integration of the framework in the company's malware scanner is described. The test design is explained and the results are analysed. Keeping in mind the focus of this project, the data is analysed in the perspective of packers and unpacking.

5.1 Test Design and Implementation

In order to test the developed unpackers in real-world conditions, the framework had to be integrated in the company's malware scanner. This had already been considered in the design phase, making integration easier.

It is clear from chapter 4 that in the developed framework the issue of packer detection and identification is not addressed. However, packer identification through PEiD signatures had already been performed on the malware database, and that provided a good packer identification to work with. Given the implemented unpackers, four PEiD signatures were selected for testing.

- `FSG_v2.0_->_bart/xt`
- `UPX_V2.00-V2.90_->_Markus_Oberhumer_&_Laszlo_Molnar_&_John_Reiser`
- `ACProtect_1.41_->_AntiCrack_Software`
- `ASPack_v2.12_`

Although the malware in the database had packer identification, the available version of the scanner was not prepared to use it. To solve this, the scanner was modified and four different scanners were built, each with support for one of the packers. In the scanning process, a step was added in order to try to unpack a PE32 file. If unpacking was successful, scanning would be done on the unpacked sections instead of the packed ones. This also allowed scanning not to be done on the unpacking stub section. Although a buffer with the unpacked data was already available

from *unpacklib*, some specific issues needed to be addressed in the integration with the scanner. One of these issues is the fact that there might exist more than one section present in the unpacked buffer. This might not seem an issue, but the reality is that sections marked as non-executable are treated differently than sections marked as executable. This meant that the information about the sections contained in the buffer should also be transmitted to the scanner. This information included section offsets and sizes and whether it is executable or not.
It was assumed that the execution simulation done within the scanner was only for dynamic unpacking. This meant that if the unpacking was successful, there was no need to run this simulation. This assumption, although reasonable to make, is not absolutely true: in some cases, the features extracted in the simulation are important in detection.

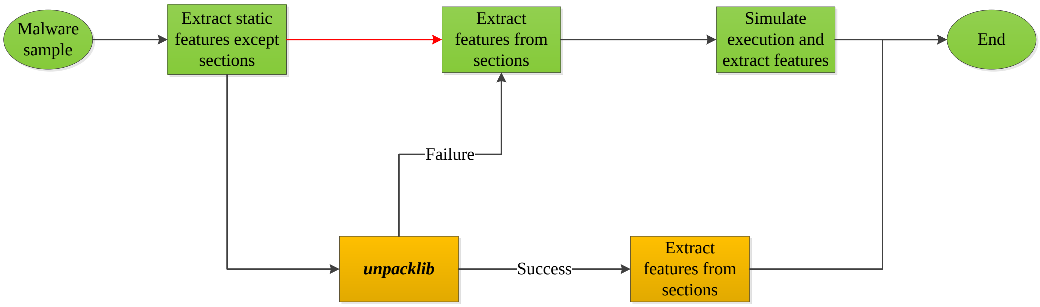

Malware sample
Extract static features except sections
Extract features from sections
Simulate execution and extract features
End
Failure
unpacklib
Success
Extract features from sections

Figure 5.1: Modified scanning process. In green is the original process.

In Figure 5.1 the change made to the scanning process can be seen. In green is the original scanning process, in orange are the new steps added (the connection in red was removed). Having the four different scanners, each with support for one of the packers, a testing environment was set up. The approach adopted was to scan the malware samples identified with the PEiD signature corresponding to the supported packer of each scanner. In order to get a basis for comparison, these samples were also scanned with the unmodified scanner. It should be said that the scanning performed on these samples, with all the scanner versions, was extremely thorough. For each of the samples, features were extracted from every scanning step and detection for signatures was performed on every one of these features. This is a different mode of operation than normal. The common user only wants to know if a file is infected or not. This means that at the first signature match on a feature, the scan can terminate. For testing purposes, the scanning was performed forcibly through all the scanning stages and different feature extractions. This approach was chosen to make sure scanning would reach the unpacking step. With this approach, naturally, more than one signature match can be found for one particular sample.

From all the scanning performed, data about time, resources, detected signatures and success of the unpacking step was collected. The resulting data is presented and analysed in the next section.

5.2 Global Results and Analysis

In Table 5.1, the results from the tests performed on each of the unpackers can be seen. As previously stated, for each of the unpackers, a set of malware samples identified by a PEiD signature corresponding to the packer was selected. These sample sets naturally varied in number but were all well above the hundreds of thousands, making the tests statistically significant.

In data presented in Table 5.1, the sample count is the number of malware samples with which the unpacker was tested. The "Unpack Success Rate" represents the percentage of the samples tested that were successfully unpacked by each of the implemented unpackers. All values asides from the sample count and the unpack success rate only consider the successfully unpacked samples. "Average Time" and "Average Memory" are the time taken and memory use of the new scanning process, relative to the old, standard scanning process. The "Old Detection Rate" and "New Detection Rate" are the percentage of samples that get detected as malware (i.e. at least one signature match) in the standard scanning process and the new process with static unpacking, respectively. A graphical perspective of this data can be seen in Figures 5.2, 5.3 and 5.4.

