



# Malware Analysis

PACKERS AND UNPACKING

# Introduction

- Packing programs, known as packers, have become extremely popular with malware writers because they help malware hide from antivirus software, complicate malware analysis, and shrink the size of a malicious executable.
- Most packers are easy to use and are freely available.
- Basic static analysis isn't useful on a packed program; packed malware must be unpacked before it can be analyzed statically, which makes analysis more complicated and challenging.
- Packers are used on executables for two main reasons: to shrink programs or to thwart detection or analysis.

# Introduction

- Even though there are a wide variety of packers, they all follow a similar pattern: They transform an executable to create a new executable that stores the transformed executable as data and contains an unpacking stub that is called by the OS.
- When malware has been packed, an analyst typically has access to only the packed file, and cannot examine the original unpacked program or the program that packed the malware.
- In order to unpack an executable, we must undo the work performed by the packer, which requires that we understand how a packer operates.

# Packer Anatomy

- All packers take an executable file as input and produce an executable file as output. The packed executable is compressed, encrypted, or otherwise transformed, making it harder to recognize and reverse-engineer.
- A packer designed to make the file difficult to analyze may encrypt the original executable and employ anti-reverse-engineering techniques, such as anti-disassembly, anti-debugging, or anti-VM. Packers can pack the entire executable, including all data and the resource section, or pack only the code and data sections.
- To maintain the functionality of the original program, a packing program needs to store the program's import information.

# Packer's Anatomy

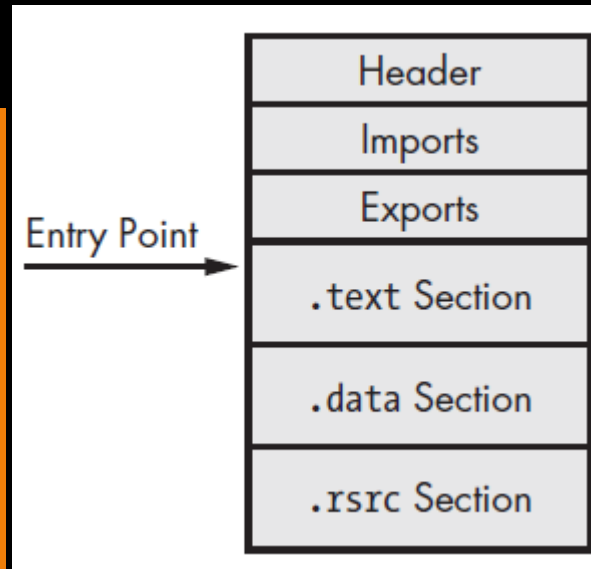
# Unpacking Stub

- With packed programs, the unpacking stub is loaded by the OS, and then the unpacking stub loads the original program.
- The code entry point for the executable points to the unpacking stub rather than the original code.
- The unpacking stub is often small, since it does not contribute to the main functionality of the program, and its function is typically simple: unpack the original executable.
- If you attempt to perform static analysis on the packed program, you will be analyzing the stub, not the original program.

# Unpacking Stub

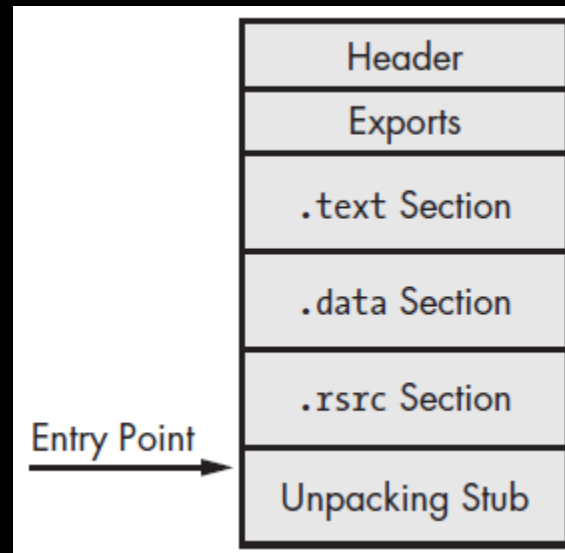
- The unpacking stub performs **three steps**:
  - Unpacks the original executable into memory
  - Resolves all of the imports of the original executable
  - Transfers execution to the original entry point (OEP)
- Note: The instruction that transfers execution to the OEP is commonly referred to as the tail jump.
- Packed executables format the PE header so that the loader will allocate space for the sections, which can come from the original program, or the unpacking stub can create the sections.
- The unpacking stub unpacks the code for each section and copies it into the space that was allocated. The exact unpacking method used depends on the goals of the packer, and it is generally contained within the stub.

