A Review of Common Practices for Malware Unpacking

Facoltà di
Dipartimento di
Corso di laurea in Cybersecurity

Abu Jafor Mohammad Saleh
Matricola 1469151

Relatore	Correlatore
Daniele Cono D'Elia	Leonardo Querzoni

A.A. 2022-2023

Contents

1 - Abstract

In today’s digital world software security is a very important aspect and plays a vital role in protecting people, information and intellectual properties from malicious users. However, malicious users develop day by day more sophisticated strategies to achieve their goals by exploiting from old to newly discovered vulnerabilities and targeting the weakest chain of the cyber security, the human.

With more sophisticated attacks the protection mechanism to detect and respond to these attacks also becomes sophisticated. Due to this reason malicious attackers implements protection layers in particular to malware attacks in order to evade detection and analysis. To such purpose packer software is the perfect solution.

The study of packers and their impact on software security and cyber-attacks is crucial for understanding the techniques employed by both malicious actors and legitimate software developers. This thesis aims to provide an exploration of packers, including their mechanisms, functionality, and implications for software analysis.

The primary objective of this thesis is to analyze and evaluate different packer techniques, dissecting their inner workings and investigating the challenges they pose to traditional security analysis approaches. By examining various packers' features, we seek to increase our understanding of their strengths, weaknesses, and potential mitigations. Furthermore, this thesis aims to propose a possible strategy to analyze advanced packers by exploring innovative approaches.

The chapter 2 and 3 introduce the concept of packers and their basic functionalities, their inner work, and different types of packers. These chapters explain the purpose of packing how they were born and how they were used differently from their initial purpose for malicious intents.

Chater 3 and 4 goes deep down on analyzing these packers understanding different approaches and strategies, their advantages, and disadvantages, and how they can be combined to have a more complete understanding of if the packed malware. I particularly the chapter 5 give guidance on how to set up a possible virtual environment for analyzing such malware. Furthermore, it describes how to identify if an executable is packed and hides malicious payload inside by analyzing various packing indicator. Lastly, it shows

step by step how to unpack a simpler version UPX and MPRESS packers for a demonstration purpose and how to extract the packed code.

Chapter 6 is focus on an advanced technique to identify packers, as they can implement techniques to hide indication from packer analysis tools. This chapter explains what feature to extract and how to analyze them in order to identify packer indications.

Chapter 7 and 9 introduce advanced packers, their behaviour and evasion techniques. The analysis of these packers is not trivial, and they are time consuming. This chapter describes the techniques employed by the packer to evade analysis by malware analyst. Complex packers described in the chapter that implement such advanced evasion techniques are Themida and PEzoNG.

Chapter 8 and 9 propose a solution for advanced packers implementing diverse evasion techniques making analyzing them time consuming and hard. The chapters propose an execution path strategy in order to execute all the branches and a signature-based technique to detect pattern a logic flow to build a database of signature for future detection.

The chapter 10 studies the possibility to perform packing and unpacking on embedded systems and IoT. As they have a very limited computational powers executing advanced evasion technique may not be possible to implement. The chapter also studies the performance of UPX and RSA packing on such systems

In conclusion, this thesis aims to enlighten the complex world of packers, providing valuable insights into their workings, their impact on software security, and potential countermeasures. By reducing the gap between packer technologies and security analysis, we strive to give a good understanding of their potential and utilization.

2 - Introduction to Packers

2.1 - Packing Purpose

Packing is a technique that is widely used to compress software to reduce in size, often it improves performance, protects intellectual properties, and prevents tamper. One of the main purposes of packing software is to reduce the size which makes the distribution and the download of the executable faster compared to its original size, especially for big software. This is really useful, for software distributed over the internet, which makes it convenient for users.

Another benefit of packing is to protect intellectual properties by hindering third parties from reverse engineering and decompiling the code. This can prevent competitors from stealing the code or creating a similar product. Various techniques of packing can be implemented in order to protect the software.

Malicious users use the packing technique to avoid detection mechanisms by detection software such as Firewalls and Intrusion Detections Systems (IDS) by compressing or encrypting the malicious code. This makes it difficult to be detected by the protection mechanisms.

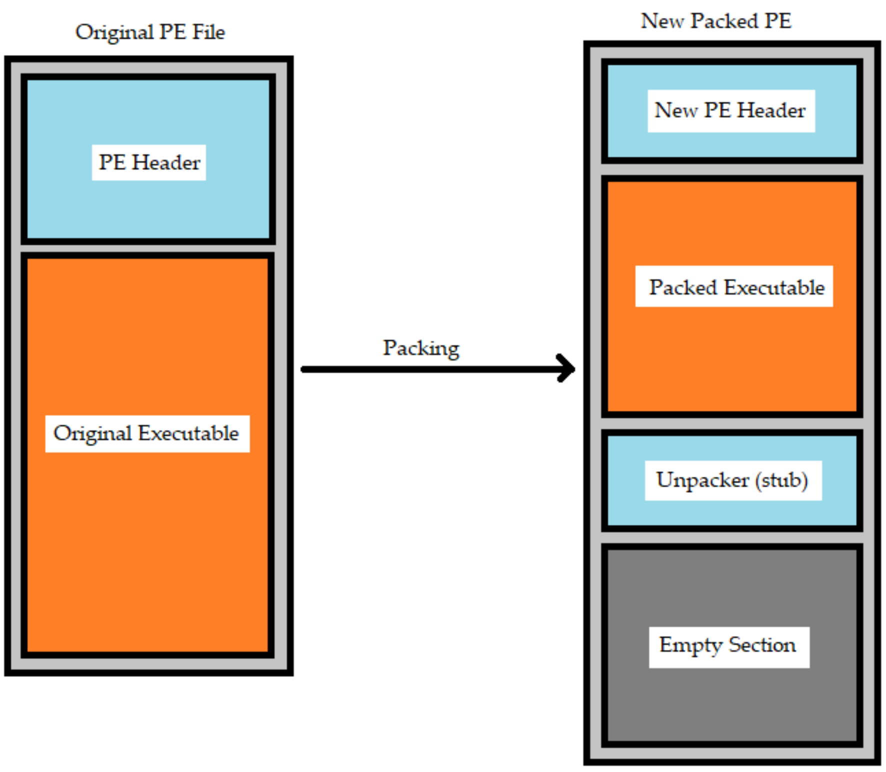
As further protection packers can implement anti-debugging and anti-tampering techniques on the malware. Malware packed in such mechanisms makes really difficult to be analyzed. Anti-debugging technique detects and tries to prevent debugging attempts by checking for example breakpoint which is a clear indication of debugging. Anti-tampering mechanisms on the other hand tried to detect changes in the code and a possible way can be by computing checksums and digital signatures. If the mentioned actions are detected by the malware, it may terminate or alter its behavior as a protection mechanism to mislead malware analysts.

Finally, the main purpose of malware packers is to increase the chances of successfully infecting a target system, avoiding detection by security measures, making the malware hard to analyze in order to understand how internally it works and mitigate any infection.

2.2 - How Packers Work

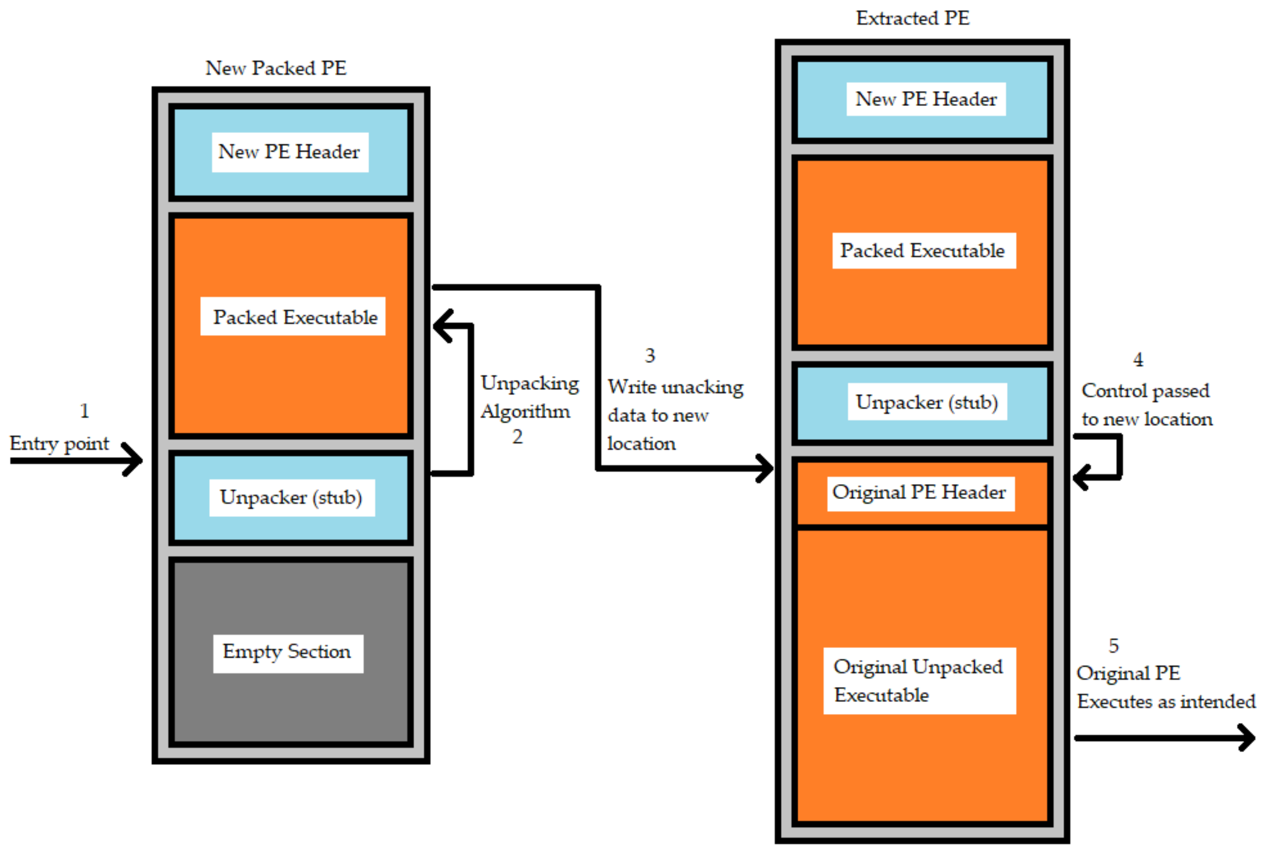
During the packing process, the packing software takes as input the original executable file (PE) and generated a new packed file. During the packing process, the packer can encrypt or compress the original PE file or a combination of both. This newly packed file contains the original PE file and the code to unpack it during the execution usually called "stub" which is the new entry point of the executable. The stub is usually designated to unencrypt or decompress the original PE but, in some cases, it can also encrypt and compress back the code.

The following image shows an example of how the executable file changes before and after the packing process:



The newly packed PE file has a different structure than the original PE, trying to analyze this file will result in an unsuccessful attempt as the real code is not extracted yet.

The packer can modify the PE header or remove it completely in an attempt to evade detections by security systems or malware analysts. The packer can also add some additional code as anti-debugging and anti-tampering technique. When the packed PE is executed the first entry point is the stub which extracts the packed code to be unpacked. Once the extractions finishes, the control is passed to the unpacked code to be executed. The following image shows the steps involved during this process:



Understanding how the packing process works and how the PE section can be modified is as important as to keep the system up to date, use strong passwords, and avoid getting files from untrusted sources to avoid getting infected.

The structure of a packed PE file is complex as it is usually customized by malware writers to obfuscate the malicious code. During the analysis of a packer the main objective is to find the Original Entry Point (OEP), which is the starting point of the original malicious code. Identifying the OEP is not always trivial as malware authors can implement techniques to make this step difficult.

Overall, the process of packing malware is constantly evolving, and malware authors are continually developing new techniques to evade detection. As security researchers, it's crucial to keep up with the latest malware-packing methods to detect and prevent.

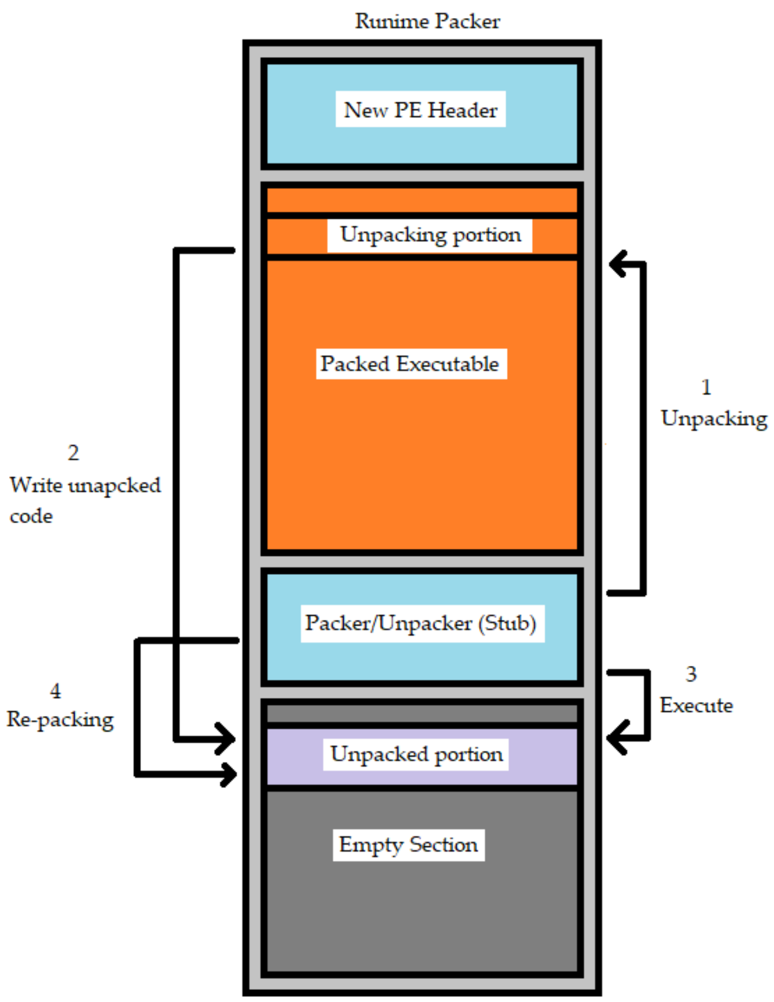
3 - Standard Packer vs Runtime Packer

One of the main advantages of using a standard packer is that it can compress the executable file, reducing its size on disk. This can be beneficial for distributing files over networks or on portable storage devices. However, because the packed file needs to be unpacked before it can execute normally, it can also make analysis and debug more difficult.

A standard packer typically works by compressing or encrypting the original executable file and then appending a small piece of code to the file. This code is responsible for decompressing or decrypting the rest of the file and generating the packed PE to be executed.

In contrast, a runtime packer modifies the executable file in memory, rather than on disk. There is a runtime component that is responsible for extracting each time the portions of code that the program needs and possibly removing those that are not currently in use. This makes it more difficult to detect and analyze the packed code since the modifications are not visible on the disk. Because runtime packers modify the program's memory space, they can also be used to protect against memory-based attacks, such as buffer overflows. By randomizing the memory layout of the program, a runtime packer can make it more difficult for attackers to exploit vulnerabilities in the code.

Different malware can implement different strategies to obfuscate the unpacked part each time. The one shown below is a simplified behaviour as they vary from malware to malware. The following image shows a simplified series of steps performed by the runtime packer:



However, there are also some drawbacks to using a runtime packer. One of the main disadvantages is that it can make debugging and analysis more difficult since the packed code is not visible on the disk. In addition, because the packer code is executed in memory, it can be more vulnerable to anti-tampering techniques, such as code obfuscation or anti-debugging measures.

Overall, both packers and runtime packers are used to make malware analysis more difficult. While standard packers compress or encrypt an executable file on a disk, runtime packers modify the executable in memory. The use of runtime packers is becoming more common among malware authors, as they can be more difficult to detect and analyze. However, in order to increase even more the complexity of the analysis different combinations of different techniques can be put in place, for example, the technique previously mentioned in combination with parallel unpacking using child processes or threads will add complexity during the analysis phase.

4 - Different approaches to analyze packed malware

4.1 - Introduction

Malware analysis is a critical aspect of the cybersecurity field as it helps to identify, understand, and mitigate malicious software. Malware can be designed to cause various types of harm, such as stealing sensitive data, damaging systems, or hijacking resources for nefarious purposes. To counter such threats, security professionals use different methods and tools to analyze malware and gain insights into its behavior.

There are three primary approaches to analyzing malware: static analysis, dynamic analysis, and hybrid analysis. Each technique has its own strengths and weaknesses, and security analysts choose the appropriate method based on their objectives and the type of malware they are dealing with.

Static analysis involves analyzing the malware's code without actually executing it. One of the main advantages of static analysis is that it is a non-intrusive method, which means that the malware is not actually run on the system, reducing the risk of contamination and being discovered by the malware writer that his malware is being analyzed. This also prevents future patches by the malware writer in order to make the malware harder to be detected.

Additionally, static analysis can be performed quickly, making it an effective method for identifying known malware. For such purposes, various tools and techniques can be used such as disassembling the executable code, examining metadata, analyzing strings, and checking the digital signature of the file.

Dynamic analysis involves running the malware in a controlled environment, such as a virtual machine or sandbox, to observe its behavior. This technique is useful for detecting malware that is designed to evade detection or that requires specific environmental conditions to activate. Dynamic analysis techniques include monitoring system calls, network traffic, and file system changes.

The dynamic analysis provides detailed insights into the behavior of the malware, allowing analysts to identify the type of malware, its purpose, and the systems it interacts with. However, dynamic analysis can be a time-consuming process, and there is a risk of contamination if the malware is not properly contained in the analysis environment.

During this step, it is important to take all the precautions to safeguard the malware. It is important to execute it with due diligence and make sure the testing environment is isolated. Also, it is important to take a safe copy of the malware or execute continuous snapshots of the testing environment in case the malware deletes itself, overwrites portions of code, or implements additional defence mechanism to void being analyzed.

The hybrid analysis, a combination both static and dynamic analysis techniques, provides a comprehensive understanding of the malware. This technique involves first performing a static analysis to identify the malware's characteristics, followed by a dynamic analysis to observe its behaviour. Hybrid analysis can be an effective method for detecting advanced malware that is designed to evade detection or that uses sophisticated techniques to obfuscate its code.

The initial static analysis and the discovery of information in this phase are really helpful during the dynamic analysis phase. The static analysis gives an overview and an idea of the malware and what to expect from it, static analysis can also reduce the time consumed by the dynamic analysis phase in case some key information are found.

4.2 - Static Analysis

One effective method of analyzing malware is through static analysis, which involves examining the code and structure of a malware sample without executing it. One of the primary techniques used in static analysis is disassembly, also known as reverse engineering. Disassembly involves converting the binary code of a program into human-readable assembly code that can be analyzed and understood.