Table 5.1: Global Results

	FSG 2.0	**UPX**	**ACProtect 1.41**	**ASPack 2.12**
Sample Count	258510	2466228	1169790	783149
Unpack Success Rate	86%	8%	0,015%	67%
Average Time (relative to old scanning process)	48%	57%	59%	76%
Average Memory (relative to old scanning process)	67%	60%	78%	109%
Old Detection Rate	96%	99,6%	99,99%	94%
New Detection Rate	87%	88%	43%	84%

The successful unpack rate of the implemented unpackers varied widely. There might be a couple of reasons for the unpacker not being able to unpack. The first reason has to do with false identification. The PEiD signatures used to identify files packed with the respective packers is vulnerable to false-positives, as is any other signature for that matter. This means that the sample set selected was not composed solely by samples packed with the intended packer. Other files were also present, packed by other packers or simply not packed. Another reason might be that some samples in these sample sets were indeed packed with the intended packer but with slight changes, and the unpacking implementation is not tolerant enough to these slight variations. Another reason might be that the unpacker crashed during unpacking. This might mean that the sample was tailored in a specific way to try to crash analysis tools like the one developed. It might still be packed by the intended packer but have, for example, a deliberately modified header that crashes the tool when it tries to load the file in memory. These crashes, however, only happened

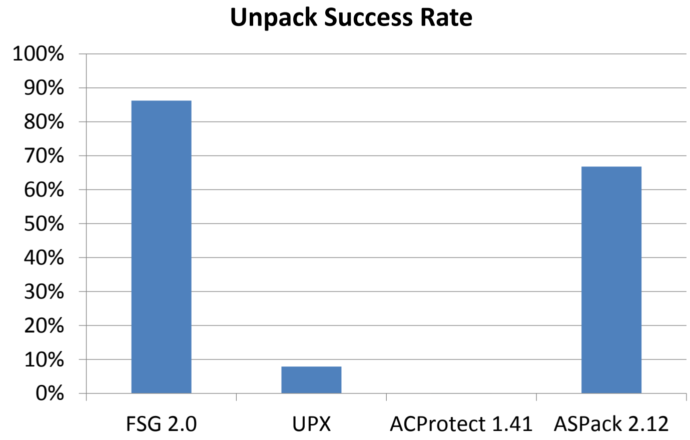


Figure 5.2: Global success rates

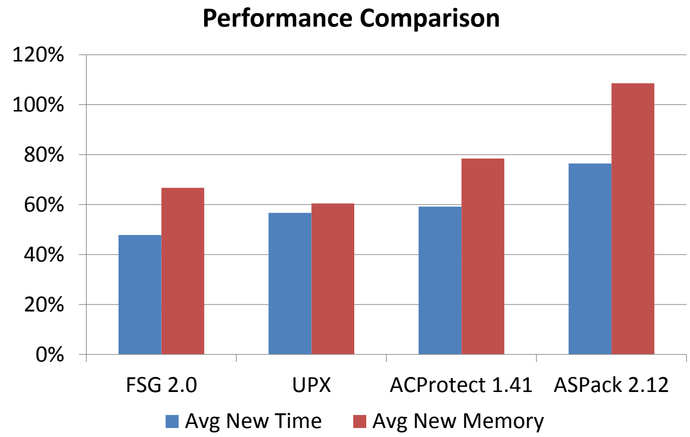


Figure 5.3: Global performance

in the ACProtect and in the ASPack unpackers and the crash rate was around 0,02% in both cases, making them statistically insignificant.

The averages of time and memory are presented relative to the standard scan. Naturally, for these values, only the samples in which static unpacking was successful are compared. In these cases, a performance improvement was to be expected since the dynamic unpacking is not performed when the static unpacking is successful, reinforcing the performance advantage of the static approach to unpacking.

The old and new sample detection rates refer to the percentage of samples, from the subset of

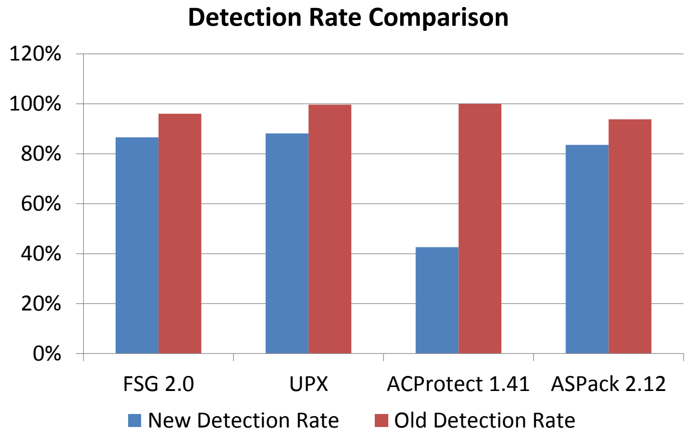


Figure 5.4: Global detection effectiveness

samples that were successfully unpacked, that had at least one detection (i.e. were detected as malware in the scan). It can be seen that even in the standard, unmodified, scan there were some samples that did not get identified as malware. In each of the cases, the rate of detection went down. Given the fundamental changes done to the scanning process, changes in the detection rate were expected. The loss of signature matches relative to the standard scan might be due to a number of things. One reason is that the signatures might have been matched on the unpacking stub or packed section on the standard scan, and on the modified scan the packed sections and stub are ignored and only unpacked sections are scanned. Another reason might be that the detection might have been made in the execution simulation on the standard scan, where the behaviour is analysed. Since this execution simulation is not performed when static unpacking is successful, those detections are lost. Another consideration that must be taken into account is that when looking at simulation as an unpacking technique, executables packed several times (several layers of packers on a given file) can be successfully unpacked. With the current implementation of the unpacking framework this is not possible, and therefore some detections can be lost in these situations. Of course it is highly desirable that a very large portion of the samples be detected by static signature matching alone because of performance. There were also cases in which the number of signature matches increased for a sample. An increase in signature matches is almost certainly because of the scan being performed on the unpacked section instead of the packed one. This means that signatures that were in the database but were not matched to this file due to that fact that it was packed can now be matched, possibly identifying the sample as very similar to another that is already present in the database in its unpacked state.