# Unpacking Stub



## Original Executable

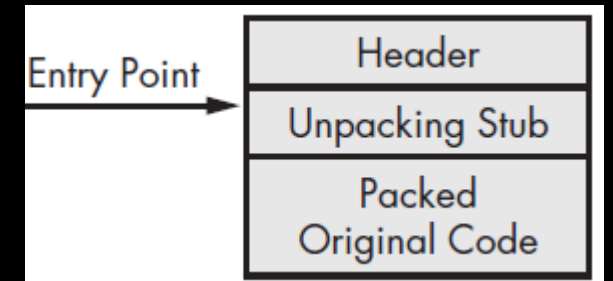
The header and sections are visible, and the starting point is set to the OEP



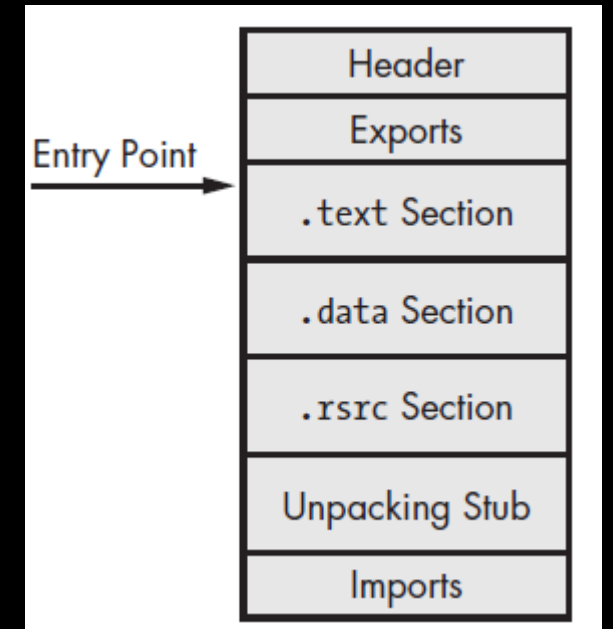
## Packed Exe loaded into memory

Stub has unpacked the original code, and valid .text and .data sections are visible.

The program's starting point still points to the unpacking stub, and there are no imports.



**Packed Executable on Disk**  
original code is packed and the unpacking stub is added



## Fully Unpacked Exe

Import table is re-constructed by Stub, and the starting point is back to the OEP.

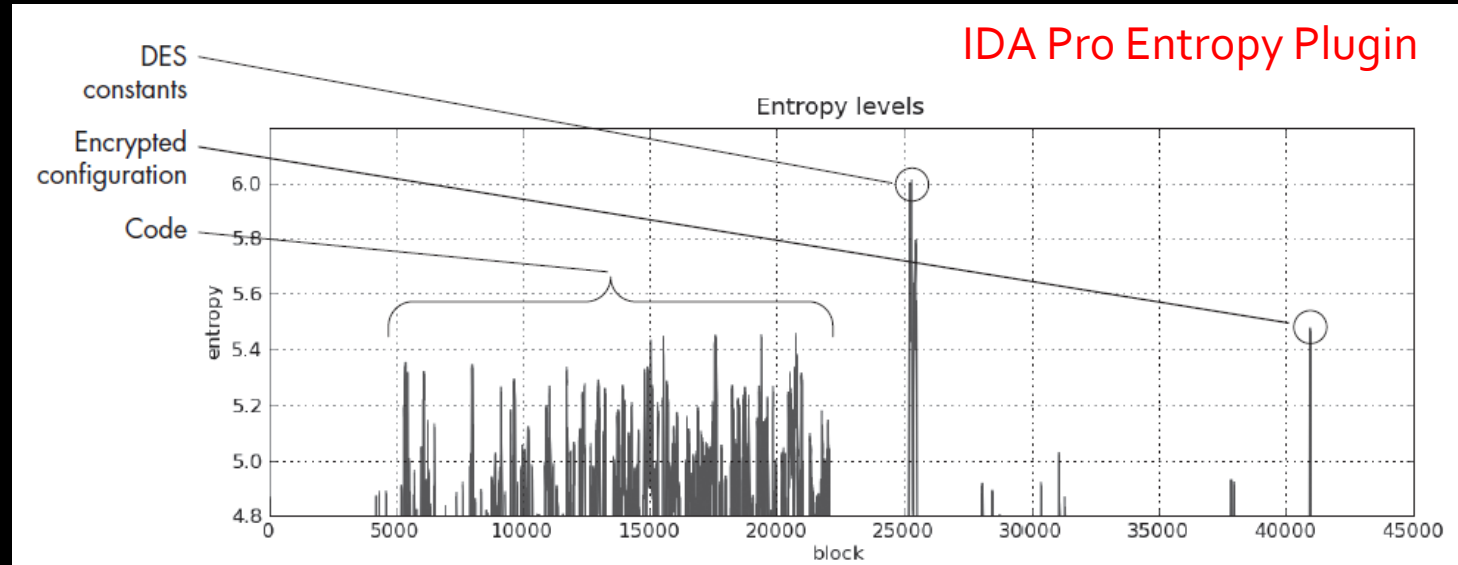
# Indicators of Packed Program

- The program has few imports, and particularly if the only imports are LoadLibrary and GetProcAddress.
- When the program is opened in IDA Pro, only a small amount of code is recognized by the automatic analysis.
- When the program is opened in OllyDbg, there is a warning that the program may be packed.
- The program shows section names that indicate a particular packer (such as UPX0).
- The program has abnormal section sizes, such as a .text section with a Size of Raw Data of 0 and Virtual Size of nonzero.