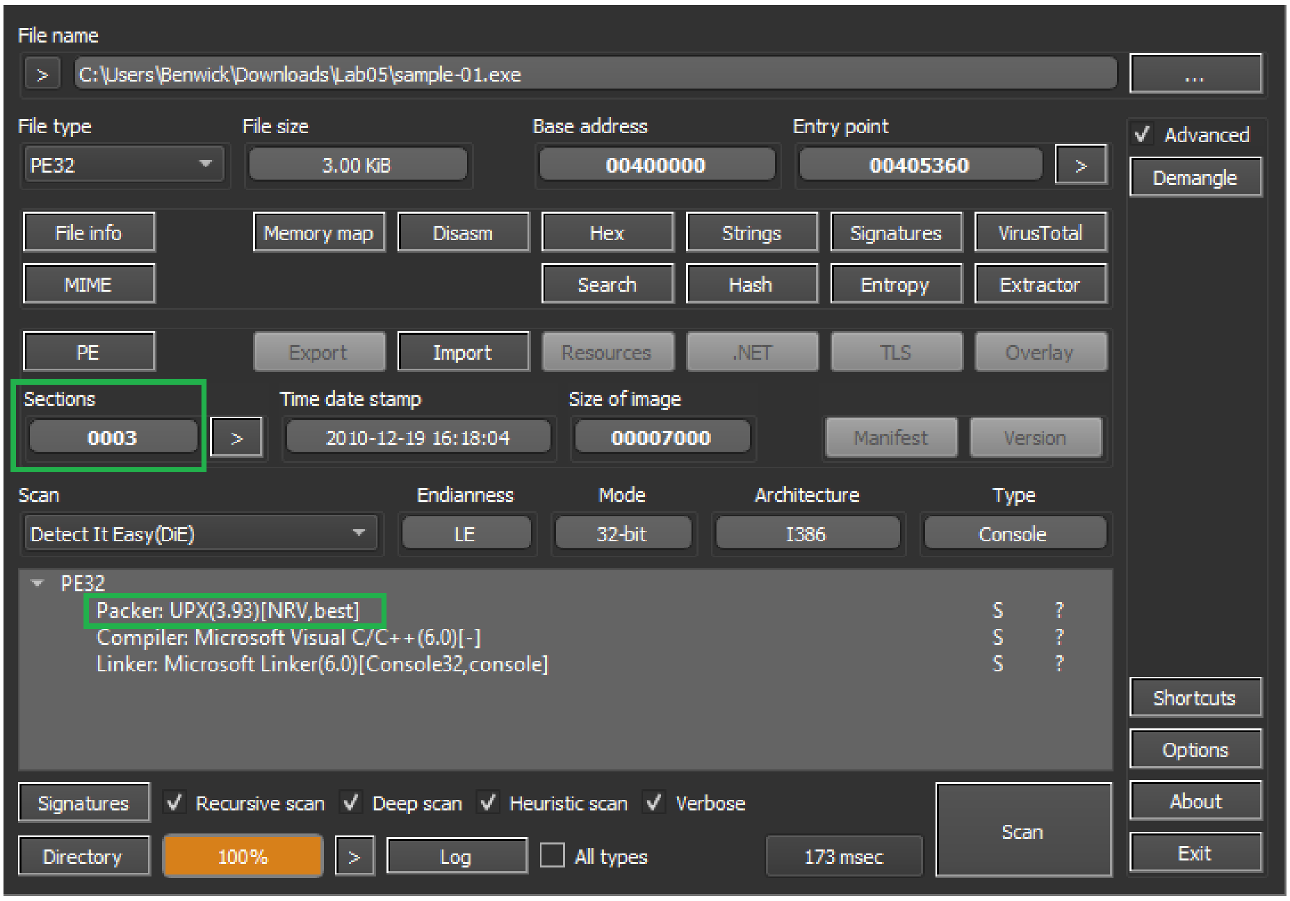
During static analysis, it is possible to extract a wide range of information about malware, including the strings contained in the malware, the system calls used, and the entropy if the malware was packed. This information can help analysts understand the intent of the malware, its capabilities, and potential weaknesses that can be exploited.

"Basic static analysis consists of examining the executable file without viewing the actual instructions. Basic static analysis can confirm whether a file is malicious, provide information

about its functionality, and sometimes provide information that will allow you to produce simple network signatures. Basic static analysis is straightforward and can be quick, but it's largely ineffective against sophisticated malware, and it can miss important behaviors [1]".

In the event of packed malware, the aim at this stage is to get a general idea of what the malware might be, i.e., whether the malware is really packed, whether there are indicators to confirm this, and the type of packing it might implement.

The following packing indicator is taken from DetectItEasy (DIE). With DIE, it is possible to identify packers and retrieve useful information about the malware sample. For instance, when analyzing a malware sample, DIE can identify whether malware is packed or not. In addition, DIE can retrieve the version number of the packer, in this specific case as shown below is 3.93, which can be used to identify known vulnerabilities and weaknesses in the packer itself which is useful during the analysis phase.


File name: C:\Users\Benwick\Downloads\Lab05\sample-01.exe

File type: PE32 | File size: 3.00 KiB | Base address: 00400000 | Entry point: 00405360 | ✓ Advanced

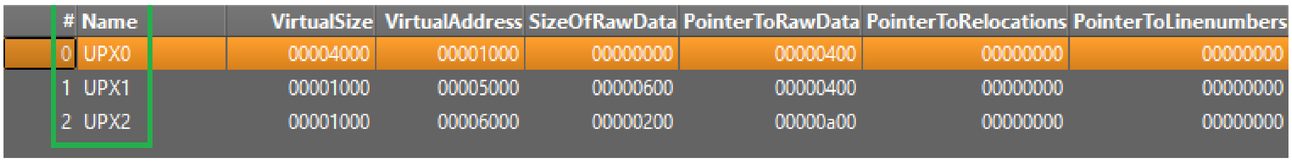
Sections: 0003 | Time date stamp: 2010-12-19 16:18:04 | Size of image: 00007000

Scan: Detect It Easy(DiE) | Endianness: LE | Mode: 32-bit | Architecture: I386 | Type: Console

```
PE32
    Packer: UPX(3.93)[NRV,best]                      S   ?
    Compiler: Microsoft Visual C/C++(6.0)[-]         S   ?
    Linker: Microsoft Linker(6.0)[Console32,console] S   ?
```

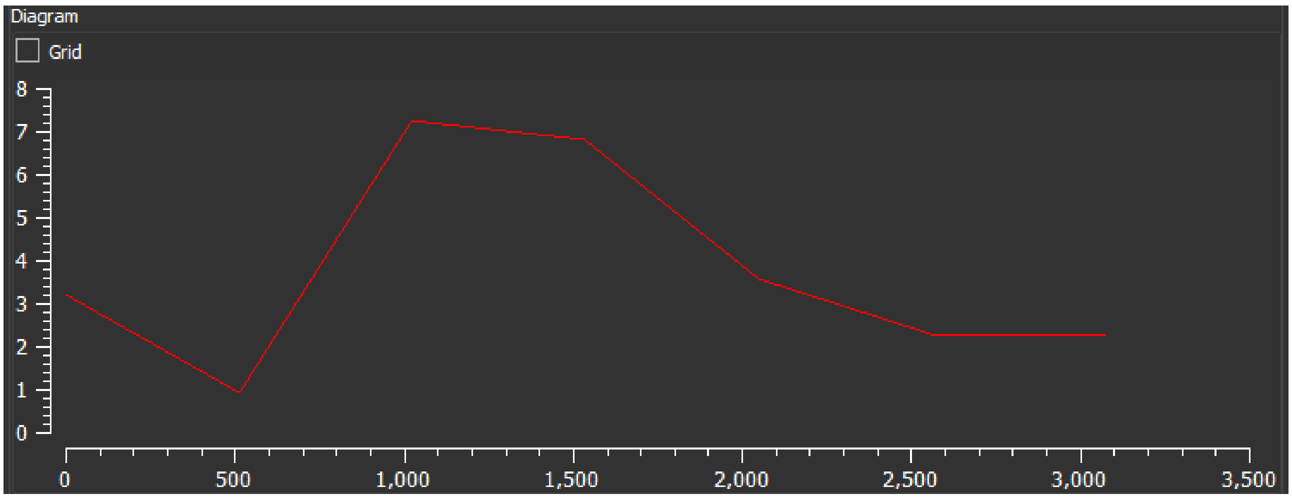
✓ Recursive scan ✓ Deep scan ✓ Heuristic scan ✓ Verbose ☐ All types | 100% | 173 msec

A useful feature of DIE is the capability to obtain the number of sections from the packed file. This can provide valuable information on the internal structure of the malware and help to confirm the presence of the packer. In fact, from the malware sample analyzed, DIE found three sections containing UPX in the name.

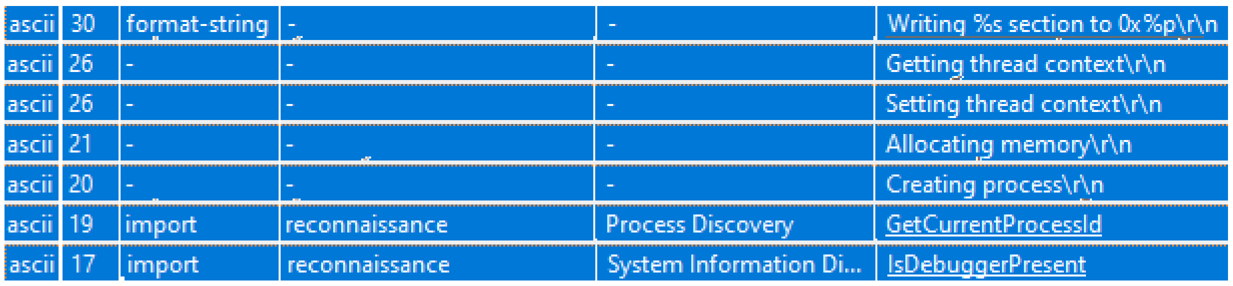


#	Name	VirtualSize	VirtualAddress	SizeOfRawData	PointerToRawData	PointerToRelocations	PointerToLinenumbers
0	UPX0	00004000	00001000	00000000	00000400	00000000	00000000
1	UPX1	00001000	00005000	00000600	00000400	00000000	00000000
2	UPX2	00001000	00006000	00000200	00000a00	00000000	00000000

To further confirm whether the malware is packed or not, it is possible to check the entropy of the file. Since packed files tend to have a high entropy due to the compression and encryption techniques used, checking the entropy can help to confirm the presence of the packer. As shown in the picture there is a spike in the entropy graph reaching the value of 7 which we can consider enough high to say that the malware is packed.



Another advantage of static analysis is that it can be performed on multiple samples simultaneously, allowing for rapid identification of commonalities and patterns across different malware variants. This can help analysts identify and understand larger malware families and the techniques they use to evade detection. This can be performed from the command line for example with the command "strings" and grepping key strings such as "host", "IP" or "socket" in search of an external connection to the command and control or via Yara rules that can identify patterns and advanced metrics.



ascii	30	format-string	-	-	Writing %s section to 0x%p\r\n
ascii	26	-	-	-	Getting thread context\r\n
ascii	26	-	-	-	Setting thread context\r\n
ascii	21	-	-	-	Allocating memory\r\n
ascii	20	-	-	-	Creating process\r\n
ascii	19	import	reconnaissance	Process Discovery	GetCurrentProcessId
ascii	17	import	reconnaissance	System Information Di...	IsDebuggerPresent

The subset of findings presented in the image above reveals important information about the behaviour of the analysed malware. One of the notable indicators is the string 'Writing %s section to 0x%p\r\n', which suggests that the unpacked code is written into memory due to the presence of packing. Additionally, the strings 'Getting thread context' and 'Setting thread context' imply that the malware employs threads to perform certain operations. The occurrence of the strings 'Creating process' and 'GetCurrentProcesID' is indicative of the creation of new processes by the malware.

Moreover, the string 'IsDebuggerPreset' raises the possibility that the malware attempts to detect whether it is being analysed and may modify its behaviour accordingly. This underscores the importance of using anti-analysis techniques by malware authors to evade detection and analysis by security researchers. This gives an idea of what the malware might do and how it might behave. This is very useful if, after the first static analysis phase, we proceed to a dynamic analysis, so that we already know what to expect from the malware.

However, there are also limitations to static analysis. For instance, some malware variants use advanced obfuscation techniques to make their code difficult to analyse, such as using anti-disassembly techniques or encrypting critical sections of their code. Additionally, static analysis alone cannot provide a complete understanding of how the malware operates, as it cannot account for the behaviour of the malware in different environments or the impact of the malware on the system.

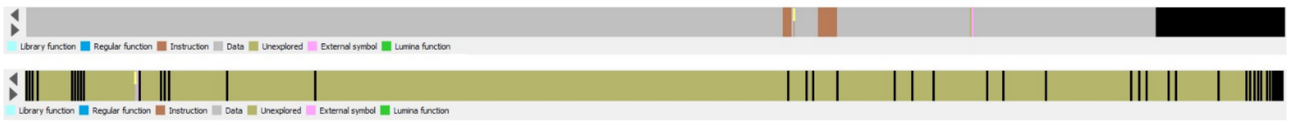
4.3 - Dynamic Analysis

Malware has become a significant threat to computer systems and networks, and its proliferation is increasing day by day. To counter this, various techniques have been developed to analyze malware and to understand its behavior, the operations it performs, and its objectives. For such purposes, dynamic analysis is used. It is the process of observing the behavior of a piece of malicious software and understanding its purpose.

The primary goal of dynamic analysis is to understand the behavior of the malware when it is executed. Dynamic analysis can help in identifying the malware's objectives, capabilities, and methods of propagation. In the case of packed malware dynamic analysis can help to find the critical pieces of code such as the stub which is the code used to extract the packed section or the tail jump, a jump instruction pointing to the original malicious code.

Dynamic analysis can also help in identifying the presence of any anti-analysis or evasion techniques used by the malware, in the event of packed malware for example it can monitor the read-and-write system calls of unpacked code and identify changes in the memory sections. It can also determine the malware's impact on the system or network, such as changes in the registry or file system, network communication, and system resource utilization.

The following image is the representation of the memory segment before and after executing the packed malware and it clearly indicates the changes it does and the extraction in memory of the packed code.



In particular with dynamic analysis, it is possible to monitor system calls, by monitoring them it is possible to have a brief understanding, in some cases detailed and precise, of what operations the malware is performing and what can be the purpose of such. For example, if we find the socket() system call, we can assume that the malware most likely communicates with a remote host that could be the command and control (C&C). The purpose of such communication could be to receive remote commands to perform operations, exfiltrate data and send it to the remote host, or even update the malware.

Another advantage of running malware is that one can see the changes it makes to the system. Malware can make different changes to the system such as changes to the file system, registry, and system resources. These changes can, for instance, be intended to persist in the event of a system restart, to hide from analysts or security software, or create DLL files.

With dynamic analysis, it is also possible to analyze multi-threaded or multi-process malware. The creation and execution of these threads or processes are not immediate via static analysis. In fact, it is easier to run the malware, and then dump the threads or the processes it created. Even if most of this information could be found by static analysis, it involves very long analysis times that can be reduced by dynamic analysis, especially in the case of packed malware where the original code is obfuscated.

Dynamic analysis has its limitations, mainly due to the complexity of modern malware. Malware authors implement a variety of techniques to evade detection, such as anti-debugging and anti-virtualization techniques. They also use advanced encryption and obfuscation methods to make the malware more difficult to analyze and they can also be designed to detect the presence of a sandbox or virtual environment. Malware often alters its behavior accordingly in presence of an analysis environment or debuggers leading the analyst in the wrong direction.

4.4 - Hybrid (Sandbox) Analysis

With dynamic analysis, it is possible to analyze the behavior of malware in a controlled environment. The main goal of dynamic analysis is to observe the behavior of malware when executed which otherwise could not be understood. Dynamic analysis can help in identifying the malware's intent, capabilities, and methods of propagation. For example, it can also identify the malware's impact on the system or network, such as changes in the registry or file system, creation of files, creation of scheduled tasks in order to survive a reboot, network communication, and system resource utilization. Dynamic analysis can also help in identifying the presence of any anti-analysis or evasion techniques used by the malware. This is especially useful in presence of thoroughly packed malware or a new packing technique in which unpacking is time-consuming. In this way, it is possible to understand up to a certain degree the analyzed malware and implement a provisional countermeasure.

The hybrid analysis differs from dynamic analysis in its scope. It combines both static and dynamic analysis. Further, it includes an analysis of external elements in which the malware may operate such as the network. Hybrid analysis does not limit to executing the malware, but it also creates a virtualized environment, even with several machines in order to allow the malware to perform the operations it would typically execute in a real-world environment.

However, there are also some limitations to hybrid analysis. For example, some malware may be designed to detect the presence of a VM or sandbox by implementing checks on the number of cores, number of RAM, and other metrics to understand if it is running in a genuine or controlled environment and alter its behaviour accordingly.

5 - Simple Packing and Unpacking Techniques

5.1 - Premise

This chapter will further introduce packers and give some in depth knowledge about internal structure of a Portable Executable (PE) file to have a good understanding of each unpacking steps performed later on. Additional it will present a detailed guide on how to carry out static and dynamic analysis, highlighting their key differences as well as their pros and cons. The tools required for each step will be listed and explained in order to provide a comprehensive understanding. The environment setup will also be described so that it can be replicated and the reasoning behind such a strategy can be understood. The main objective of this chapter is to demonstrate a possible method of unpacking UPX and MPRESS packers in detail and provide insights into the expected outcomes.

5.2 - Popular Packers

UPX and MPRESS are two popular packing commonly used in malware packing. UPX is an open-source executable packer that can compress executables and DLLs without damaging or changing the original code and it is completely self-contained. It uses a combination of multiple compression algorithms, including LZMA, Huffman coding, and Run Length Encoding (RLE), to compress the code. UPX supports a wide range of executable formats, such as Windows PE and Linux ELF. UPX is widely used by malware authors as it is easy to use and can quickly compress executables.

UPX is a popular packer in the malware development community due to its effectiveness in reducing the size of the executable and the availability of both encryption and compression. However, it is also well-known among security researchers and antivirus vendors, and many have developed signatures and heuristics to detect and analyze UPX-packed executables. As a result, malware developers have increasingly turned to more sophisticated packing algorithms, such as MPRESS, to evade detection.

On the other hand, the MPRESS packing is well known for its ability to hardly compress malware to make it harder to be detected and significantly reduce its size. It achieves this by replacing large sections of the original code with compressed and encrypted versions. The algorithm then adds the stub to compress/decompress the original code at runtime, making it difficult for static analysis tools to detect the true purpose of the malware.

MPRESS uses a variety of advanced techniques to make it difficult for analysts to reverse engineer the packed code. It uses a custom stub to pack and unpack the code, which makes it difficult for anti-virus software to detect and unpack the malware. The algorithm also uses anti-debugging techniques to prevent analysts from debugging the packed code or altering its execution in any way.

One of the main advantages of MPRESS is its speed and efficiency it has. The packing process is relatively fast, and the resulting packed file is highly compressed, which reduces the overall size of the file. This makes it easier for malware authors to distribute the packed file by being stealthier.

It is common among malware writers to create their own custom versions of packers. UPX is an open-source tool, which means that the source code is accessible and can modify it as needed. However, creating a custom version of a packer is not a trivial task and requires a good understanding of assembly language, binary file formats, and compression algorithms. In the case of MPESS is even harder as it is a proprietary packer and creating a custom version requires to reverse engineer it. By doing so malware writers will go against its license agreement.

Although a custom packer may not be as effective as the original packer in terms of compression ratio and speed, the main goal of malware writers is to make the malware less detectable possible and harder to be analyzed. Malware writers often introduce additional peace of code in order to change the malware signature to evade security systems or to deviate from the analyst analysis by introducing anti-analysis techniques.