The availability of unpacked data early on in the executable analysis allows for signatures to be generated over this data and improve the detection rate and performance in the future.

5.3 Individual Packer Analysis

In the next subsections, the results gathered for each of the unpackers will be presented with the aid of some tables and charts. The tables and charts presented for each packer were standardized. Table 5.2 shows the figures and tables and their corresponding meanings. The "Item" value refers to the item number from the enumeration below that explains the content. For each of the packers a Figure or Table is referenced to a description of its content.

Table 5.2: Figure meanings

Item no.	**FSG 2.0**	**UPX**	**ACProtect 1.41**	**ASPack 2.12**
1	Table 5.3	Table 5.4	Table 5.5	Table 5.6
2	Figure 5.5	Figure 5.11	Figure 5.17	Figure 5.23
3	Figure 5.6	Figure 5.12	Figure 5.18	Figure 5.24
4	Figure 5.7	Figure 5.13	Figure 5.19	Figure 5.25
5	Figure 5.9	Figure 5.15	Figure 5.21	Figure 5.27
6	Figure 5.8	Figure 5.14	Figure 5.20	Figure 5.26
7	Figure 5.10	Figure 5.16	Figure 5.22	Figure 5.28

1. This table presents the main measured results. All values asides from the sample count and the unpack success rate only consider the successfully unpacked samples. Asides from the measured results that were already explained for Table 5.1 in section 5.2, two more values are shown: "Total Detection Count Rate" and "Same Detection Rate".

    - Same Detection Rate - An average of the percentage of signatures that are present on both of the following sets: for a set of signatures matched for a sample that was successfully unpacked using the modified scanner, there is another set of signatures matched using the standard scanning procedure, for the same sample.
    - Total Detection Count Rate - The total number of signatures matched for all samples that were successfully unpacked using the modified scanner relative to the total number of signatures matched using the standard scanning procedure, for the same samples.

2. This chart presents the success rate of the unpacker.

3. This chart shows the impact of the new scanning process on signature matching. This chart compares the signature matches, in number, of the samples where static unpacking was successful, with the signature matches of the same sample scanned with the standard scanning process. That is, if a successfully unpacked sample with the new scanning method had more or less signature matches compared with the standard scan.

4. This chart presents the comparison of the average speed of the two different scans. It presents this comparison with two data sets: "Successfully Unpacked" and "Global". As

the name suggests, "Global" is the whole sample set selected by the PEiD signature. "Successfully Unpacked" is the subset of samples from "Global" that were static unpacking was successful. The fact the simulation still runs when unpacking is unsuccessful explains the slight increase in time when considering all the samples, compared to only the ones successfully unpacked.

5. This chart shows how many scans were faster or slower in the modified scan compared with the standard scan, from the successfully unpacked samples.

6. This chart presents a comparison in terms of memory usage. There are two sample sets, same as described before in item 4, this time for peak memory usage

7. This chart shows how many scans used more or less memory in the modified scan compared with the standard scan, from the successfully unpacked samples.

5.3.1 FSG 2.0 Analysis

Table 5.3: FSG 2.0 Global Results

FSG 2.0 Results	
Sample Count	258510
Unpack Success rate	86%
Average Time (relative to old scanning process)	48%
Average Memory (relative to old scanning process)	67%
Old Detection Rate	96%
New Detection Rate	87%
Total Detection Count Rate (relative to old scanning process)	124%
Same Detection Rate	40%

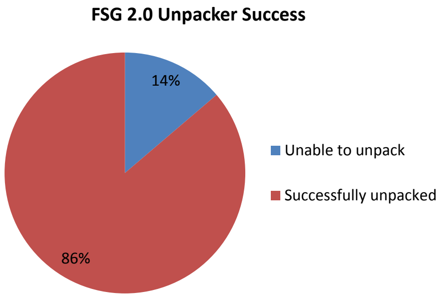


Figure 5.5: Success rate of the FSG 2.0 unpacker

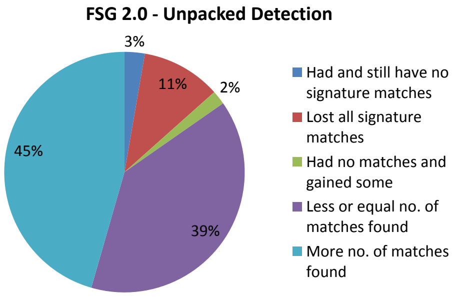


Figure 5.6: Analysis of the FSG 2.0 unpacker impact in detection

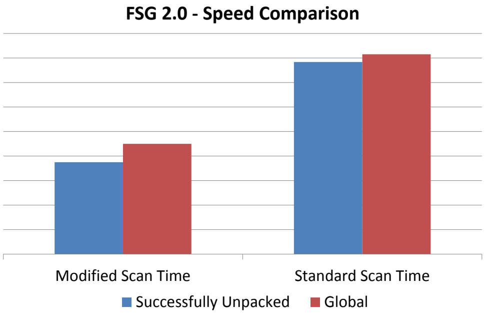


Figure 5.7: Speed comparison between standard and FSG 2.0 unpacking modified scanner

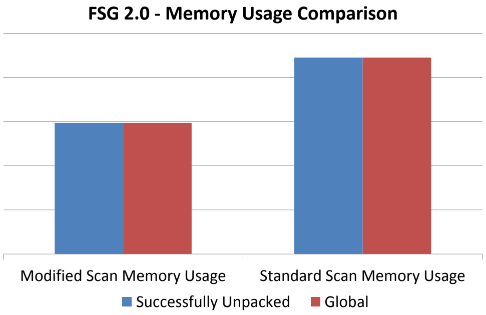


Figure 5.8: Memory usage comparison between standard and FSG 2.0 unpacking modified scanner