# Entropy Calculation

- The Packed executables can also be detected via a technique known as entropy calculation.
- Entropy is a measure of the disorder in a system or program, and while there is not a well-defined standard mathematical formula for calculating entropy, there are many well-formed measures of entropy for digital data.
- **Compressed or encrypted data** more closely resembles random data, and therefore has **high entropy**; executables that are **not encrypted or compressed** have **lower entropy**.
- Automated tools for detecting packer used entropy calculation free automated tool is Mandiant Red Curtain, that can scan a filesystem for suspected packed binaries

# Entropy Calculation



- The graph shows only high values, from 4.8 to 6.2. Recall that the maximum entropy value for that chunk size is 6.
- the plateau between blocks 4000 and 22000. This represents the actual code, and it is typical of code to reach an entropy value of this level.
- A more interesting feature is the spike at the end of the file to about 5.5. The fact that it is a fairly high value unconnected with any other peaks makes it stand out. When analyzed, it is found to be DES-encrypted configuration data for the malware, which hides its command-and-control information.

# Unpacking Packer

- There are three options for unpacking a packed executable:
  1. Automated static unpacking
  2. Automated dynamic unpacking
  3. Manual dynamic unpacking

## Automated static unpacking

- Automated static unpacking programs decompress and/or decrypt the executable.
- This is the fastest method, and when it works, it is the best method, since it does not run the executable, and it restores the executable to its original state.
- Automatic static unpacking programs are specific to a single packer.
- PE Explorer, a free program for working with EXE and DLL files, comes with several static unpacking plug-ins as part of the default setup. The default plug-ins support NSPack, UPack, and UPX.

# Automated dynamic unpacking

- Automated dynamic unpackers run the executable and allow the unpacking stub to unpack the original executable code.
- Once the original executable is unpacked, the program is written to disk, and the unpacker reconstructs the original import table.
- Note: The automated unpacking program must determine where the unpacking stub ends and the original executable begins, which is difficult.

# Manual dynamic unpacking

- Manual unpacking can sometimes be done quickly, with minimal effort; other times it can be a long, arduous process.
- There are two common approaches to manually unpacking a program:
  - Discover the packing algorithm and **write a program to run it in reverse**. By running the algorithm in reverse, the program undoes each of the steps of the packing program.
  - Run the packed program so that the unpacking stub does the work for you, and then **dump the process out of memory, and manually fix up the PE header** so that the program is complete. This is the more efficient approach.

# UPX Packer

- The most common packer used for malware is the Ultimate Packer for eXecutables (UPX). UPX is open source, free, and easy to use, and it supports a wide variety of platforms.
- UPX **compresses the executable**, and is designed for performance rather than security.
- UPX is popular because of its high decompression speed, and the small size and low memory requirements of its decompression routine.
- It is not designed to be difficult to reverse-engineer, and it does not pose much of a challenge for a malware analyst.
- Most programs packed with UPX can be unpacked with UPX as well, and the command line has a -d option that you can use to decompress a UPX-packed executable.

# PE Compact & ASPack

- **PECompact:**
  - PECompact is a commercial packer designed for speed and performance.
  - Programs packed with this packer can be difficult to unpack, because it includes anti-debugging exceptions and obfuscated code.
- **ASPack:**
  - ASPack is focused on security, and it employs techniques to make it difficult to unpack programs.
  - ASPack uses self-modifying code, which makes it difficult to set breakpoints and to analyze in general.

# PE Compact & ASPack

- **Petite:**
  - Petite is similar to ASPack in a number of ways.
  - Petite also uses anti-debugging mechanisms to make it difficult to determine the OEP, and the Petite code uses single-step exceptions in order to break into the debugger.
- **WinUpack:**
  - WinUpack is a packer with a GUI front end, designed for optimal compression, and not for security.
  - There is a command-line version of this packer called UPack, and there are automated unpackers specific to Upack and WinUpack.

# Themida

- Themida is a very complicated packer with many features. Most of the features are **anti-debugging** and **anti-analysis**, which make it a very secure packer that's difficult to unpack and analyze.
- Themida contains features that prevent analysis with VMware, debuggers, and Process Monitor (procmon). Themida also has a kernel component, which makes it much more difficult to analyze.
- Some automated tools are designed to unpack Themida files, but their success varies based on the version of Themida and the settings used when the program was packed.

# References

- Practical Malware Analysis by Michael Sikorski