5.3 - Import Address Table

It is very important to introduce the concept of Import Address Table (IAT) to understand all the steps of malware analysis later on. In particular to understand the process of extracting and creating the PE file containing the unpacked code from a packed file.

The Import Address Table (IAT) is a crucial component in PE (Portable Executable) files used on Windows-based systems. The IAT is responsible for storing pointers to functions that are imported from other DLLs (Dynamic-Link Libraries), which is an essential part of allowing a program to call functions that are not part of its own code. This is a fundamental aspect of modular and maintainable programming.

During the resolution stage of a typical Windows PE executable, the header of the executable contains essential metadata that informs the operating system about the symbols from external libraries that the executable relies upon. When the operating system's loader processes the executable, it loads the necessary libraries into memory if they are not already loaded. The loader then assigns the addresses of the imported symbols to specific memory locations as specified by the metadata within the executable.

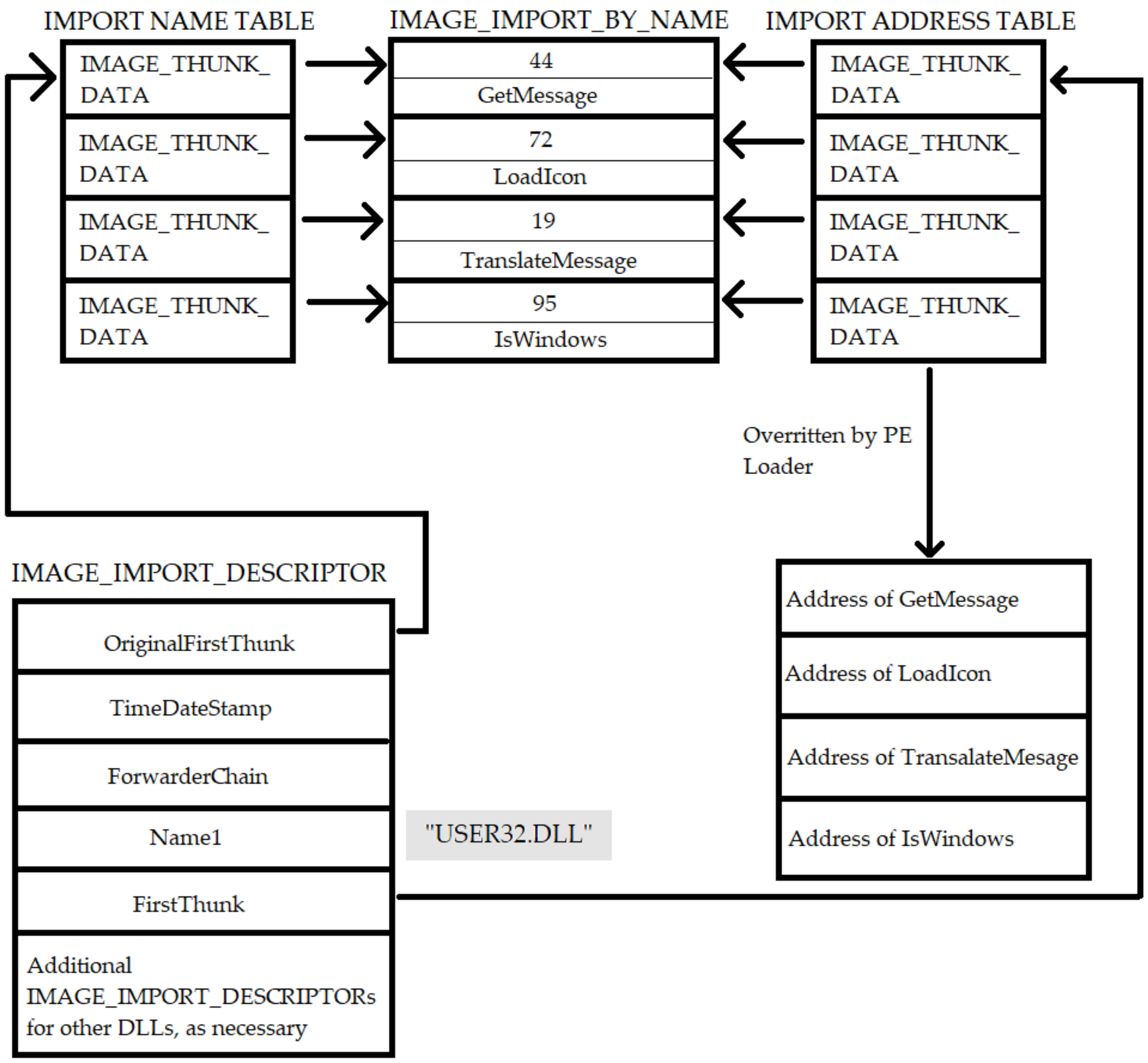
To clarify the image below, the IMAGE_IMPORT_BY_NAME is a structure within the Import Address Table (IAT). It is used to store the import information for a specific function, including the function name and its corresponding hint value. The IMAGE_IMPORT_BY_NAME structure is employed when the import is performed using the name of the function, rather than its ordinal value, which are the two methods to import functions.

The Import Name Table (INT) stores the names of the imported functions from external DLLs. The Import Address Table (IAT) is responsible for holding the addresses of the imported functions after they have been resolved by the loader. It is an array of function pointers located within the PE file's data section. The IAT is populated by the loader with the actual addresses of the imported functions.

The purpose of the IMAGE_IMPORT_BY_NAME structure is to provide the necessary information for importing a function by its name. It contains the name of the function and its corresponding hint value, which is an index into the export name table of the DLL. The

IMAGE_IMPORT_BY_NAME structure is used when the loader needs to resolve an imported function by name rather than by ordinal.

In the context of the IMAGE_IMPORT_BY_NAME structure in the Portable Executable (PE) format, the Hint value is an optional field that provides an additional hint to help identify the imported function. The Hint value is typically an index or position of the function within the export table of the DLL from which it is being imported. It allows for faster and more efficient resolution of imported functions, especially when the import code knows the specific Hint value to look for.



However, in the case of packed executables created by packers, this metadata is often modified. Packers take on the responsibility of performing the resolution stage themselves instead of relying on the operating system's loader.

Packers typically employ techniques to handle the resolution of symbols and imported libraries internally. They analyze the packed executable, identify the required symbols, and dynamically resolve their addresses within the packed code. This allows the packed executable to function without relying on the standard resolution mechanism provided by the operating system's loader.
By bypassing the standard resolution stage, packers can modify the structure of the executable and hide important information about dependencies, making it more challenging for reverse engineers and analysis tools to understand the executable's inner workings.

It's worth noting that the resolution stage performed by packers may vary depending on the specific packing technique used and the goals of the packer. Some packers may prioritize minimizing the size of the executable and focus on simple resolution mechanisms, while others may employ more advanced techniques to evade detection and analysis.

The IAT reconstruction is a feature of tools like Scylla [6], typically involves scanning the binary of a loaded module (a process's executable image in memory) and identifying patterns that match the standard structure of an IAT. In other words, it's looking for arrays of memory addresses that point to valid locations within loaded DLLs. This process is often needed when analyzing malware or reversing a packed executable, where the IAT may be altered.

Once potential IAT locations are identified, the tool can often rebuild a valid IAT, allowing the binary to be dumped from memory and run as a standalone executable. This can be incredibly useful in malware analysis and similar reverse engineering tasks.

5.3 - Environment Setup

For this setup, I opted to use a virtualized environment, specifically VMware Workstation[2] with Windows 10 Operating System. I chose Windows as several malware samples are readily available, and it is the most widely used and targeted operating system. Additionally, some critical tools are only accessible on Windows. While compatibility layers can allow these tools to be used on other operating systems such as Linux, their proper execution cannot be guaranteed.

To ensure efficient performance, I configured the Operating System with the minimum requirements of 4GB of RAM, 60GB of storage, and 4 cores, as the tools utilized are lightweight and do not require significant computational power. I made certain modifications within the Operating System such as disabling the Windows 10 Virus and Threat Protection. This was necessary to prevent Windows 10 from detecting the malware samples as malicious files and automatically deleting them.

Some other steps can be performed according to the needs:

To Disable Windows Defender

1. Open Windows Security (type Windows Security in the search box)
2. Virus & threat protection → Virus & threat protection settings → Manage settings.
3. Switch Tamper Protection to Off

To permanently disable Real Time Protection

1. Open Local Group Policy Editor (type gpedit in the search box)
2. Computer Configuration → Administrative Templates → Windows Components → Microsoft Defender Antivirus → Real-time Protection
3. Enable Turn off real-time protection.
4. Reboot

To permanently disable Microsoft Defender:

1. Open Local Group Policy Editor (type gpedit in the search box)
2. Computer Configuration → Administrative Templates → Windows Components → Microsoft Defender Antivirus
3. Enable Turn off Microsoft Defender Antivirus
4. Reboot

Disabling or removing the virtual machine's network card is a crucial step to prevent malware from communicating externally and causing unintentional malicious traffic. It is also important to keep the virtualization software, in my case VMware, up to date with the latest version to avoid exploitation of known vulnerabilities that could potentially allow the malware to escape the virtual environment and infect the host machine.

5.4 - UPX

For this demonstration of unpacking malware packed with UPX, I will be using the following tools:

- PEStudio [3]
- Detect It Easy [4]
- IDA (Free) [5]
- Scylla [6]

Before diving into how and why these tools are used, let's first introduce them and their purpose. PEStudio [3] provides information about malware and potential behavior, allowing us to gain an idea of what it might be. The tool also indicates whether a file is malicious or not, drawing from different sources. For example, it may analyze strings within the file that indicate modifications to registry keys, libraries, or imports that usual software do not contain or often used by malware. However, this can also result in many false positives. Additionally, PEStudio [3] provides an important indication of the presence of packers through the entropy value, which gives an indication of the file's level of compression.

DetectItEasy is a software tool that provides a straightforward approach to detecting the presence of packers. Its capabilities are similar to those of PEStudio [3] in terms of packer detection and using multiple tools can be useful in confirming the presence of packing.

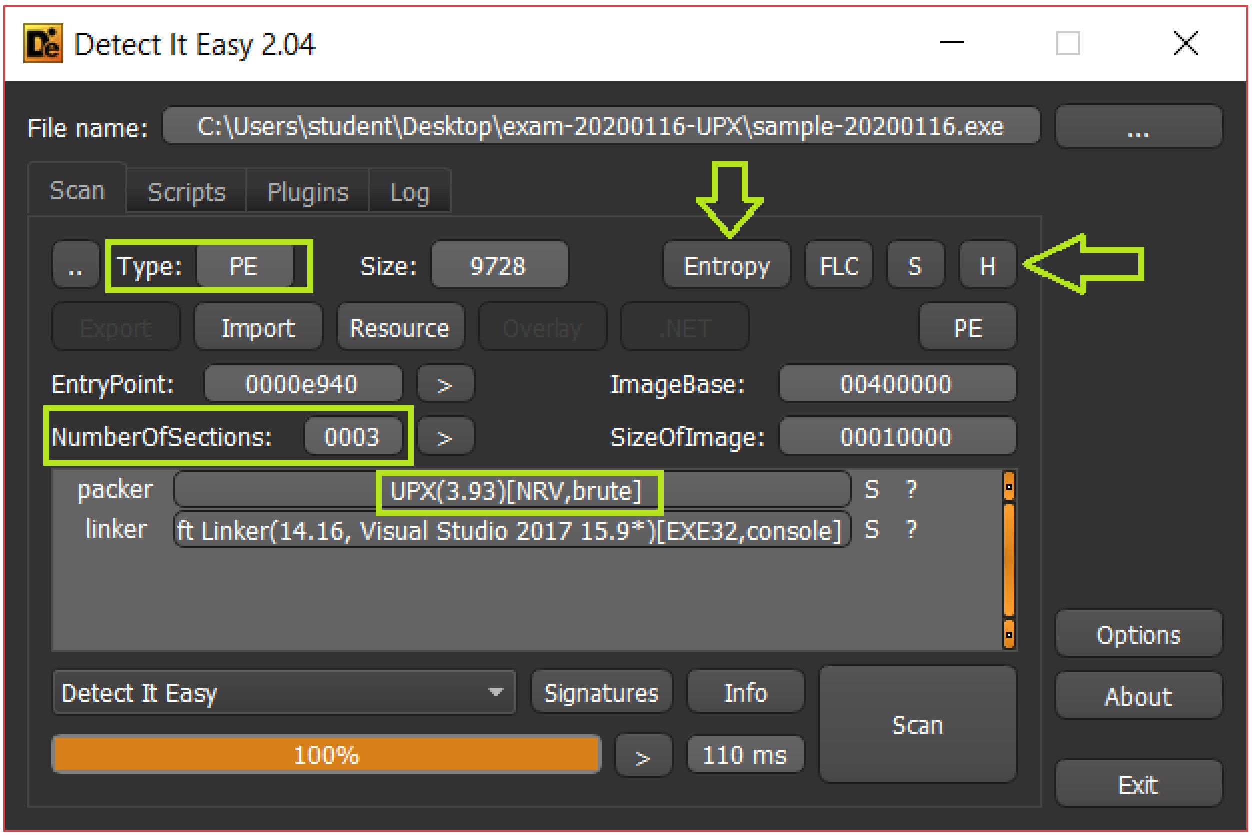
IDA (Free) [5], a free version of IDA [5] with limited functionality is mainly used to analyze the code inside the malware and to evaluate the malware behavior. In this case, we will use IDA [5] to find the tail jump (as described in “Dynamic Analysis” chapter) in the case of UPX and the Original Entry Point (OEP) in the case of MPRESS.

Finally, we can utilize Scylla [6] to reconstruct the Import Address Table (IAT) of the unpacked code, dump the unpacked code from memory onto the disk and create a new executable file that includes only the original code.

Packing Indicators

In this section, we will attempt to collect all the evidence and indications of packing. This is the initial step in confirming the presence of packers and, if possible, identifying the packer's name. To accomplish this, we will utilize both PEStudio [3] and Detect it Easy [4] (DIE) and compared their results to ensure their similarity. It's important to note that despite performing the same task of detecting packer presence, their outcomes may differ slightly, potentially resulting in minor discrepancies in the result.

The first step to perform is to load the binary with Detect It Easy [4] and analyze the information it provided.


Detect It Easy 2.04

File name: C:\Users\student\Desktop\exam-20200116-UPX\sample-20200116.exe

Type: PE
Size: 9728
EntryPoint: 0000e940
ImageBase: 00400000
NumberOfSections: 0003
SizeOfImage: 00010000

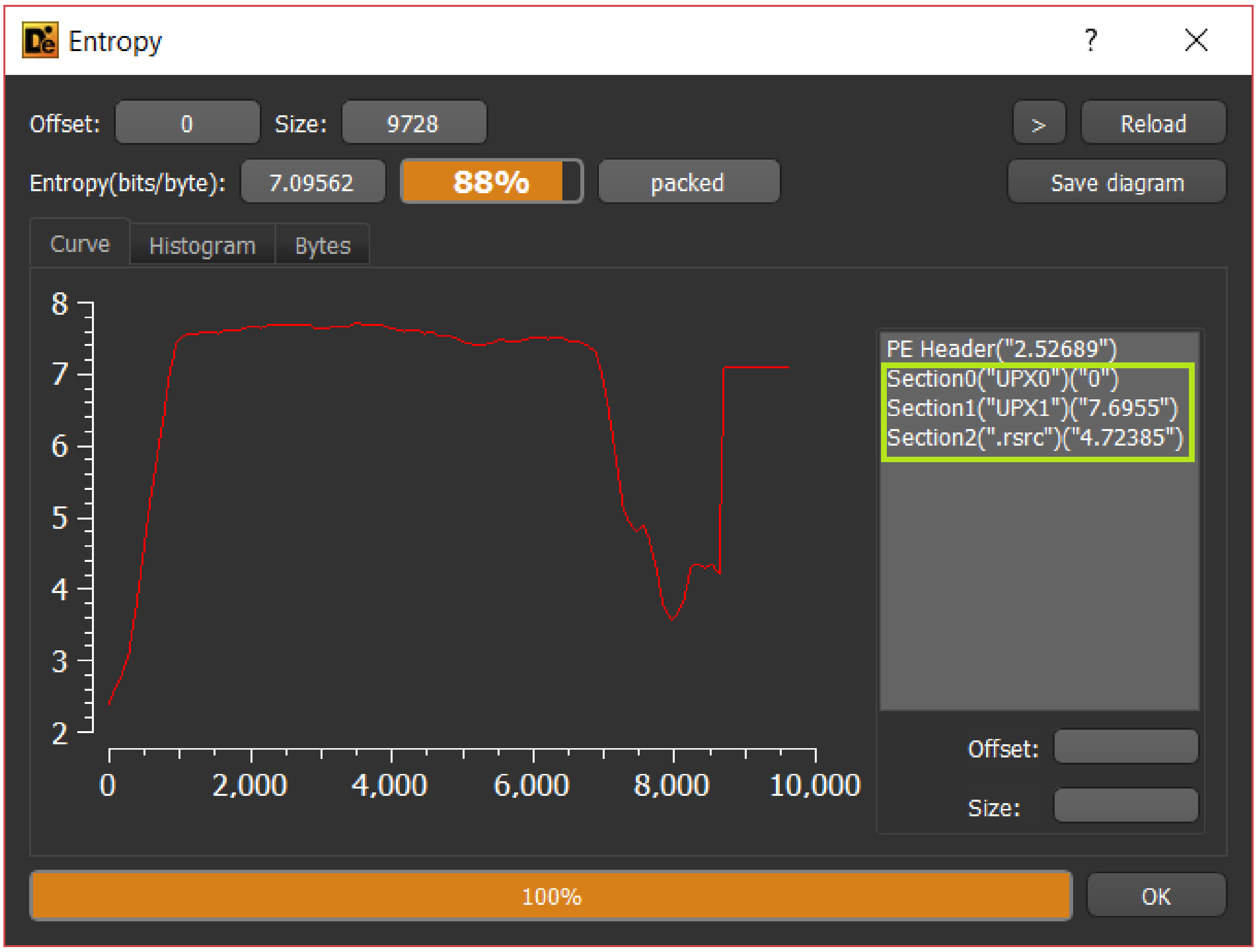
packer UPX(3.93)[NRV,brute] S ?
linker ft Linker(14.16, Visual Studio 2017 15.9*)[EXE32,console] S ?

Detect It Easy
100%
110 ms

After loading the binary with Detect It Easy [4], it reveals that it was indeed packed with UPX and even provided version number 3.93. This information is crucial for the subsequent unpacking process.

In addition to the packing information, Detect It Easy [4] also provide us with the number of sections present in the binary. This is important as it allow us to cross-check the section count with that reported by other tools. Discrepancies between section counts could indicate malicious activity such as section padding or code injection.

Another important piece of information that we obtained from Detect It Easy [4] was the entropy value of the binary. Entropy is a measure of the randomness of the data in the binary. In the context of malware analysis, it can help indicate the presence of obfuscation or encryption techniques used by the malware author to evade detection.