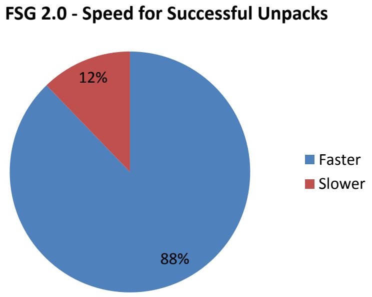


Figure 5.9: For the successfully unpacked samples, speed relative to standard scan

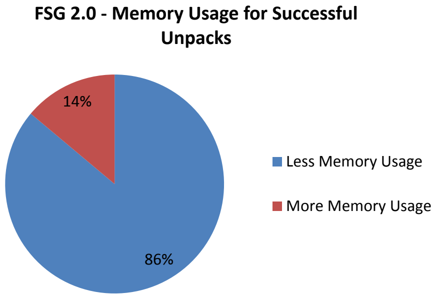


Figure 5.10: For the successfully unpacked samples, memory usage relative to standard scan

As discussed before in section 5.2, the main reasons for unsuccessful unpacking is to do with incorrect packer identification. In the case of FSG 2.0, several samples that could not be unpacked were selected at random and manually analysed. They revealed to be packed by FSG 1.33, or not packed at all, confirming the hypothesis.

The detection rate dropped from 96% to 87%. Meaning 13% of the samples are not detected as malware using the new scanning process. Although this is true, in total, there were more signature matches than before, 24% more, most certainly detected in the unpacked data. The same signature rate of 40% (see section 5.3) is a surprisingly low value. This low value might be attributed to the fact that signatures found in the unpacked section were possibly not previously matched. Comparatively, signatures matched on the packed section were not matched in the new scanning process.

5.3.2 UPX Analysis

Table 5.4: UPX Global Results

UPX Results	
Sample Count	2466228
Unpack Success rate	8%
Average Time (relative to old scanning process)	57%
Average Memory (relative to old scanning process)	60%
Old Detection Rate	99,6%
New Detection Rate	88%
Total Detection Count Rate (relative to old scanning process)	62%
Same Detection Rate	91%

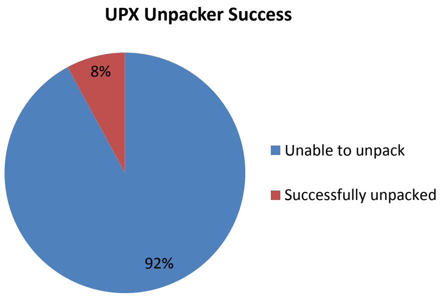


Figure 5.11: Success rate of the UPX unpacker

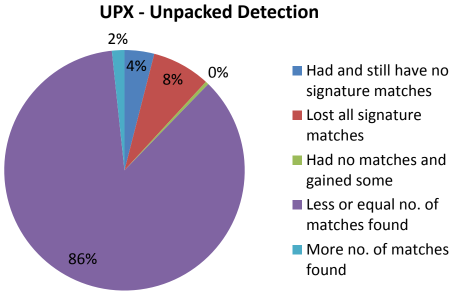
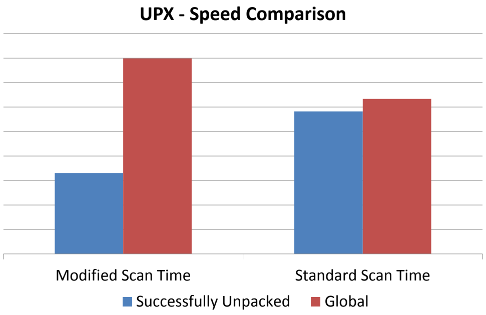


Figure 5.12: Analysis of the UPX unpacker impact in detection



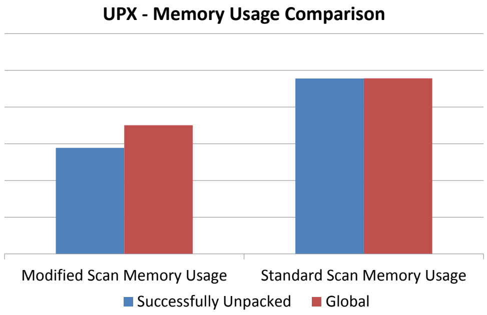


Figure 5.13: Speed comparison between standard and UPX unpacking modified scanner

Figure 5.14: Memory usage comparison between standard and UPX unpacking modified scanner

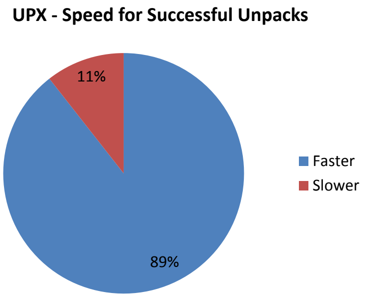


Figure 5.15: For the successfully unpacked samples, speed relative to standard scan

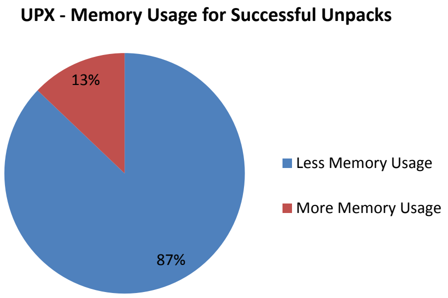


Figure 5.16: For the successfully unpacked samples, memory usage relative to standard scan

The analysis of the results regarding the unpacking of UPX must take a crucial detail into account: the standard scan already had static UPX unpacking support. Although this is true, the support was from an older version of UPX. The new implemented UPX unpacker is able to unpack all the binaries previously supported as well as the newer versions and bug fixes.

The low success rate (8%) is mostly attributed to the fact that the UPX packer and unpacker is open-source. This means that anyone wanting to use UPX to protect their executable might do some slight changes to the packing procedure to render the official UPX unpacker ineffective. This scenario is a lot more appealing because of the availability of the source code. Another reason for the low success rate was the presence of many unpacked samples in the selected sample set. Again, these situations were proven by randomly selecting samples that were unsuccessfully unpacked and analysing them manually.

In the case of UPX, the number of most of the samples' signature matches either remained the same or increased. This was to be expected since most of the samples unpacked by the new implementation were already unpacked in the standard scan. The samples that were not unpacked in the old implementation had naturally different signature matches. Around 3% had more signatures matched to them, supporting the hypothesis of this implementation being able to unpack more samples than before. 8% lost all the signature matches, either because they were not unpacked in the standard scan or because they were only detected in the simulation phase. Also to be expected was the high same detection rate (see Table 5.4) since many of the samples involved could already be unpacked in the previous implementation. The fact that the number of matched signatures went down in total suggests that indeed there were many samples with signature matches in the simulation phase.

It is easy to see that there is a significant performance improvement with the new scanning process, for some of the successfully unpacked samples. In global, however, there was a time penalty in the new UPX unpacker relative to the old one. At the moment of writing, it was not clear why this time penalty occurs. One possible reason is the new implemented code being slower at dealing with unknown samples than the old implementation.

5.3.3 ACProtect 1.41 Analysis

Table 5.5: ACProtect 1.41 Global Results

ACProtect 1.41 Results	
Sample Count	1169790
Unpack Success rate	0,015%
Average Time (relative to old scanning process)	59%
Average Memory (relative to old scanning process)	78%
Old Detection Rate	99,99%
New Detection Rate	43%
Total Detection Count Rate (relative to old scanning process)	158%
Same Detection Rate	95%

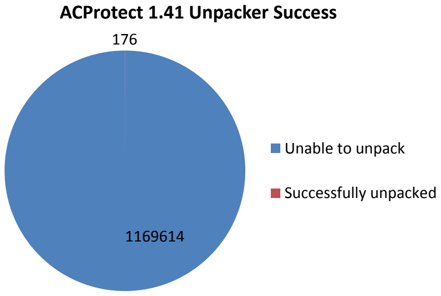


Figure 5.17: Success rate of the ACProtect 1.41 unpacker

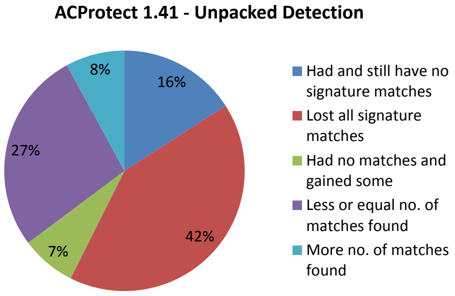


Figure 5.18: Analysis of the ACProtect 1.41 unpacker impact in detection

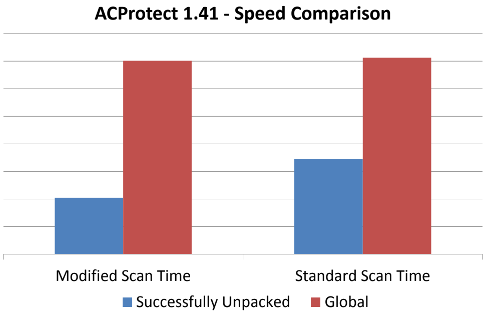


Figure 5.19: Speed comparison between standard and ACProtect 1.41 unpacking modified scanner

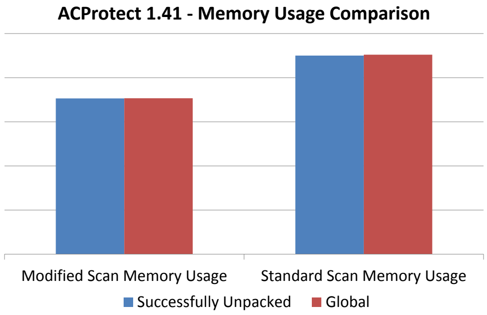


Figure 5.20: Memory usage comparison between standard and ACProtect 1.41 unpacking modified scanner

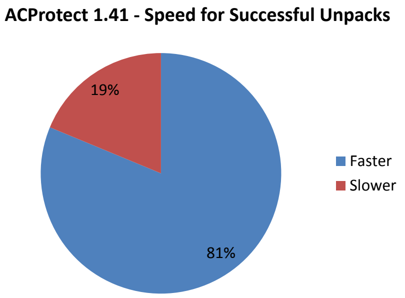


Figure 5.21: For the successfully unpacked samples, speed relative to standard scan

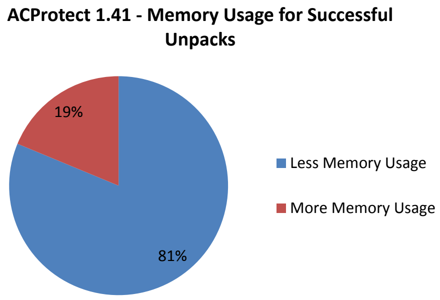


Figure 5.22: For the successfully unpacked samples, memory usage relative to standard scan