Offset: 0 Size: 9728
Entropy(bits/byte): 7.09562 88% packed

PE Header("2.52689")
Section0("UPX0")("0")
Section1("UPX1")("7.6955")
Section2(".rsrc")("4.72385")

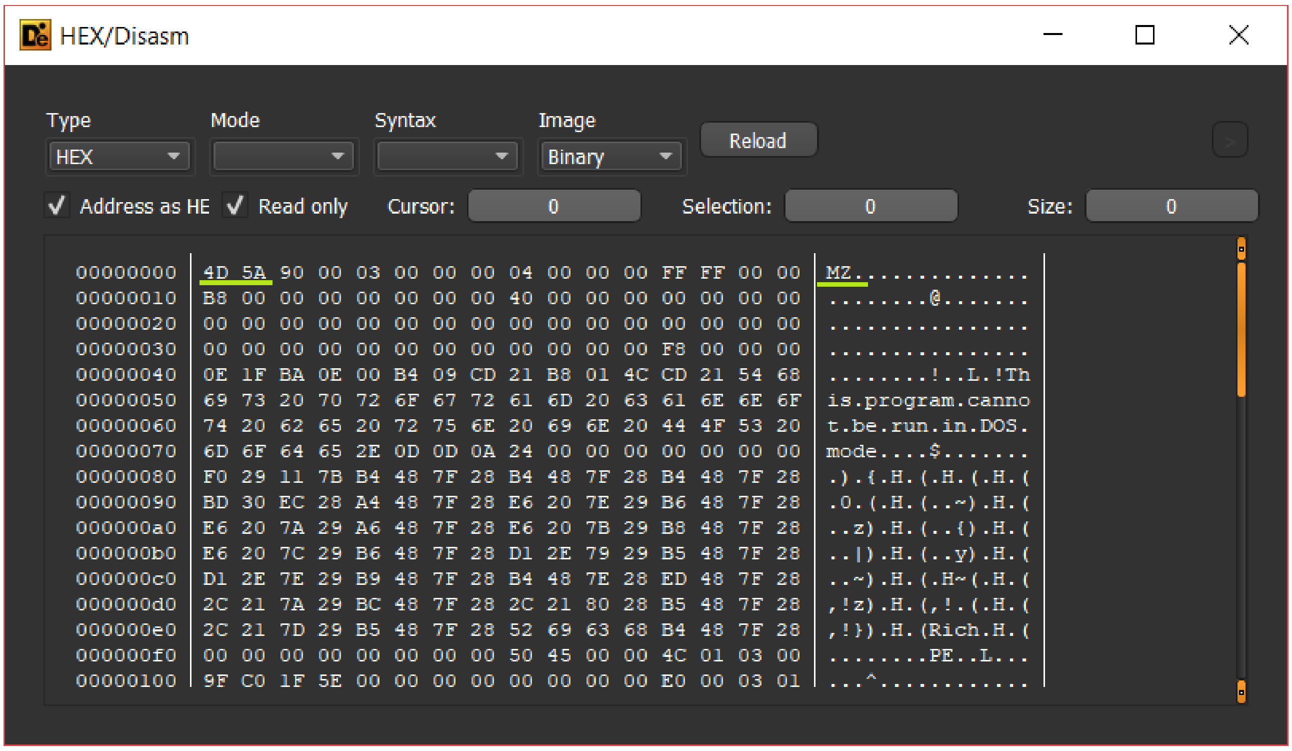
Offset:
Size:

100%

After loading the binary into Detect It Easy [4], we can further check if it was packed or not. High entropy in a binary is usually a good indication of packing, but sometimes it can show that the binary is not packed even though it has a high level of entropy. For this sample I have a value of 7.09, the value is very high, and it is a clear indication of packing.

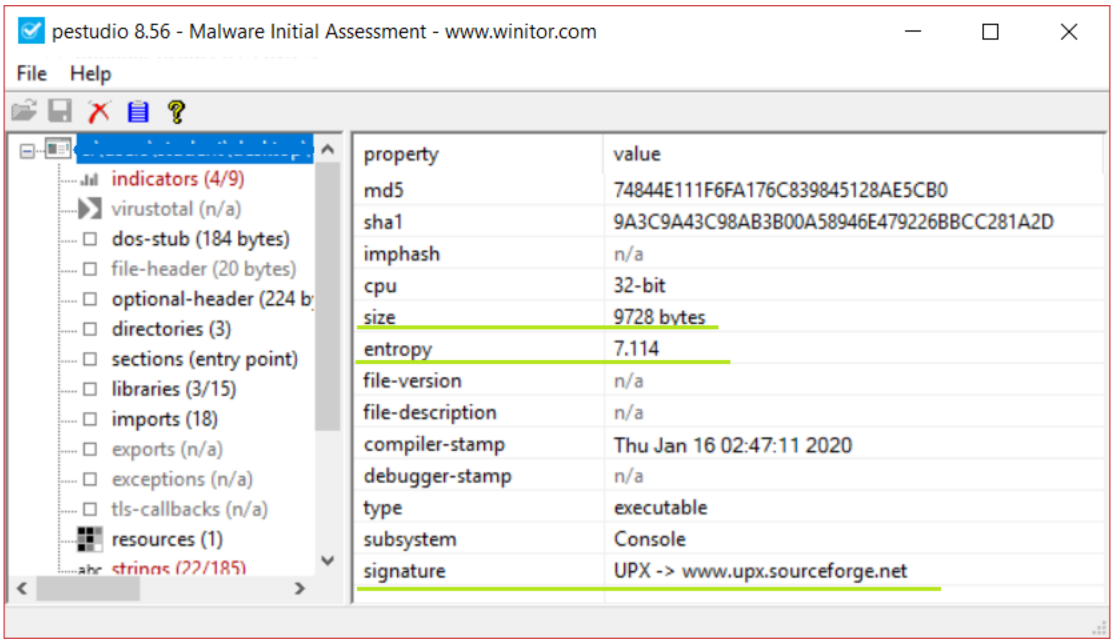
We also need to check the type of binary, which we can do by checking the H-hexadecimal key in the main window of Detect It Easy [4]. Although we already know it was a PE (Portable Executable), we just want to be sure.

The binary had the character "MZ", as shown below, which indicates that it is a PE. However, it is possible for attackers to change these characters to disguise the true type of the binary. Therefore, we also check the number of next bytes to verify if it was really a PE or not.



```
00000000  4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00  MZ..............
00000010  B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00  ........@.......
00000020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000030  00 00 00 00 00 00 00 00 00 00 00 00 F8 00 00 00  ................
00000040  0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68  ........!..L.!Th
00000050  69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F  is.program.canno
00000060  74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20  t.be.run.in.DOS.
00000070  6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00  mode....$.......
00000080  F0 29 11 7B B4 48 7F 28 B4 48 7F 28 B4 48 7F 28  .).{.H.(.H.(.H.(
00000090  BD 30 EC 28 A4 48 7F 28 E6 20 7E 29 B6 48 7F 28  .0.(.H.(..~).H.(
000000a0  E6 20 7A 29 A6 48 7F 28 E6 20 7B 29 B8 48 7F 28  ..z).H.(..{).H.(
000000b0  E6 20 7C 29 B6 48 7F 28 D1 2E 79 29 B5 48 7F 28  ..|).H.(..y).H.(
000000c0  D1 2E 7E 29 B9 48 7F 28 B4 48 7E 28 ED 48 7F 28  ..~).H.(.H~(.H.(
000000d0  2C 21 7A 29 BC 48 7F 28 2C 21 80 28 B5 48 7F 28  ,!z).H.(,!.(.H.(
000000e0  2C 21 7D 29 B5 48 7F 28 52 69 63 68 B4 48 7F 28  ,!}).H.(Rich.H.(
000000f0  00 00 00 00 00 00 00 00 50 45 00 00 4C 01 03 00  ........PE..L...
00000100  9F C0 1F 5E 00 00 00 00 00 00 00 00 E0 00 03 01  ...^............
```

At this point, we can used PEStudio [3] to gather more information about the binary and to confirm whether it is packed or not and if the results from both tools are similar. The first thing we must check is the size of the binary to get an idea of what we are dealing with. It is unlikely to have few imports if the file is very large, while for a small file, it is plausible but not certain. Therefore, it is best not to trust. For this sample the size 9728 bytes is in the norm, and the number of imports (18 imports) are not adequate. In general, programs of small to medium complexity already include a large number of functions imported from external libraries, a number that easily can cross 50 imports.


pestudio 8.56 - Malware Initial Assessment - www.winitor.com

File Help

- [redacted]
  - indicators (4/9)
  - virustotal (n/a)
  - dos-stub (184 bytes)
  - file-header (20 bytes)
  - optional-header (224 b
  - directories (3)
  - sections (entry point)
  - libraries (3/15)
  - imports (18)
  - exports (n/a)
  - exceptions (n/a)
  - tls-callbacks (n/a)
  - resources (1)
  - strings (22/185)

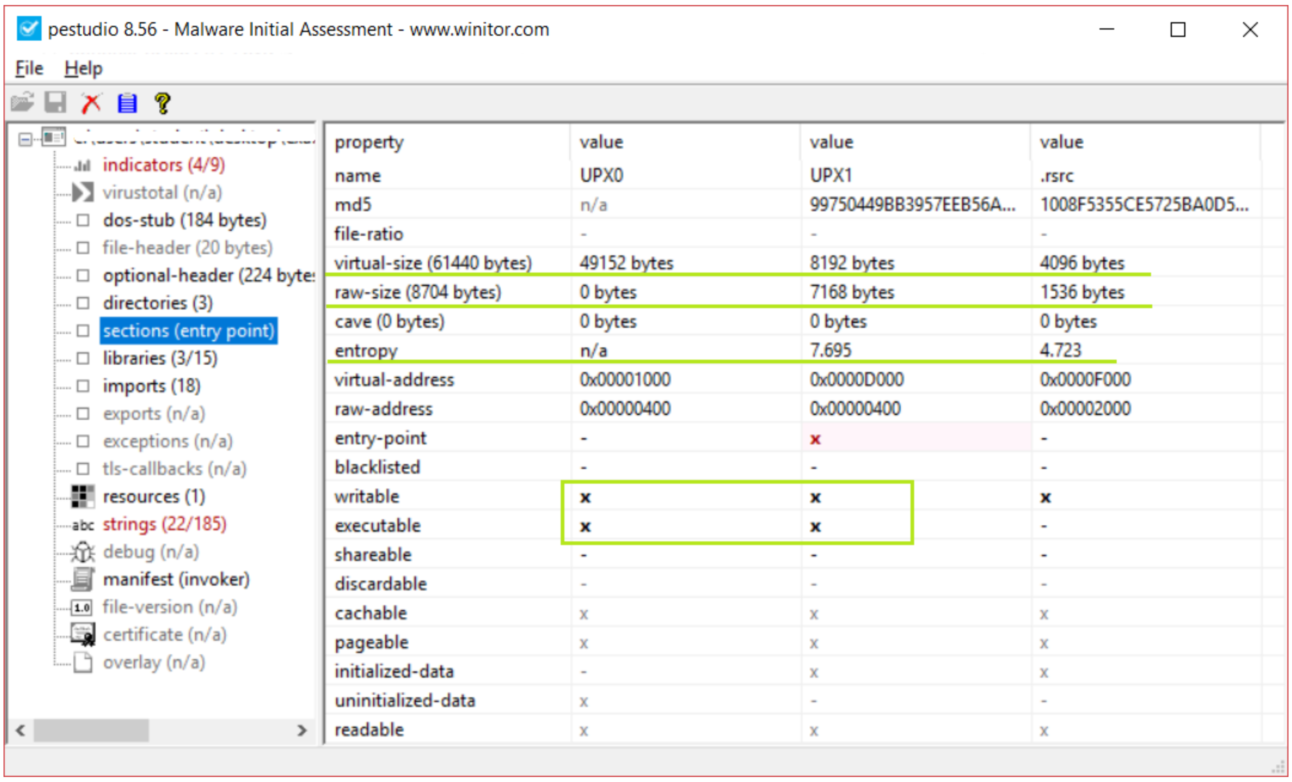
property	value
md5	74844E111F6FA176C839845128AE5CB0
sha1	9A3C9A43C98AB3B00A58946E479226BBCC281A2D
imphash	n/a
cpu	32-bit
size	9728 bytes
entropy	7.114
file-version	n/a
file-description	n/a
compiler-stamp	Thu Jan 16 02:47:11 2020
debugger-stamp	n/a
type	executable
subsystem	Console
signature	UPX -> www.upx.sourceforge.net

Here, in addition to the file size, we also see the entropy (7,114) which is very high, close to Detect It Easy's [4] value and therefore it seems that it is packed. The Signature is further confirming that it is packed with UPX.

The next indicator can be found in the sections, in particular "Virtual-Size" and "Raw-Size" of the sections. If at last in one of them, the difference is so high it's a good indicator of packing.

If at least one section has a very high entropy, then it is another good indicator of packing (MAX-Entropy: 8), also if at least one section has WX (Writable and Executable) permission is also an indicator of packing.

In the following image, it is evident that the UPX0 section exhibits significant differences in both virtual size and raw size. Furthermore, the UPX1 section displays a notably high entropy value of 7168. These observations provide further confirmation of the packer's presence within the malware.



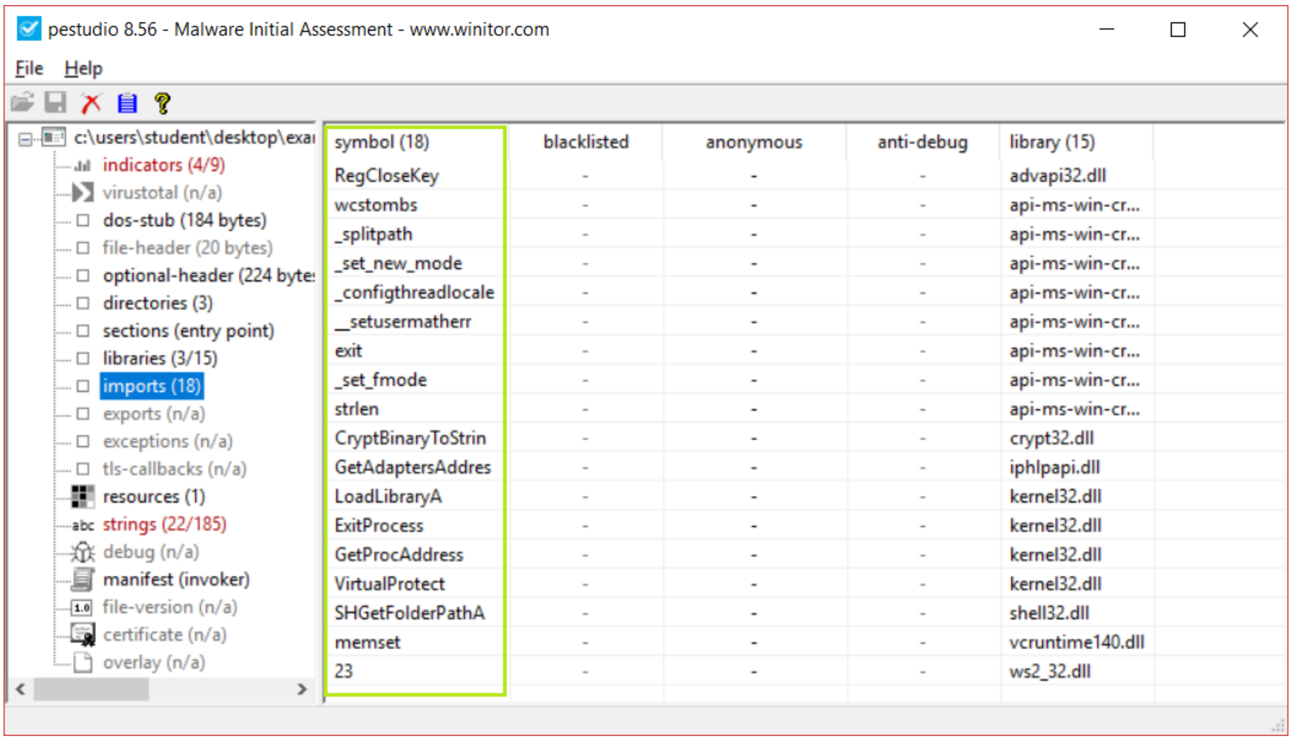
pestudio 8.56 - Malware Initial Assessment - www.winitor.com

File Help

- [illegible]
  - indicators (4/9)
  - virustotal (n/a)
  - dos-stub (184 bytes)
  - file-header (20 bytes)
  - optional-header (224 bytes)
  - directories (3)
  - sections (entry point)
  - libraries (3/15)
  - imports (18)
  - exports (n/a)
  - exceptions (n/a)
  - tls-callbacks (n/a)
  - resources (1)
  - strings (22/185)
  - debug (n/a)
  - manifest (invoker)
  - file-version (n/a)
  - certificate (n/a)
  - overlay (n/a)

property	value	value	value
name	UPX0	UPX1	.rsrc
md5	n/a	99750449BB3957EEB56A...	1008F5355CE5725BA0D5...
file-ratio	-	-	-
virtual-size (61440 bytes)	49152 bytes	8192 bytes	4096 bytes
raw-size (8704 bytes)	0 bytes	7168 bytes	1536 bytes
cave (0 bytes)	0 bytes	0 bytes	0 bytes
entropy	n/a	7.695	4.723
virtual-address	0x00001000	0x0000D000	0x0000F000
raw-address	0x00000400	0x00000400	0x00002000
entry-point	-	x	-
blacklisted	-	-	-
writable	x	x	x
executable	x	x	-
shareable	-	-	-
discardable	-	-	-
cachable	x	x	x
pageable	x	x	x
initialized-data	-	x	x
uninitialized-data	x	-	-
readable	x	x	x

Now we can check the imports, if they are so few it's a good indicator of packing, if "GetProcess" and "LoadLibraryA" are the only imports shown in the PEStudio [3] then we can be sure that the binary is packed.



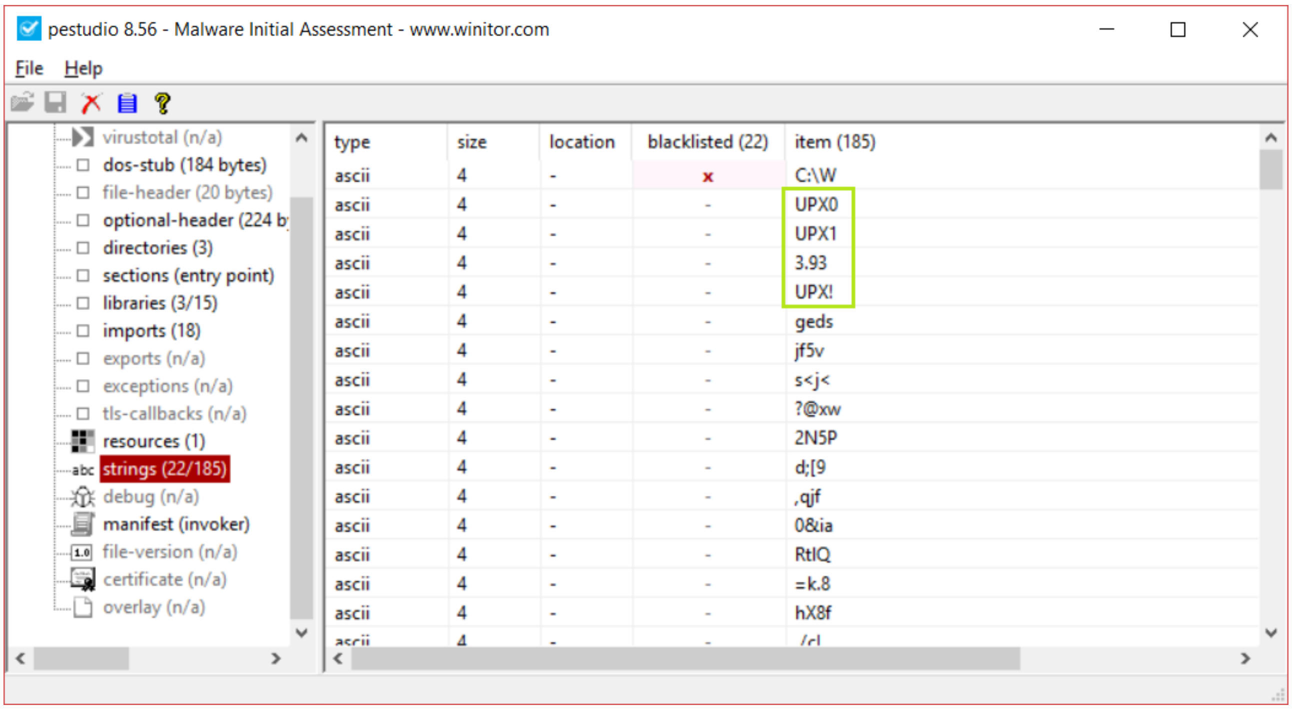
pestudio 8.56 - Malware Initial Assessment - www.winitor.com

File Help

- c:\users\student\desktop\exa
  - indicators (4/9)
  - virustotal (n/a)
  - dos-stub (184 bytes)
  - file-header (20 bytes)
  - optional-header (224 bytes)
  - directories (3)
  - sections (entry point)
  - libraries (3/15)
  - imports (18)
  - exports (n/a)
  - exceptions (n/a)
  - tls-callbacks (n/a)
  - resources (1)
  - strings (22/185)
  - debug (n/a)
  - manifest (invoker)
  - file-version (n/a)
  - certificate (n/a)
  - overlay (n/a)

symbol (18)	blacklisted	anonymous	anti-debug	library (15)
RegCloseKey	-	-	-	advapi32.dll
wcstombs	-	-	-	api-ms-win-cr...
_splitpath	-	-	-	api-ms-win-cr...
_set_new_mode	-	-	-	api-ms-win-cr...
_configthreadlocale	-	-	-	api-ms-win-cr...
__setusermatherr	-	-	-	api-ms-win-cr...
exit	-	-	-	api-ms-win-cr...
_set_fmode	-	-	-	api-ms-win-cr...
strlen	-	-	-	api-ms-win-cr...
CryptBinaryToStrin	-	-	-	crypt32.dll
GetAdaptersAddres	-	-	-	iphlpapi.dll
LoadLibraryA	-	-	-	kernel32.dll
ExitProcess	-	-	-	kernel32.dll
GetProcAddress	-	-	-	kernel32.dll
VirtualProtect	-	-	-	kernel32.dll
SHGetFolderPathA	-	-	-	shell32.dll
memset	-	-	-	vcruntime140.dll
23	-	-	-	ws2_32.dll

In this particular case, we have very few imports but there may be cases where we have only the previously mentioned imports. Finally, in the list of strings, we search for the presence of the packer's name. And to make the search easy we can reorder the list in ascending order as the packer names are very short.



type	size	location	blacklisted (22)	item (185)
ascii	4	-	x	C:\W
ascii	4	-	-	UPX0
ascii	4	-	-	UPX1
ascii	4	-	-	3.93
ascii	4	-	-	UPX!
ascii	4	-	-	geds
ascii	4	-	-	jf5v
ascii	4	-	-	s<j<
ascii	4	-	-	?@xw
ascii	4	-	-	2N5P
ascii	4	-	-	d;[9
ascii	4	-	-	,qjf
ascii	4	-	-	0&ia
ascii	4	-	-	RtlQ
ascii	4	-	-	=k.8
ascii	4	-	-	hX8f

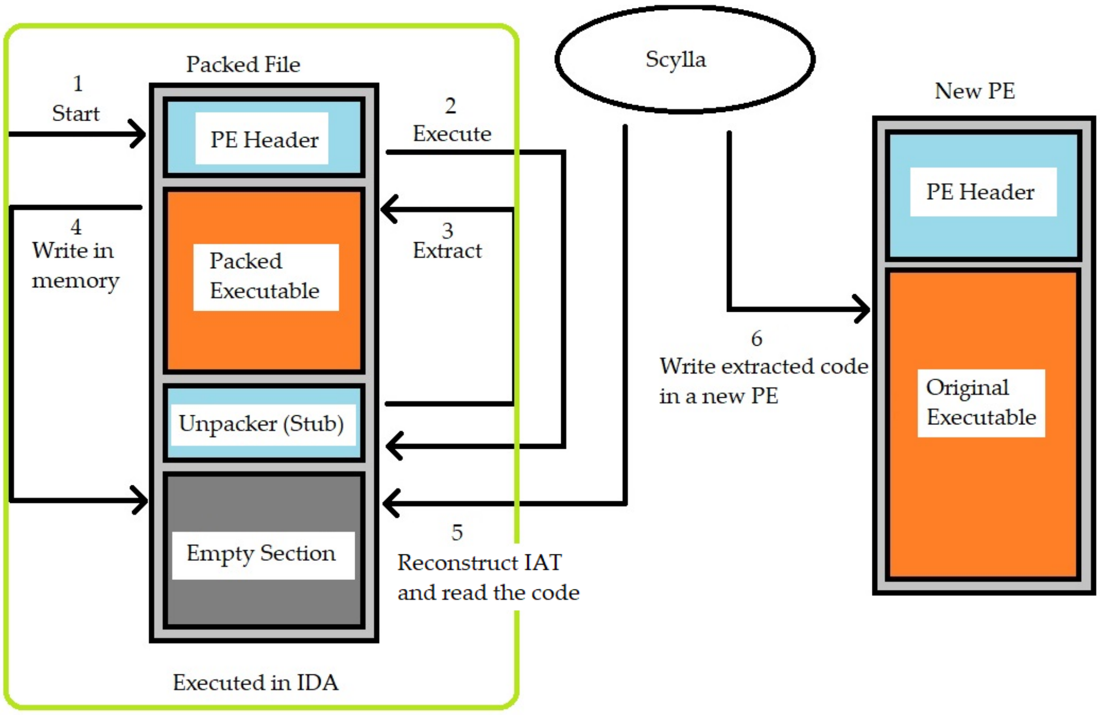
It is important to note that while we have confirmed the presence of the UPX packer at the beginning, the subsequent checks were conducted for the sake of completeness. While identifying the packer and its name was straightforward for this sample, more sophisticated malware samples may present more significant challenges, requiring the identification of additional indicators. Malware authors often employ tactics such as altering section names and signatures to make detecting the packer more difficult. The steps outlined in this analysis apply to any malware; however, they may not always suffice, and in some cases, code analysis using IDA [5] may be necessary to determine the specific packer employed, particularly in the case of custom packers.

Now that we made sure that the binary was packed, we will outline the process of how to unpack it, and the two cases where it was packed with MPRESS and the case where it was packed with UPX.

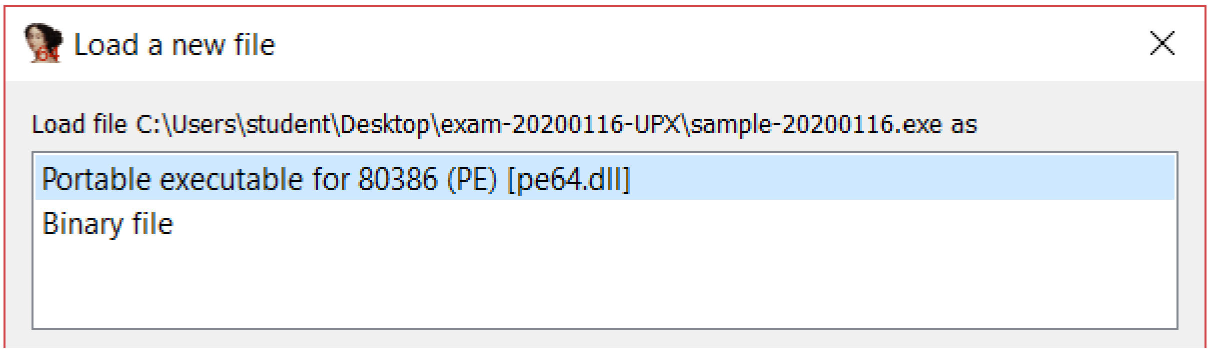
For the first part, we used a packed binary with UPX, but the results of PEStudio [3] and Detect It Easy [4] would have been almost identical if the file had been packed with MPRESS.

Manual Unpacking of UPX

In this section, the code will be analyzed using IDA [5] to identify the packed code and extract it into a new Portable Executable (PE) file, aiming to recreate the original file. It's important to note that the resulting file is typically very similar but rarely identical to the original file, despite executing in the same manner. This process is essentially the reverse of what was described in the "How Packers Work" section.



To perform the unpacking, we use IDA [5]. Any version is fine, but in our case, we ca use IDA Free [5] and loaded the binary as a PE file.

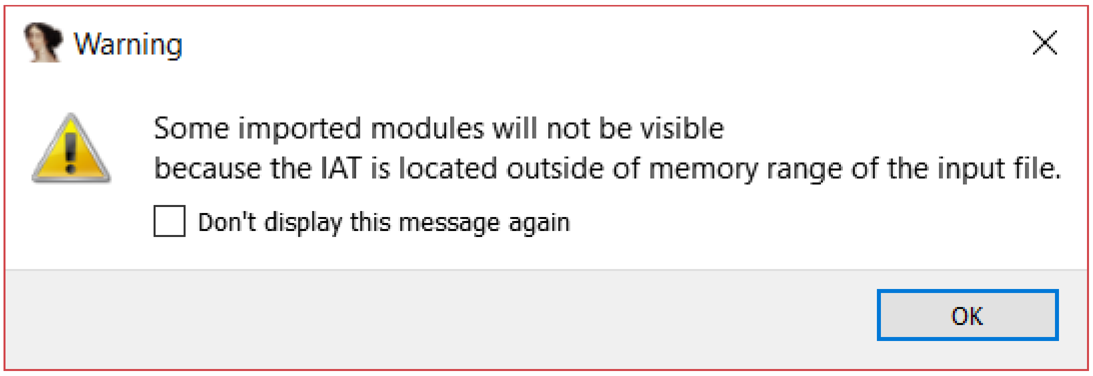


Load a new file

Load file C:\Users\student\Desktop\exam-20200116-UPX\sample-20200116.exe as

Portable executable for 80386 (PE) [pe64.dll]
Binary file

After loading the binary in IDA [5], it warns us that certain imports are not visible and that the IAT (Import Address Table) is located outside of the memory range. This was a clear indication that the binary was packed.

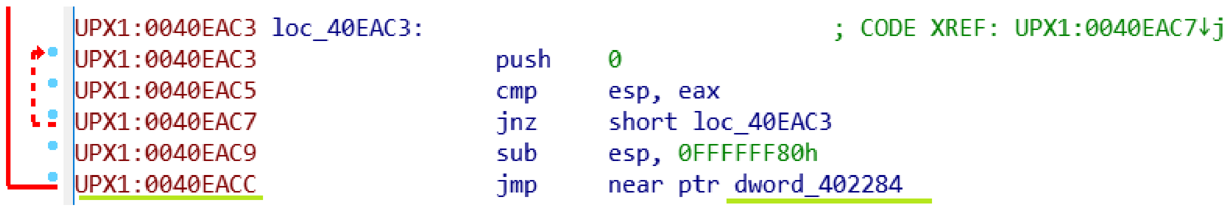


Warning

Some imported modules will not be visible
because the IAT is located outside of memory range of the input file.

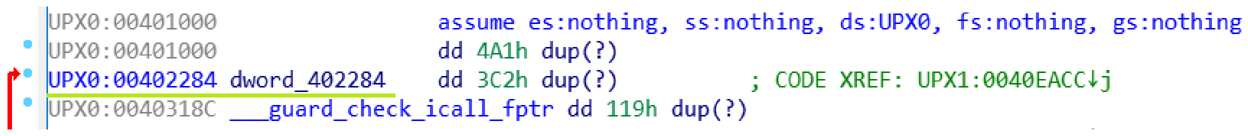
Don't display this message again

In order to unpack UPX, we have to search for a specific type of jump called a "tail jump". This term refers to the jump's typical location at the end of the program, but this is not always the case. The "tail jump" had a particular characteristic in that it traversed from one program section to another, making it a long jump.



```
UPX1:0040EAC3 loc_40EAC3:                             ; CODE XREF: UPX1:0040EAC7↓j
UPX1:0040EAC3                 push    0
UPX1:0040EAC5                 cmp     esp, eax
UPX1:0040EAC7                 jnz     short loc_40EAC3
UPX1:0040EAC9                 sub     esp, 0FFFFFF80h
UPX1:0040EACC                 jmp     near ptr dword_402284
```

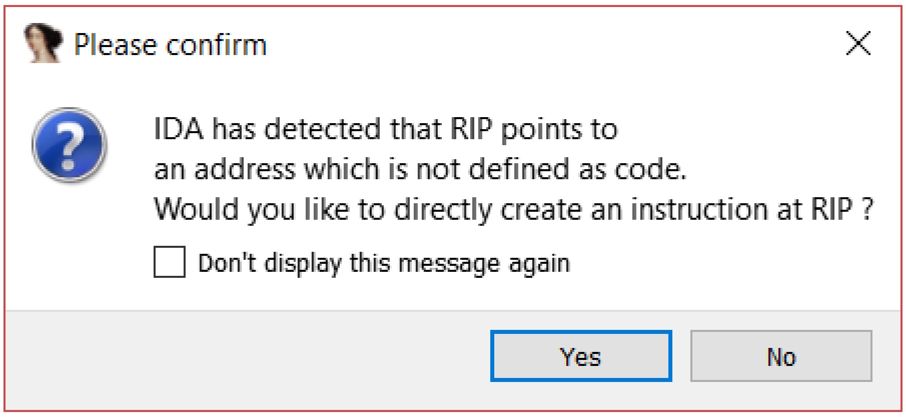
Here we have a jump in the UPX1 section which went to another section and jumped to address 402284 which is UPX0.



```
UPX0:00401000                 assume es:nothing, ss:nothing, ds:UPX0, fs:nothing, gs:nothing
UPX0:00401000                 dd 4A1h dup(?)
UPX0:00402284 dword_402284    dd 3C2h dup(?)          ; CODE XREF: UPX1:0040EACC↓j
UPX0:0040318C ___guard_check_icall_fptr dd 119h dup(?)
```

Upon examining the code, we can observe that the address 402284 belongs to the UPX0 section and there are no similar jumps elsewhere, indicating that we have successfully identified the tail jump.

Next, we set a breakpoint at address 402284 and execute the binary and the resulting message serves as further confirmation that the binary has been packed.

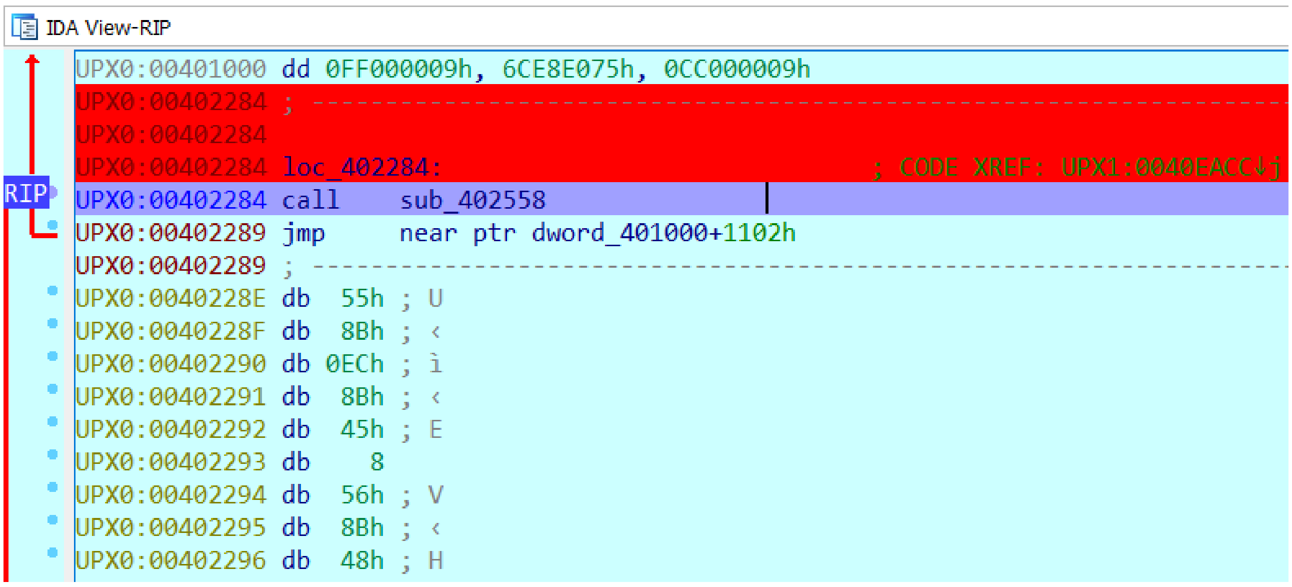


Please confirm

IDA has detected that RIP points to
an address which is not defined as code.
Would you like to directly create an instruction at RIP ?

☐ Don't display this message again

After pressing "Yes" and continuing the execution, the code has been changed, and I could see the hidden portion of the code previously packed.



IDA View-RIP

```
UPX0:00401000 dd 0FF000009h, 6CE8E075h, 0CC000009h
UPX0:00402284 ; ---------------------------------------------------------------
UPX0:00402284
UPX0:00402284 loc_402284:                               ; CODE XREF: UPX1:0040EACC↓j
UPX0:00402284 call    sub_402558
UPX0:00402289 jmp     near ptr dword_401000+1102h
UPX0:00402289 ; ---------------------------------------------------------------
UPX0:0040228E db  55h ; U
UPX0:0040228F db  8Bh ; ‹
UPX0:00402290 db 0ECh ; ì
UPX0:00402291 db  8Bh ; ‹
UPX0:00402292 db  45h ; E
UPX0:00402293 db    8
UPX0:00402294 db  56h ; V
UPX0:00402295 db  8Bh ; ‹
UPX0:00402296 db  48h ; H
```

We were at a "call" statement, which was the correct position for me to be in. At that point, we can use either the address of the jump instruction (402284) or the address (402558) as the OEP (Original Entry Point), and both addresses would have worked well.