The extremely low (0,015%) success rate observed is mostly attributed to the fact that this particular PEiD signature is very short and extremely prone to false positives. Given the unusually low success rate, an attempt was made to try to find a better explanation. Many samples were selected at random from the set of samples that were not successfully unpacked and were analysed manually. Not one of the samples analysed was packed with ACProtect, whatever version it might be. Many UPX packed samples and a few unpacked samples were observed. Although this might account for most of the situation, it is unlikely that this is the only reason. Considering the specific challenges posed by this packer and the solutions implemented to solve them, it is likely that they might have an impact in this issue. Specifically, the decryption of data implemented has its limitations (see section 4.5). The most obvious limitation is the fact that if the decryption loop ends or begins within the confines of the array being decrypted; decryption is not successful and unpacking fails. Although this consideration might have had a significant impact in the results, no sample could be found that confirmed it.

Most of the samples lost some of the signature matches. Considering that only 176 samples were successfully unpacked, these numbers have a relatively small statistic relevance. Given the great loss in detection rate, it is interesting that the total count of signature matches is 58% greater with the new scanning process. This suggests that the samples that lost all signatures might have had signature matches in the unpacking stub section with the standard scan.

Even when considering the whole sample set, including the ones where unpacking was unsuccessful, there was still a performance improvement, although less significant.

5.3.4 ASPack 2.12 Analysis

Table 5.6: ASPack 2.12 Global Results

ASPack 2.12 Results	
Sample Count	783149
Unpack Success rate	67%
Average Time (relative to old scanning process)	76%
Average Memory (relative to old scanning process)	109%
Old Detection Rate	94%
New Detection Rate	84%
Total Detection Count Rate (relative to old scanning process)	96%
Same Detection Rate	83%

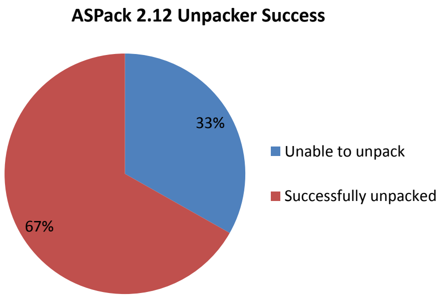


Figure 5.23: Success rate of the ASPack 2.12 unpacker

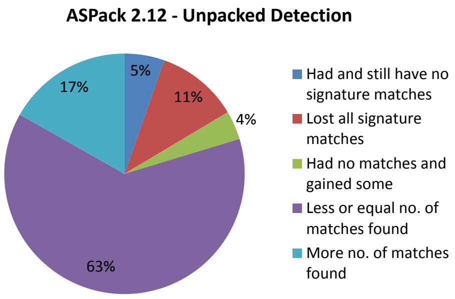


Figure 5.24: Analysis of the ASPack 2.12 unpacker impact in detection

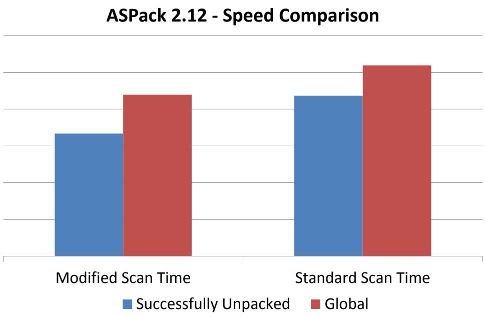


Figure 5.25: Speed comparison between standard and ASPack 2.12 unpacking modified scanner

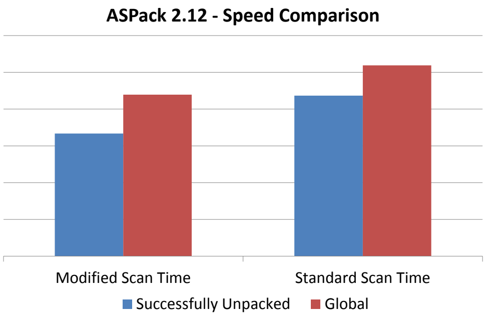


Figure 5.26: Memory usage comparison between standard and ASPack 2.12 unpacking modified scanner

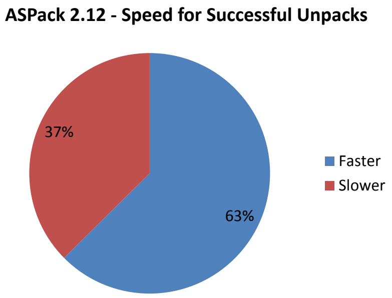


Figure 5.27: For the successfully unpacked samples, speed relative to standard scan

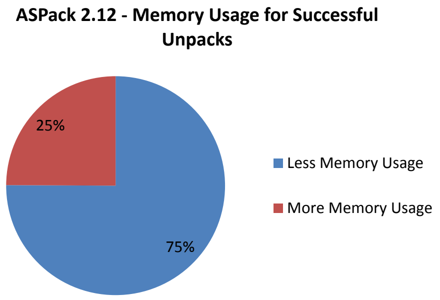


Figure 5.28: For the successfully unpacked samples, memory usage relative to standard scan

67% of the were successfully unpacked. Once again, some samples that were unsuccessfully unpacked were selected at random and analysed manually. Most of them proved to be UPX packed and other proved to be packed by other versions of ASPack, which, naturally, the implementation could not unpack.

Considerations about the signature matches made for the previous packers also apply in the case of ASPack 2.12.

From Figure 5.25, it is clear that there is a significant speed improvement with the new scanning process. Although, on average, there an increase in memory usage, looking at Figure 5.28 it is clear that for most of the samples, the memory usage was lower.