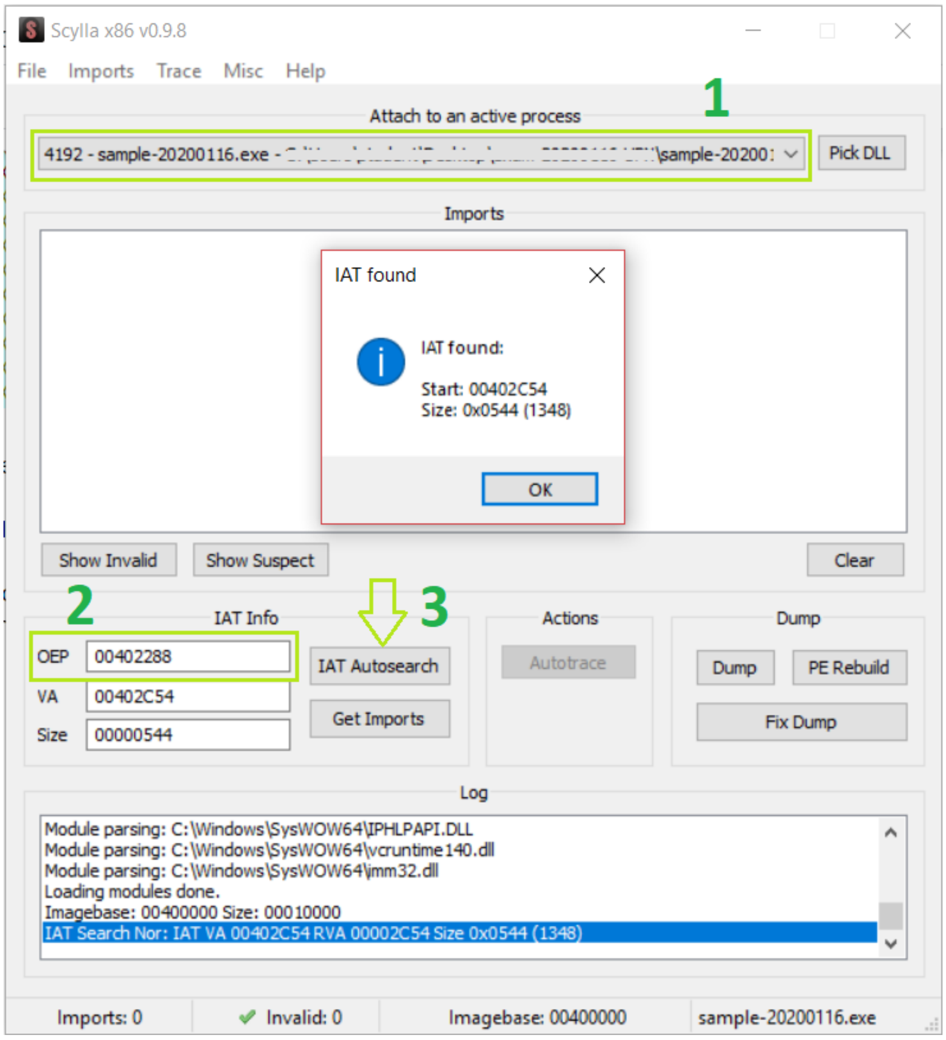
To unpack, we utilize Scylla [6] and use one of the two addresses as input in the OEP field. Throughout these processes, we must keep IDA [5] running in debug mode and avoided closing it.

The next steps to perform in Scylla [6] are the following:

1. Select the process of IDA [5] running the sample.
2. Insert in the OEP field the address found on IDA [5].
3. Click “IAT Autosearch”

These steps ensures that Scylla [6] can read from the process memory the unpacked code and extract it to recreate the executable file. In particular Scylla [6] starts to read from the OEP address and search for patters that match potential IAT (Import Address Table) to reconstruct it and create a new PE file with correct header and the unpacked code.

As we correctly provided the address Scylla [6] easily found the IAT. It is a good practice to restart Scylla [6] every time to ensure that it successfully finds the start address and size. This is necessary because Scylla [6] can be buggy and may not always provide accurate results.



4192 - sample-20200116.exe - ...\sample-20200:

IAT found

IAT found:

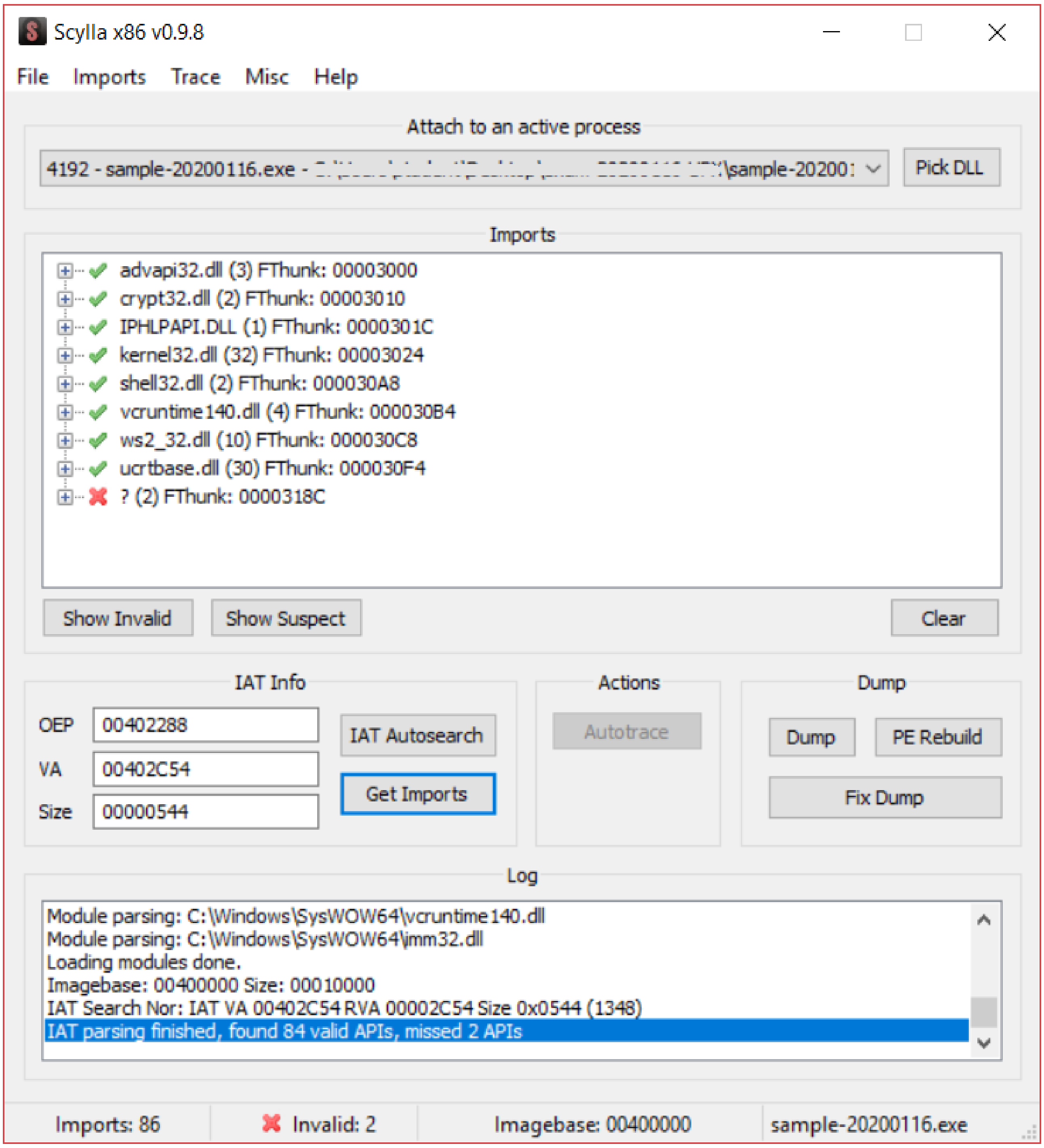
Start: 00402C54
Size: 0x0544 (1348)

OEP 00402288
VA 00402C54
Size 00000544

```
Module parsing: C:\Windows\SysWOW64\IPHLPAPI.DLL
Module parsing: C:\Windows\SysWOW64\vcruntime140.dll
Module parsing: C:\Windows\SysWOW64\imm32.dll
Loading modules done.
Imagebase: 00400000 Size: 00010000
IAT Search Nor: IAT VA 00402C54 RVA 00002C54 Size 0x0544 (1348)
```

Imports: 0 | ✓ Invalid: 0 | Imagebase: 00400000 | sample-20200116.exe

At that point, we are able to obtain the imports by simply clicking on "Get Imports".



4192 - sample-20200116.exe - [illegible]\sample-20200:

```
advapi32.dll (3) FThunk: 00003000
crypt32.dll (2) FThunk: 00003010
IPHLPAPI.DLL (1) FThunk: 0000301C
kernel32.dll (32) FThunk: 00003024
shell32.dll (2) FThunk: 000030A8
vcruntime140.dll (4) FThunk: 000030B4
ws2_32.dll (10) FThunk: 000030C8
ucrtbase.dll (30) FThunk: 000030F4
? (2) FThunk: 0000318C
```

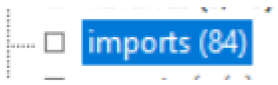
OEP 00402288
VA 00402C54
Size 00000544

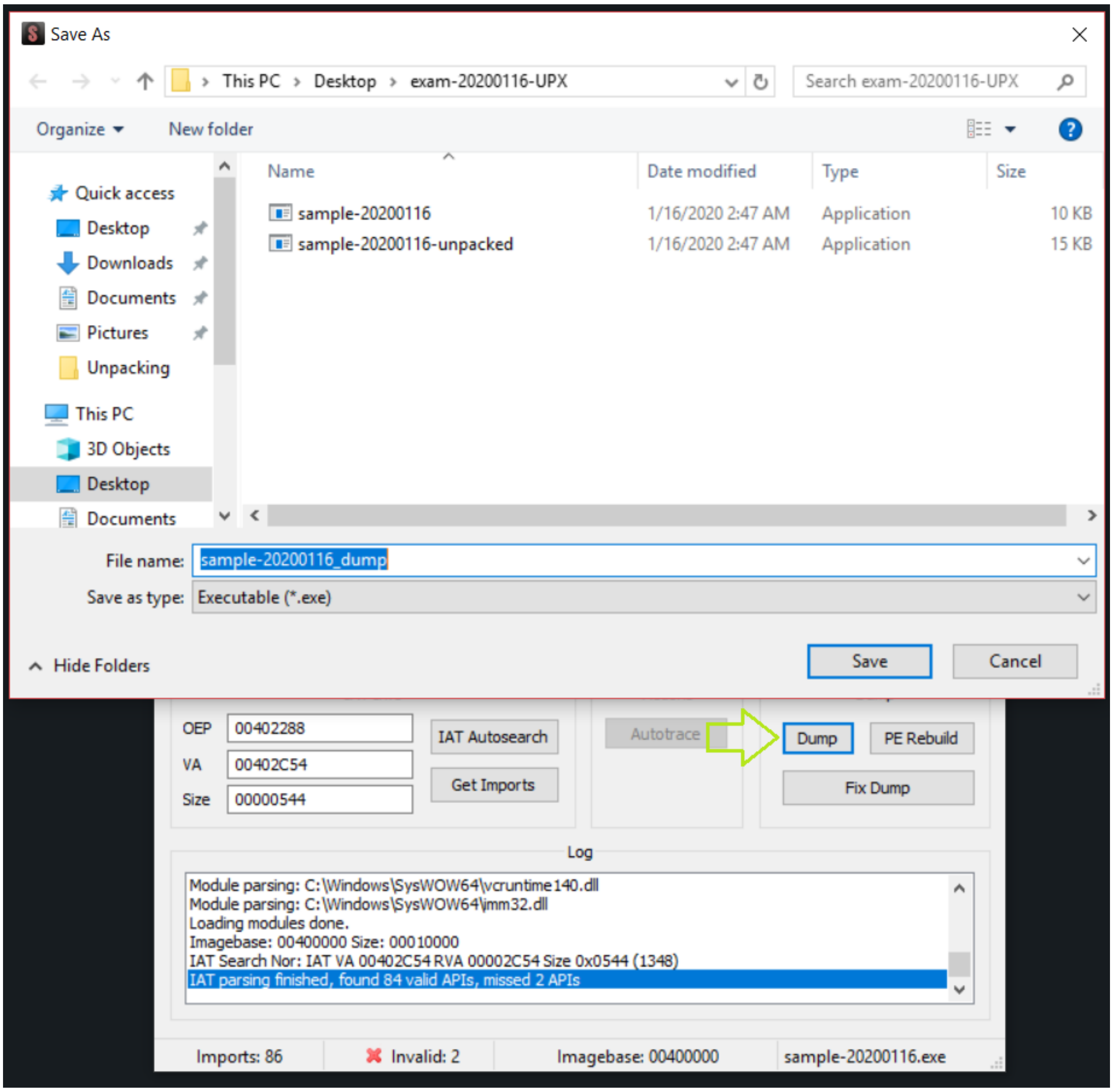
```
Module parsing: C:\Windows\SysWOW64\vcruntime140.dll
Module parsing: C:\Windows\SysWOW64\imm32.dll
Loading modules done.
Imagebase: 00400000 Size: 00010000
IAT Search Nor: IAT VA 00402C54 RVA 00002C54 Size 0x0544 (1348)
IAT parsing finished, found 84 valid APIs, missed 2 APIs
```

Imports: 86 | Invalid: 2 | Imagebase: 00400000 | sample-20200116.exe

84 imports are found, and since we own the unpacked binary, we can double-check if the number of found imports matches with what Scylla [6] found.

In fact, PEStudio [3] told us that the original unpacked binary had 84 imports (as shows below), so we can confirm that our process had been successful. To create the unpacked binary, we simply clicked on "Dump" and saved the file, and it will create a new executable.





Name	Date modified	Type	Size
sample-20200116	1/16/2020 2:47 AM	Application	10 KB
sample-20200116-unpacked	1/16/2020 2:47 AM	Application	15 KB

File name: sample-20200116_dump

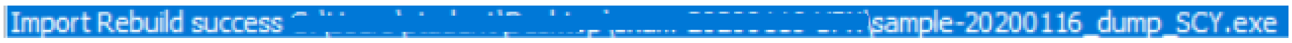
Save as type: Executable (*.exe)

OEP: 00402288
VA: 00402C54
Size: 00000544

```
Module parsing: C:\Windows\SysWOW64\vcruntime140.dll
Module parsing: C:\Windows\SysWOW64\imm32.dll
Loading modules done.
Imagebase: 00400000 Size: 00010000
IAT Search Nor: IAT VA 00402C54 RVA 00002C54 Size 0x0544 (1348)
IAT parsing finished, found 84 valid APIs, missed 2 APIs
```

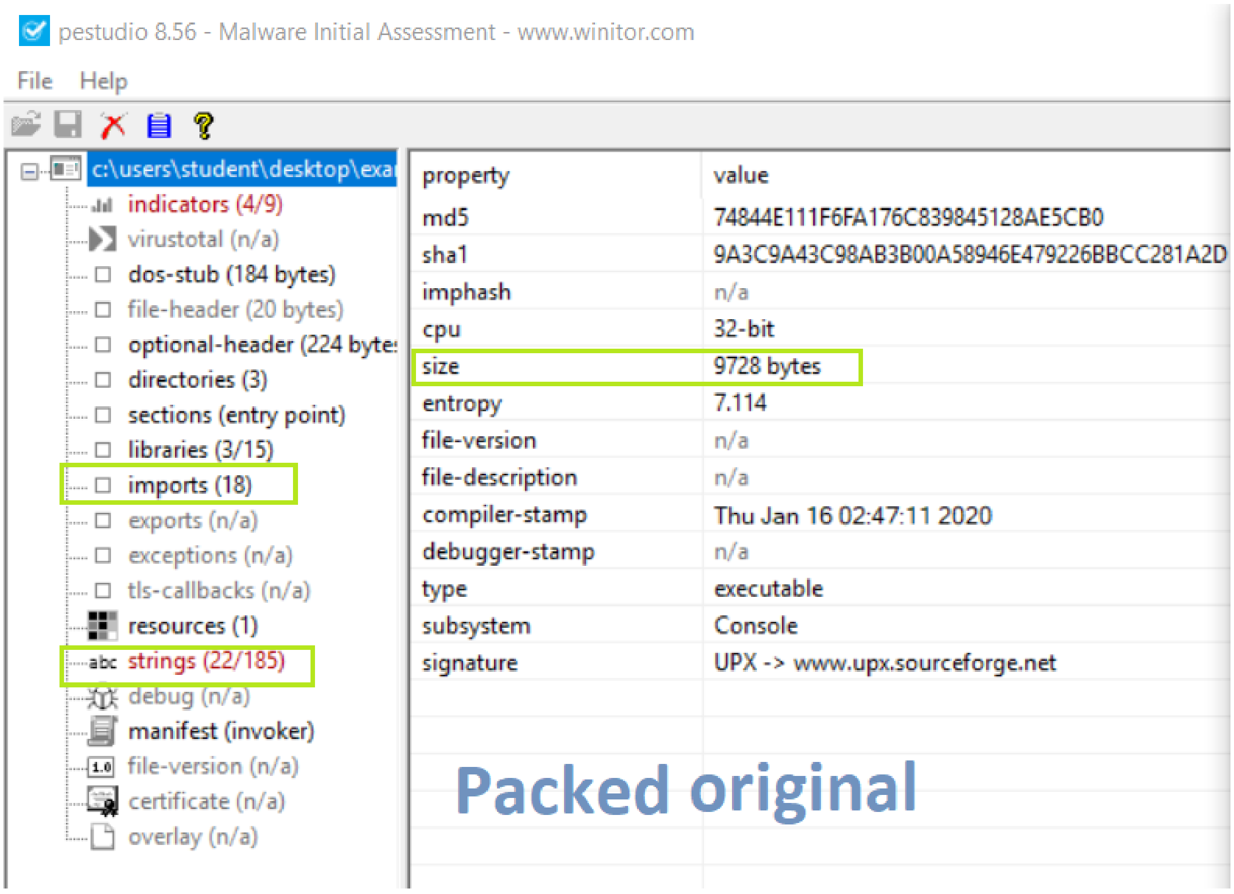
Imports: 86 | ✖ Invalid: 2 | Imagebase: 00400000 | sample-20200116.exe

Then, click on "Fix Dump" and chose the dump that had been created previously to create the unpacked binary. It may not be identical to the original file, but it will be really close.



```
Import Rebuild success [redacted]\sample-20200116_dump_SCY.exe
```

To further check, we load the new binary with PEStudio [3] against the packed file too see how they differ from each other.

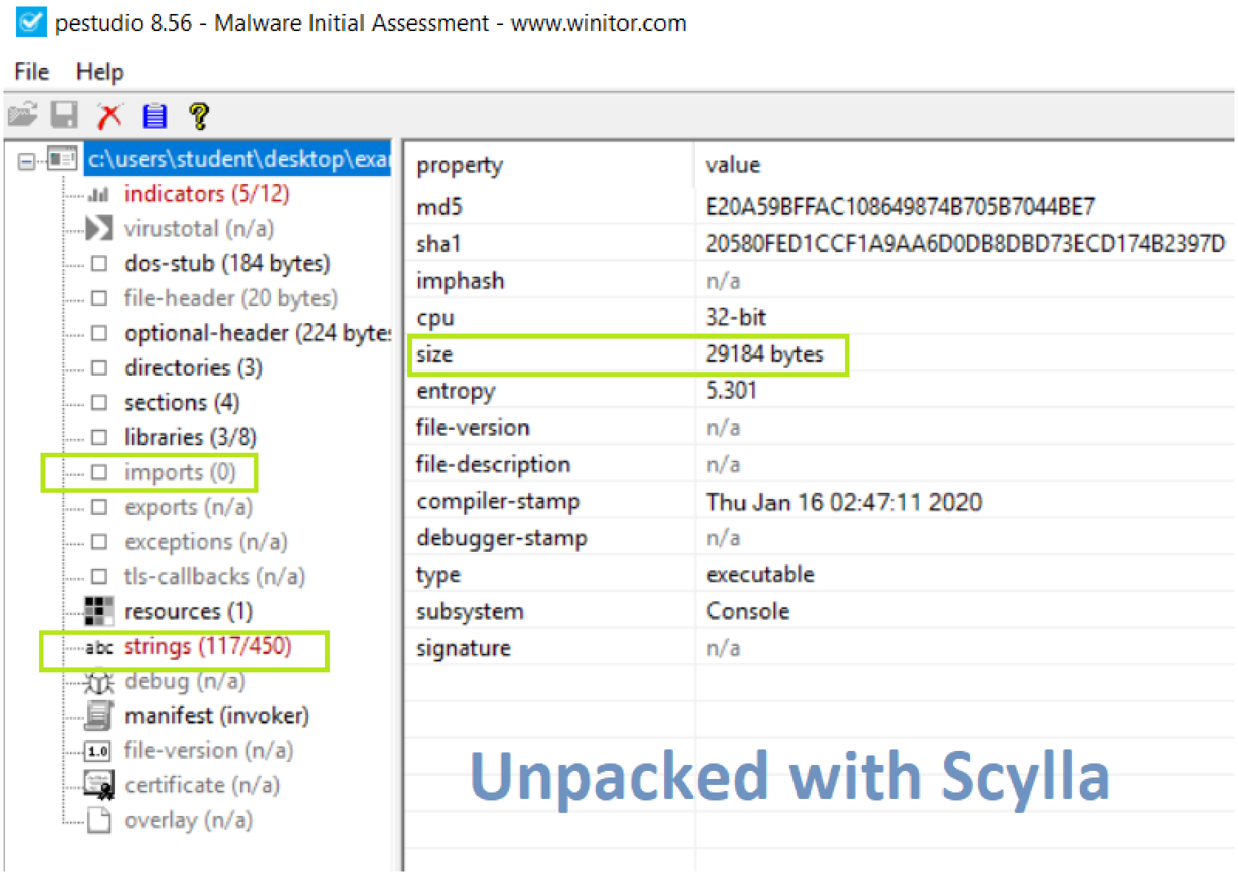

pestudio 8.56 - Malware Initial Assessment - www.winitor.com

File Help

c:\users\student\desktop\exa
- indicators (4/9)
- virustotal (n/a)
- dos-stub (184 bytes)
- file-header (20 bytes)
- optional-header (224 byte
- directories (3)
- sections (entry point)
- libraries (3/15)
- imports (18)
- exports (n/a)
- exceptions (n/a)
- tls-callbacks (n/a)
- resources (1)
- strings (22/185)
- debug (n/a)
- manifest (invoker)
- file-version (n/a)
- certificate (n/a)
- overlay (n/a)

property	value
md5	74844E111F6FA176C839845128AE5CB0
sha1	9A3C9A43C98AB3B00A58946E479226BBCC281A2D
imphash	n/a
cpu	32-bit
size	9728 bytes
entropy	7.114
file-version	n/a
file-description	n/a
compiler-stamp	Thu Jan 16 02:47:11 2020
debugger-stamp	n/a
type	executable
subsystem	Console
signature	UPX -> www.upx.sourceforge.net

Packed original


pestudio 8.56 - Malware Initial Assessment - www.winitor.com

File Help

c:\users\student\desktop\exa
- indicators (5/12)
- virustotal (n/a)
- dos-stub (184 bytes)
- file-header (20 bytes)
- optional-header (224 byte
- directories (3)
- sections (4)
- libraries (3/8)
- imports (0)
- exports (n/a)
- exceptions (n/a)
- tls-callbacks (n/a)
- resources (1)
- strings (117/450)
- debug (n/a)
- manifest (invoker)
- file-version (n/a)
- certificate (n/a)
- overlay (n/a)

property	value
md5	E20A59BFFAC108649874B705B7044BE7
sha1	20580FED1CCF1A9AA6D0DB8DBD73ECD174B2397D
imphash	n/a
cpu	32-bit
size	29184 bytes
entropy	5.301
file-version	n/a
file-description	n/a
compiler-stamp	Thu Jan 16 02:47:11 2020
debugger-stamp	n/a
type	executable
subsystem	Console
signature	n/a

Unpacked with Scylla

In the preview's images, we are able to see the difference between the original packed binary and the unpacked binary that we have created. The first thing we notice is that the size of our file was much larger, which is expected. Another thing that we notice is that the import of the new unpacked binary is 0. This is actually a problem with Scylla [6], as long as the import were showing in Scylla [6] there is no problem with the unpacked binary even if PEStudio [3] are not showing them. Lastly, we can see that there are a higher number of strings than before which is also an expected change in the binary.

And with that, concludes the manual unpacking of a packed binary with UPX.

5.5 - MPRESS

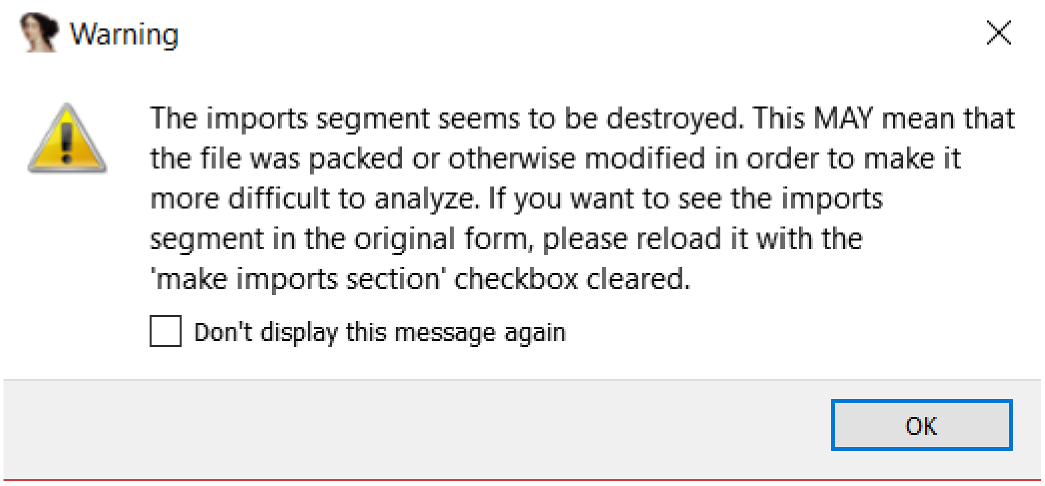
Unpacking executable files that have been compressed and protected by the MPRESS packer involves a distinct process compared to the previously discussed UPX packer. While the initial steps to identify the presence of packing in a file remain the same, the subsequent unpacking process and analysis of the MPRESS-packed executables require a different approach.

Similar to the UPX packer, the first step involves conducting a preliminary analysis of the file using tools such as PEStudio [3] and DetectItEasy [4] to identify indicators of packing. These indicators include reduced numbers of imports, unusually small sections, significant differences between the virtual size and raw size of sections, and other relevant characteristics.

It is important to introduce concept of hardware breakpoint to understand the steps where it is involved and its difference from software breakpoint. A hardware breakpoint is a powerful debugging feature offered by modern processors and debuggers that allows software developers and analysts to pause the execution of a program at a specific memory address or when a particular condition is met. Hardware breakpoint is an essential tool for debugging complex software, analyzing malware, and investigating software vulnerabilities. These breakpoints are typically implemented using specialized registers called debug registers, which are part of the processor's debugging architecture.

To set a hardware breakpoint, the debugger configures one of the available debug registers with the desired memory address or condition. The debug register is then monitored by the processor, which halts program execution when any instruction attempts to read, write, or execute the specified address or the condition is met.

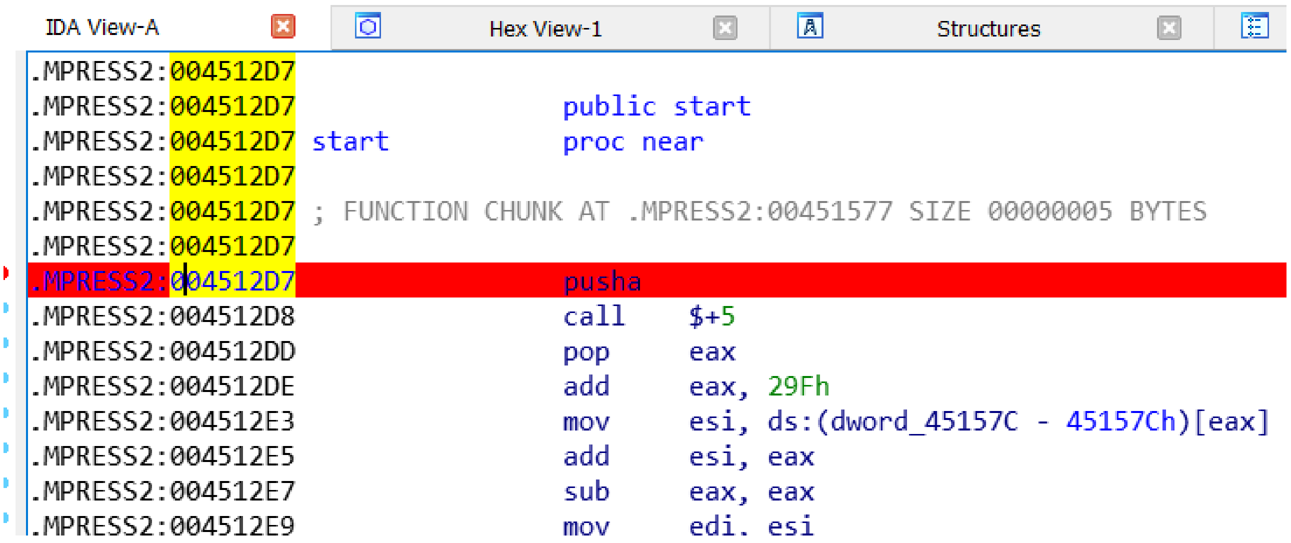
Given this introduction we can proceed to the process of identifying a packed binary with MPRESS which is identical to that of UPX, as shown above. Upon loading the binary packed with MPRESS we receive a similar message to the UPX one, but in this case, clearly stating that the file was packed.


Warning

The imports segment seems to be destroyed. This MAY mean that the file was packed or otherwise modified in order to make it more difficult to analyze. If you want to see the imports segment in the original form, please reload it with the 'make imports section' checkbox cleared.

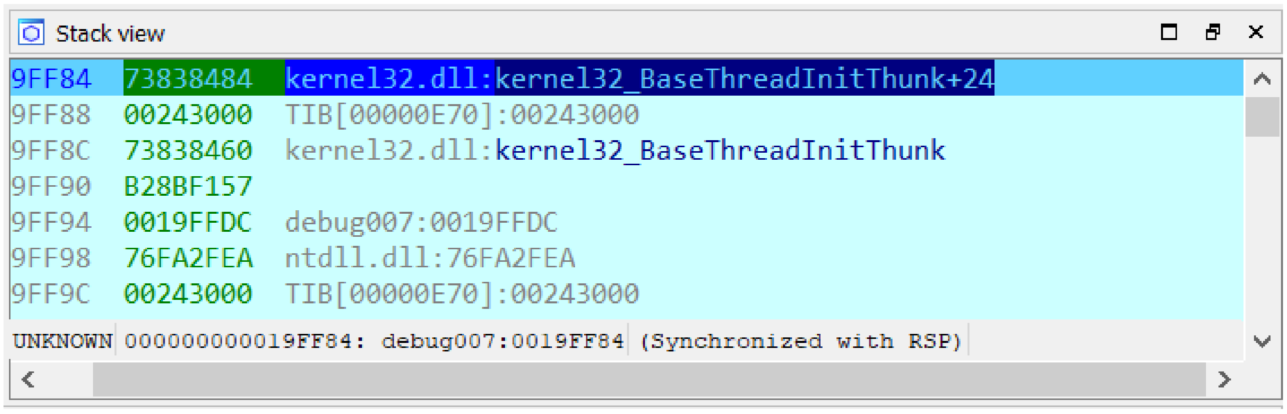
☐ Don't display this message again

Once the binary had been loaded, we are supposed to face a "pusha" instruction. If this is not the case, we can simply select the start function and insert a breakpoint.



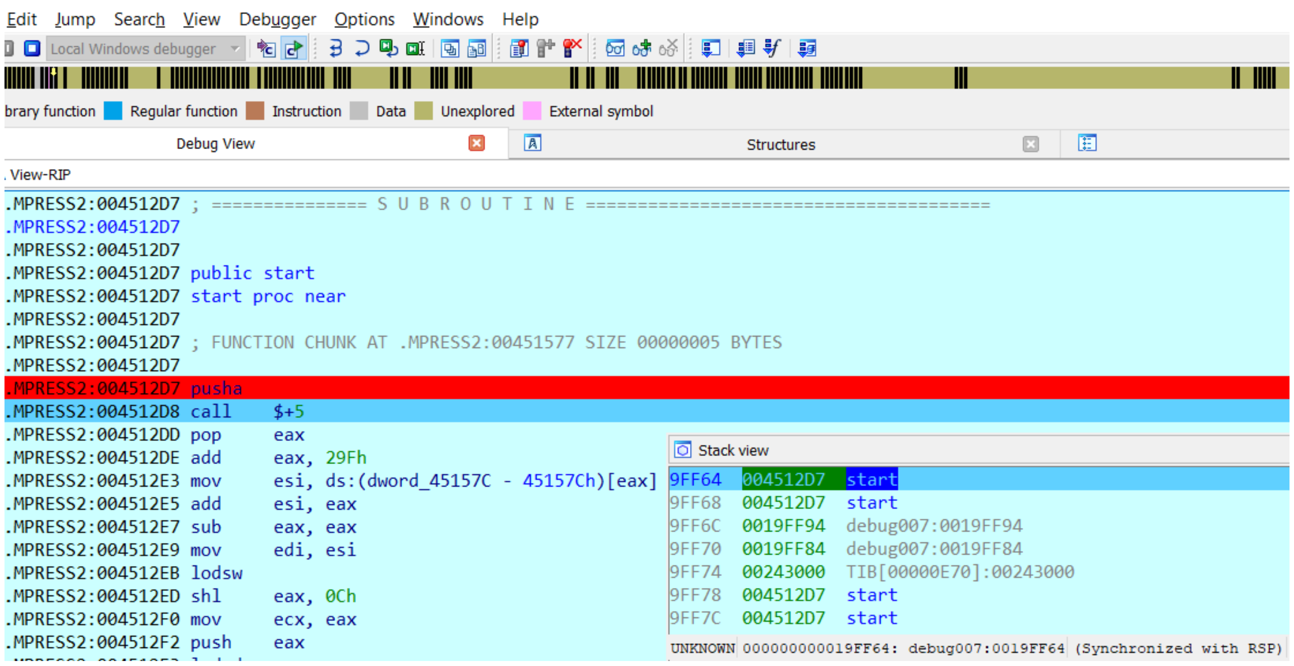
```
.MPRESS2:004512D7
.MPRESS2:004512D7                 public start
.MPRESS2:004512D7 start           proc near
.MPRESS2:004512D7
.MPRESS2:004512D7 ; FUNCTION CHUNK AT .MPRESS2:00451577 SIZE 00000005 BYTES
.MPRESS2:004512D7
.MPRESS2:004512D7                 pusha
.MPRESS2:004512D8                 call    $+5
.MPRESS2:004512DD                 pop     eax
.MPRESS2:004512DE                 add     eax, 29Fh
.MPRESS2:004512E3                 mov     esi, ds:(dword_45157C - 45157Ch)[eax]
.MPRESS2:004512E5                 add     esi, eax
.MPRESS2:004512E7                 sub     eax, eax
.MPRESS2:004512E9                 mov     edi, esi
```

Then, we run the binary, paying particular attention to the Stack View window (usually it is located in the bottom-right corner of the IDA [5] window).



```
9FF84  73838484  kernel32.dll:kernel32_BaseThreadInitThunk+24
9FF88  00243000  TIB[00000E70]:00243000
9FF8C  73838460  kernel32.dll:kernel32_BaseThreadInitThunk
9FF90  B28BF157
9FF94  0019FFDC  debug007:0019FFDC
9FF98  76FA2FEA  ntdll.dll:76FA2FEA
9FF9C  00243000  TIB[00000E70]:00243000
UNKNOWN 000000000019FF84: debug007:0019FF84 (Synchronized with RSP)
```

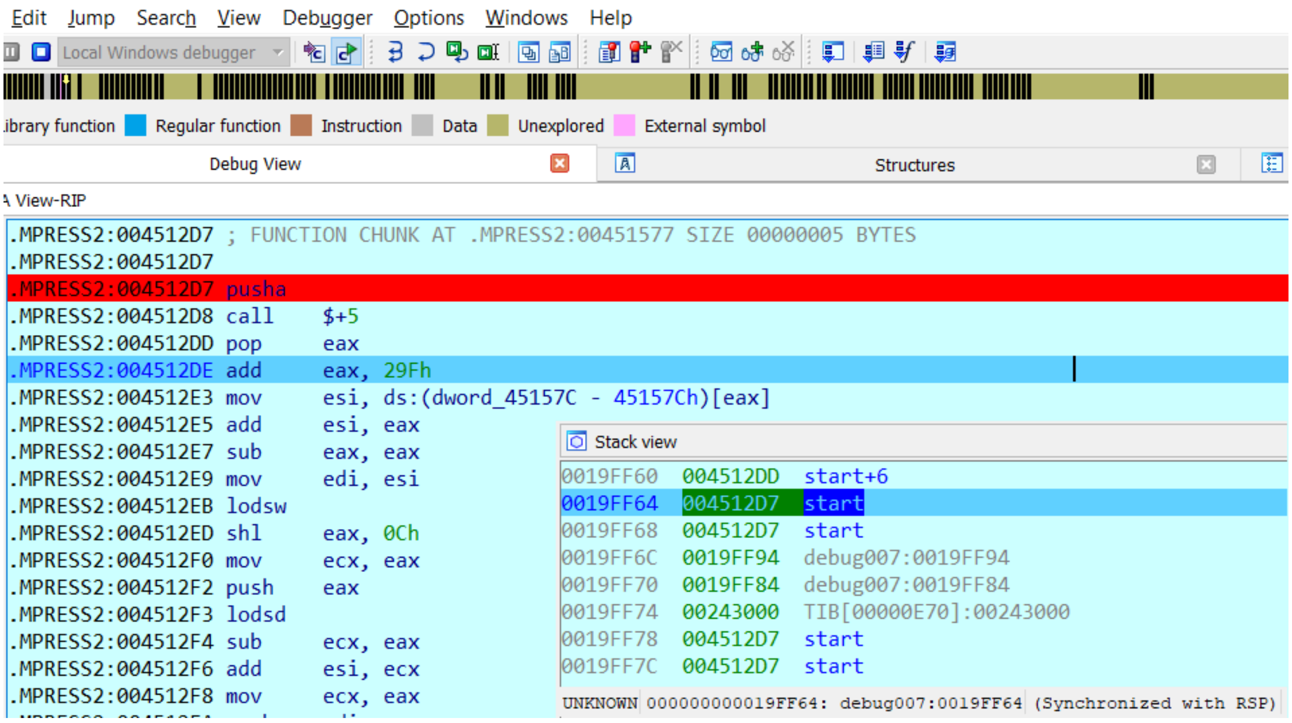
Upon stepping over once we can notice that the addresses on the stack had changed (shown in the image below).



```
.MPRESS2:004512D7 ; =============== S U B R O U T I N E =======================================
.MPRESS2:004512D7
.MPRESS2:004512D7
.MPRESS2:004512D7 public start
.MPRESS2:004512D7 start proc near
.MPRESS2:004512D7
.MPRESS2:004512D7 ; FUNCTION CHUNK AT .MPRESS2:00451577 SIZE 00000005 BYTES
.MPRESS2:004512D7
.MPRESS2:004512D7 pusha
.MPRESS2:004512D8 call    $+5
.MPRESS2:004512DD pop     eax
.MPRESS2:004512DE add     eax, 29Fh
.MPRESS2:004512E3 mov     esi, ds:(dword_45157C - 45157Ch)[eax]
.MPRESS2:004512E5 add     esi, eax
.MPRESS2:004512E7 sub     eax, eax
.MPRESS2:004512E9 mov     edi, esi
.MPRESS2:004512EB lodsw
.MPRESS2:004512ED shl     eax, 0Ch
.MPRESS2:004512F0 mov     ecx, eax
.MPRESS2:004512F2 push    eax
```

```
9FF64  004512D7  start
9FF68  004512D7  start
9FF6C  0019FF94  debug007:0019FF94
9FF70  0019FF84  debug007:0019FF84
9FF74  00243000  TIB[00000E70]:00243000
9FF78  004512D7  start
9FF7C  004512D7  start
UNKNOWN 000000000019FF64: debug007:0019FF64 (Synchronized with RSP)
```