5.4 Conclusions

The results presented in this chapter show that, although with varying levels of success, the implemented unpackers were able to unpack real-world malware samples. Considerations on the effectiveness of the unpackers and on their efficiency and performance were made. Because the developed framework is to be integrated into the scanning engine of the company, performance is a big consideration. Asides from the UPX unpacker that was slower *on average* and ASPack that used more memory *on average*, the implemented static unpackers outperformed the dynamic unpacking done by the simulated execution. Although some of the effectiveness of the scanner was compromised, this is not necessarily a problem, new signatures can be generated over the now "visible" sections and improve the detection rate.

Chapter 6

Conclusions and Further Work

In this chapter, the project is analysed from the point of view of goals achieved and contributions made. The results gathered from testing the different implementations are summarised and briefly discussed. Finally, some suggestions are made for further development and research.

6.1 Achievement of Objectives

In this project, a static unpacking framework was implemented. A clear division was made to the unpacking framework that reflects the two main phases of unpacking: unpacking and rebuilding an executable. To this end, two libraries were developed: *unpacklib* for unpacking data; and *unpackrebuildlib* for fixing the imports and rebuilding an executable with the unpacked data from *unpacklib*. This division allows access to the unpacked data, without a rebuilt executable. For purposes of pure signature-based detection, access to the unpacked data is sufficient. Naturally, if a full-blown executable is needed for deep dynamic analysis, that can be achieved by using the *unpackrebuildlib*. Given the amount of different existing packers, the solution was designed with modularity in mind. Allowing for the incremental addition of support for subsequent packers. In the duration of the project, four different packers were analysed and support for them was implemented in the framework. Due to the lack of availability of unpacking code, most of the packers had to be reversed and an unpacker implemented (see chapter 4). The first packer to be incorporated was FSG 2.0 which is simply a compressor that hides the API imports. The second packer was UPX, another compressor that, being open-source, made implementation simpler. ACProtect 1.41 posed a bigger challenge because of its many protecting features. Also, in the case of ACProtect, an unpacked executable was not produced because it was decided that, given the predicted effort involved, unpacking of data would suffice. The last packer to be integrated in the duration of the project was ASPack 2.12, which was also just a compressor.

As far as the goals of the project go, all the goals were met. An unpacking framework was produced with a modular design. This modular design proved to be appropriate during the integration of support for the different packers. Integration of the framework with the company's malware scanning engine was done and extensively tested. Testing was done on real malware samples identified as being packed by PEiD[Sna] signatures. The different packers supported by the unpacking framework performed differently in testing. 86% of samples identified as being packed with FSG 2.0 were unpacked successfully. 67% in the case of ASPack 2.12. UPX performed less well, with only 8% being successfully unpacked. The worst was ACProtect 1.41, in which only 0,015% of samples were unpacked. A problem that plagued testing was the amount malware samples that were incorrectly identified as being packed with a packer. Having said this, the success of unpacking for the different packers is tightly related to the quality of the PEiD signature, that is, the amount of false-positives. When the samples did get unpacked, there was an average performance improvement in all of the packers when compared to the dynamic unpacking solution of the scanning engine. Time was almost halved and memory usage was reduced in most of the cases.

Detection rates went down in all cases due to a combination of factors. One factor was the loss of behavioural signatures, which were not extracted when unpacking was done. Another reason was the fact that many signatures were previously found in the unpacking stubs. Another factor was malware packed several times which the framework does not address. Only the outer layer of packing is removed in these cases.

The performance improvements provided by the unpacking framework result in a performance improvement for the end-user. This improvement is of great importance due to the real-time nature of the normal operating mode of antivirus software. The detection rate drop is an issue that can be easily addressed by the company. One of the biggest challenges of this project was the development of unpacking algorithms for the different packers. This was mainly due to the lack of information about the packer's behaviour and features. The analyses done in chapter 4 provide a comprehensive understanding of the packers approached. This information can be potentially useful for someone looking to unpack or simply understand the said packers.

6.2 Further Work

Several issues are left unsolved with this implementation. The first and most obvious point that needs further work is the addition of support for more packers. The four packers addressed are but drop of water in an ocean of different packers and versions. As of 2008, Symantec had identified over 2000 different packers belonging to over 200 families[GFC08]. The addition of more complex packers such as newer versions of Armadillo or Themida would be especially useful due to their protection characteristics. The order by which these packers can be integrated in the framework could follow the trends of malware in the wild. Meaning that instead of prioritizing the packers using the simple statistical usage of a database that might have samples that are no longer observed in the wild, the statistical packer usage could be analysed only on the samples currently circulating. This change in the order of integration of packers has a bigger impact on the user since this better reflects the environment the software will ultimately operates in.

An issue not addressed in the solution implemented is the multi-layered packer problem. That is, when a file is packed multiple times, forming several packing layers, one on top of the other. This could be addressed by rebuilding the whole file and re-feeding it into the scanner, allowing for it to be unpacked again if needed.

One problem that clearly needs to be addressed is the quality of the PEiD signatures. This could easily be done in the end of the deep analysis of each packer. The insight gained from this analysis allows for a definition of a good signature.

The signatures lost and gained when scanning unpacked files need to be analysed in detail to ascertain the true reason for the fluctuations. They should be critically reviewed in order to understand if this behaviour is desired or not. As was the case with all the packers except UPX (since an unpacker had already been implemented), signatures were found in packed sections and unpacking stub. This is undesirable and these situations should be reviewed.

Finally, static and dynamic unpacking solutions could be treated as a whole. Treating them as a whole unpacking solution allows for some interesting unpacking algorithms to be defined. These might be useful when unpacking more complex packers.

References

[Böh08] L. Böhne. Pandora's bochs: Automatic unpacking of malware. *University of Mannheim*, 6, 2008.

[Com] Hewlett-Packard Development Company. Hp openvms systems. http://h71000.www7.hp.com/, last accessed on 3rd de June 2013.

[Cora] Microsoft Corporation. Microsoft pe and coff specification. http://msdn.microsoft.com/en-us/windows/hardware/gg463119.aspx, last accessed on 3rd de June 2013.

[Corb] Microsoft Corporation. Microsoft Windows. http://windows.microsoft.com/, last accessed on 3rd de June 2013.

[Corc] Microsoft Corporation. Ms-dos overview. http://www.microsoft.com/resources/documentation/windows/xp/all/proddocs/en-us/windows_dos_overview.mspx?mfr=true, last accessed on 3rd de June 2013.

[Eil05] Eldad Eilam. *Reversing: Secrets of Reverse Engineering*. Wiley, 2005.

[Fur] Gaspar Furtado. Design and integration of a generic unpacking framework for runtime packers. https://feupload.fe.up.pt/get/PrwXLoDAmfKnkBq, last accessed on 3rd de June 2013.

[GFC08] F. Guo, P. Ferrie, and T.C. Chiueh. A study of the packer problem and its solutions. In *Recent Advances in Intrusion Detection*, pages 98–115. Springer, 2008.

[Gop06] Goppit. Portable executable file format – a reverse engineer view. *CodeBreakers Magazine*, January 2006. Available at http://www.woodmann.com/collaborative/knowledge/images/Bin_Portable_Executable_File_Format_%E2%80%93_A_Reverse_Engineer_View_2012-1-31_16.43_CBM_1_2_2006_Goppit_PE_Format_Reverse_Engineer_View.pdf.

[Inc] Sourcefire Inc. Clam antivirus. http://www.clamav.net/lang/en/, last accessed on 3rd de June 2013.

[Kat] Randy Kath. The portable executable file format from top to bottom. http://www.csn.ul.ie/~caolan/publink/winresdump/winresdump/doc/pefile2.html, last accessed on 3rd de June 2013.

[KPY07] M.G. Kang, P. Poosankam, and H. Yin. Renovo: A hidden code extractor for packed executables. In *Proceedings of the 2007 ACM workshop on Recurring malcode*, pages 46–53. ACM, 2007.

[LH07] R. Lyda and J. Hamrock. Using entropy analysis to find encrypted and packed malware. *Security & Privacy, IEEE*, 5(2):40–45, 2007.

[MCJ07] L. Martignoni, M. Christodorescu, and S. Jha. Omniunpack: Fast, generic, and safe unpacking of malware. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pages 431–441. IEEE, 2007.

[New] Jan Newger. Idastealth plugin. http://newgre.net/idastealth, last accessed on 3rd de June 2013.

[NHS] Casey Sheehan Nick Hnatiw, Tom Robinson and Nick Suan. "pimp my pe": Parsing malicious and malformed executables. Technical report, Sunbelt Software.

[Ost02] Russ Osterlund. Windows 2000 loader: What goes on inside windows 2000: Solving the mysteries of the loader. *MSDN Magazine*, March 2002. Available at http://msdn.microsoft.com/en-us/magazine/cc301727.aspx.

[Pie02a] Matt Pietrek. Inside windows: An in-depth look into the win32 portable executable file format. *MSDN Magazine*, February 2002. Available at http://msdn.microsoft.com/en-us/magazine/cc301805.aspx.

[Pie02b] Matt Pietrek. Inside windows: An in-depth look into the win32 portable executable file format, part 2. *MSDN Magazine*, March 2002. Available at http://msdn.microsoft.com/en-us/magazine/cc301808.aspx.

[Reva] ReversingLabs. Titanengine | reversinglabs.com. http://www.reversinglabs.com/resources/open-source/titanengine.html, last accessed on 3rd de June 2013.

[Revb] ReversingLabs. Undocumented pecoff. http://media.blackhat.com/bh-us-11/Vuksan/BH_US_11_VuksanPericin_PECOFF_WP.pdf, last accessed on 3rd de June 2013.

[RHD+06] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee. Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*, pages 289–300. IEEE, 2006.

[RSI12a] Mark E. Russinovich, David A. Solomon, and Alex Ionescu. *Windows® Internals, Part 1: Covering Windows Server® 2008 R2 and Windows 7*. Microsoft Press, 2012.

[RSI12b] Mark E. Russinovich, David A. Solomon, and Alex Ionescu. *Windows Internals, Part 2: Covering Windows Server® 2008 R2 and Windows 7*. Microsoft Press, 2012.

[SA] Hex-Rays SA. Executive summary: Ida pro – at the cornerstone of it security. https://www.hex-rays.com/products/ida/ida-executive.pdf, last accessed on 3rd de June 2013.

[SBY+08] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. BitBlaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security. Keynote invited paper.*, Hyderabad, India, December 2008.

[SH12] Michael Sikorski and Andrew Honig. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. No Starch Press, 2012.

[Sna] Jibz Snaker, Qwerton. Pe identifier (peid). http://tuts4you.com/download.php?view.398, last accessed on 3rd de June 2013.

[Ste] Joe Stewart. Ollybone. http://www.joestewart.org/ollybone/, last accessed on 3rd de June 2013.

[Yus] Oleh Yuschuk. Ollydbg. http://www.ollydbg.de/, last accessed on 3rd de June 2013.