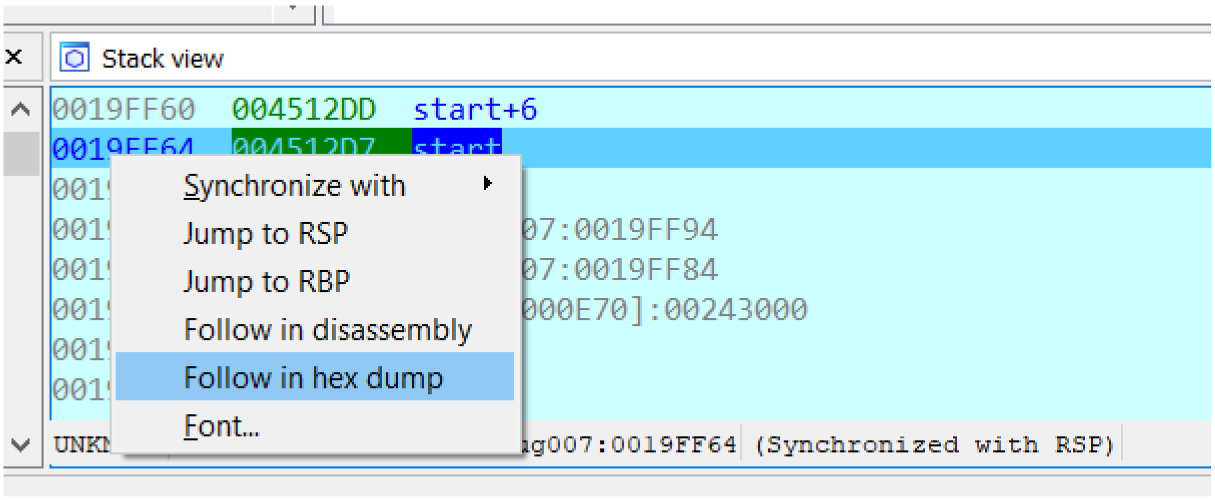
At this point, the stack is pointing to start. We have to step over until we come back again to "start" since the start appears twice on the stack, once at 9FF64 and once at 9FF68. In this example, it was enough to do "Step Over" twice.



```
.MPRESS2:004512D7 ; FUNCTION CHUNK AT .MPRESS2:00451577 SIZE 00000005 BYTES
.MPRESS2:004512D7
.MPRESS2:004512D7 pusha
.MPRESS2:004512D8 call    $+5
.MPRESS2:004512DD pop     eax
.MPRESS2:004512DE add     eax, 29Fh
.MPRESS2:004512E3 mov     esi, ds:(dword_45157C - 45157Ch)[eax]
.MPRESS2:004512E5 add     esi, eax
.MPRESS2:004512E7 sub     eax, eax
.MPRESS2:004512E9 mov     edi, esi
.MPRESS2:004512EB lodsw
.MPRESS2:004512ED shl     eax, 0Ch
.MPRESS2:004512F0 mov     ecx, eax
.MPRESS2:004512F2 push    eax
.MPRESS2:004512F3 lodsd
.MPRESS2:004512F4 sub     ecx, eax
.MPRESS2:004512F6 add     esi, ecx
.MPRESS2:004512F8 mov     ecx, eax
```

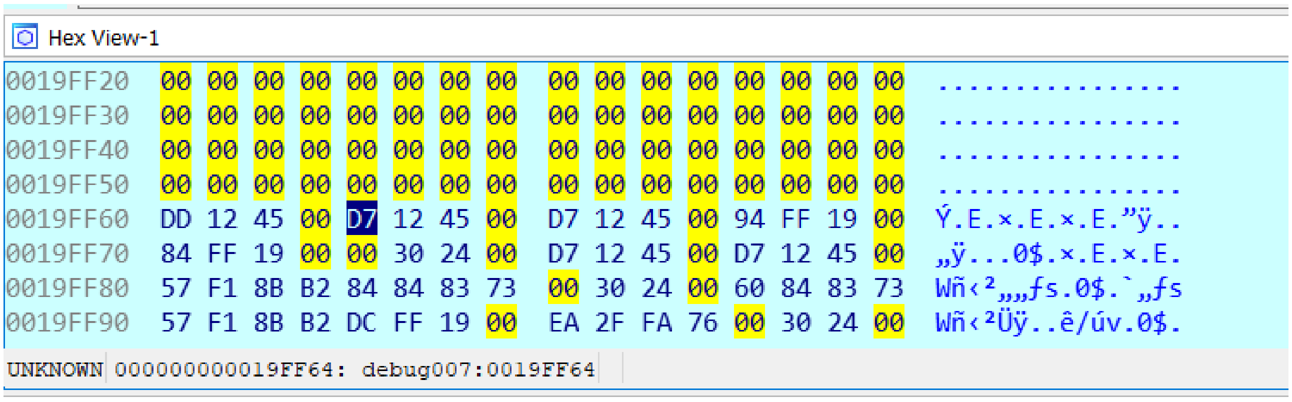
```
0019FF60  004512DD  start+6
0019FF64  004512D7  start
0019FF68  004512D7  start
0019FF6C  0019FF94  debug007:0019FF94
0019FF70  0019FF84  debug007:0019FF84
0019FF74  00243000  TIB[00000E70]:00243000
0019FF78  004512D7  start
0019FF7C  004512D7  start
UNKNOWN 000000000019FF64: debug007:0019FF64 (Synchronized with RSP)
```

Note that we started from the "pusha" instruction at 4512D7 and then moved to 4512DE. Then we right clicked on the stack address 19FF64 and followed it in the hex dump.



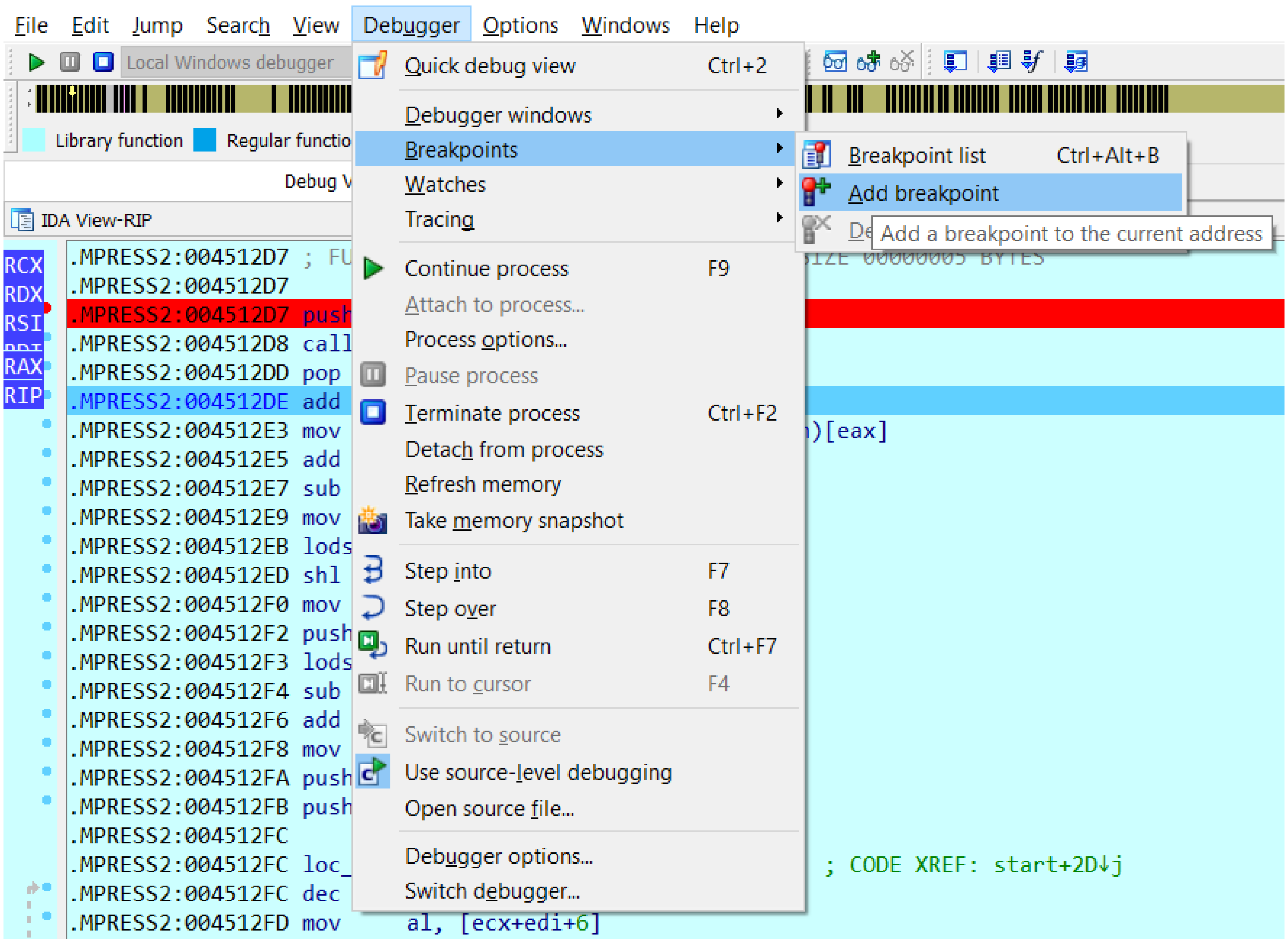
```
0019FF60  004512DD  start+6
0019FF64  004512D7  start
UNKN                g007:0019FF64 (Synchronized with RSP)
```

The “Hex View-1” window changed, and a byte gets highlighted.



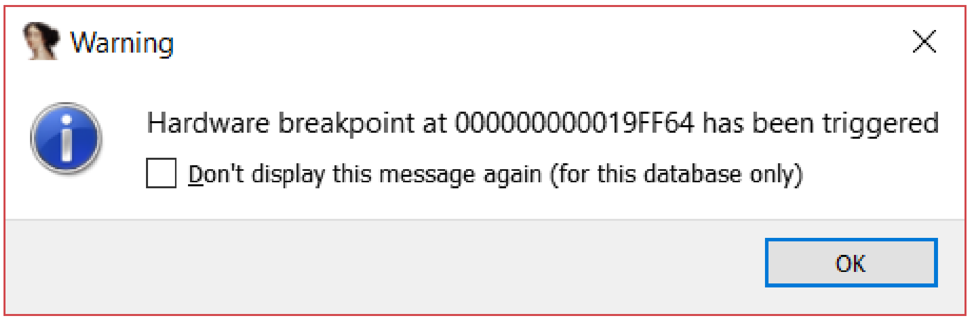
```
Hex View-1
0019FF20  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0019FF30  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0019FF40  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0019FF50  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0019FF60  DD 12 45 00 D7 12 45 00  D7 12 45 00 94 FF 19 00  Ý.E.×.E.×.E.”ÿ..
0019FF70  84 FF 19 00 00 30 24 00  D7 12 45 00 D7 12 45 00  „ÿ...0$.×.E.×.E.
0019FF80  57 F1 8B B2 84 84 83 73  00 30 24 00 60 84 83 73  Wñ‹²„„ƒs.0$.`„ƒs
0019FF90  57 F1 8B B2 DC FF 19 00  EA 2F FA 76 00 30 24 00  Wñ‹²Üÿ..ê/úv.0$.
UNKNOWN 000000000019FF64: debug007:0019FF64
```

At this point, we select the four bytes (DWORD) starting from D7 and put a breakpoint on it. We also set the breakpoint to be a hardware breakpoint with read and write permission (WR), in this case, IDA [5] automatically enables these permissions. Although it is possible to view and modify the breakpoint which we don't need in this case.



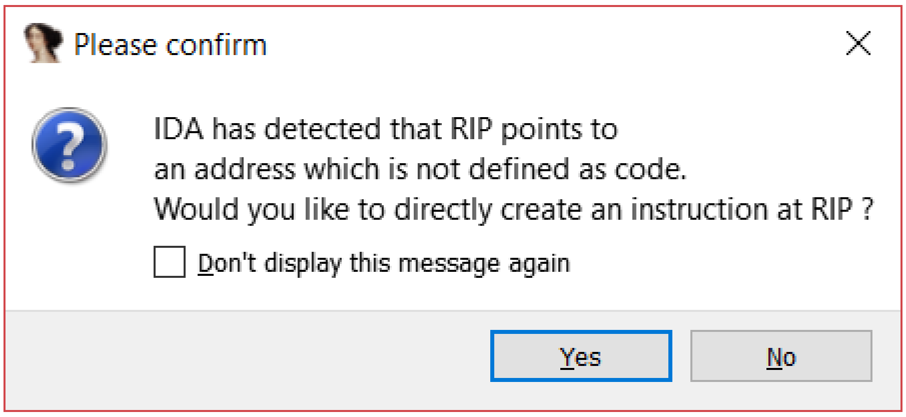
```
.MPRESS2:004512D7 ; FU
.MPRESS2:004512D7
.MPRESS2:004512D7 push
.MPRESS2:004512D8 call
.MPRESS2:004512DD pop
.MPRESS2:004512DE add
.MPRESS2:004512E3 mov                    )[eax]
.MPRESS2:004512E5 add
.MPRESS2:004512E7 sub
.MPRESS2:004512E9 mov
.MPRESS2:004512EB lods
.MPRESS2:004512ED shl
.MPRESS2:004512F0 mov
.MPRESS2:004512F2 push
.MPRESS2:004512F3 lods
.MPRESS2:004512F4 sub
.MPRESS2:004512F6 add
.MPRESS2:004512F8 mov
.MPRESS2:004512FA push
.MPRESS2:004512FB push
.MPRESS2:004512FC
.MPRESS2:004512FC loc_                   ; CODE XREF: start+2D↓j
.MPRESS2:004512FC dec
.MPRESS2:004512FD mov     al, [ecx+edi+6]
```

At this point we run the binary with the Play button. Once the binary is running, at a certain point it hit the hardware breakpoint. This was done in search of the Original Entry Point (OEP).


Warning

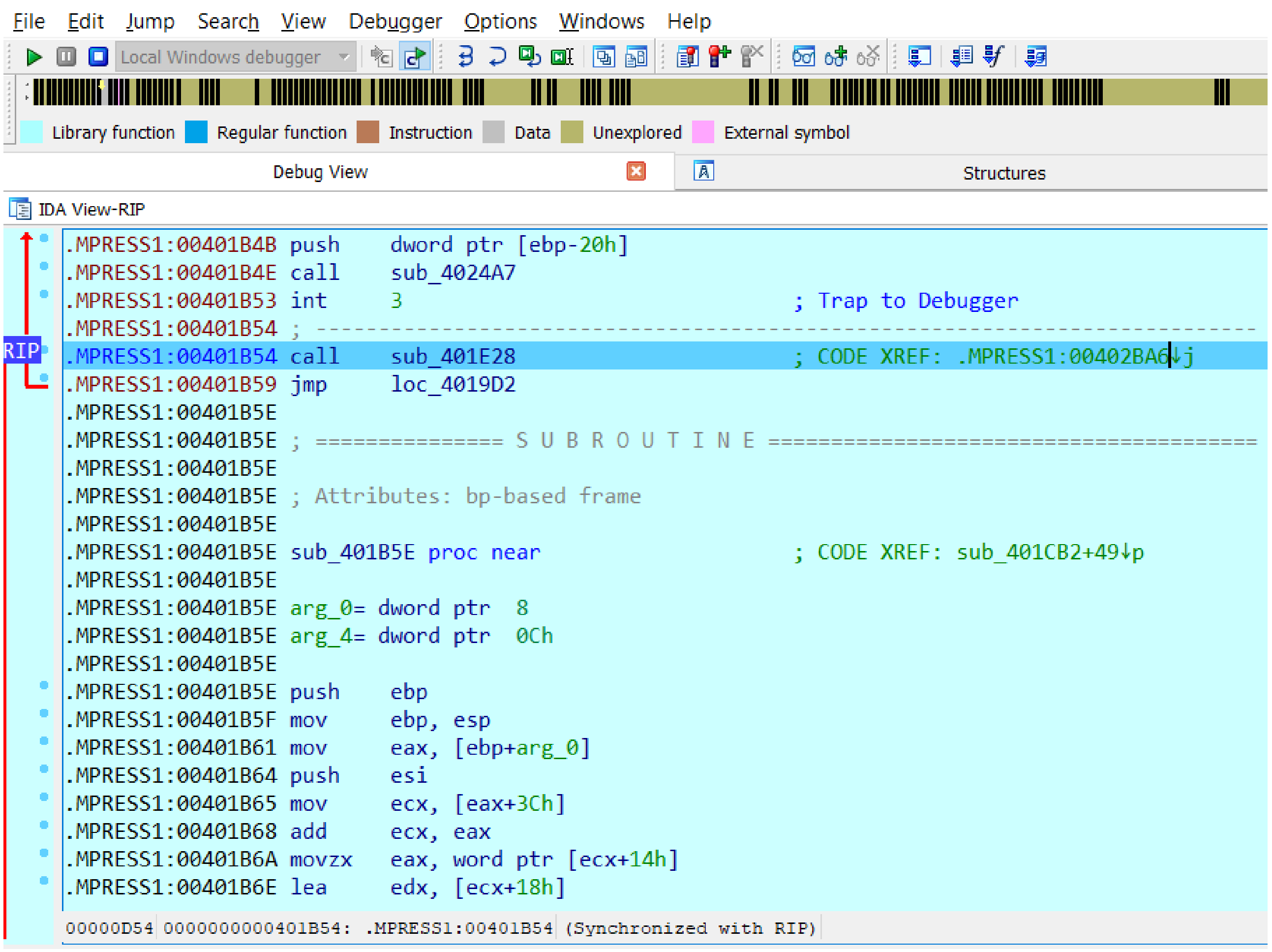
Hardware breakpoint at 000000000019FF64 has been triggered

IDA [5] then detected that RIP point is not in a defined address similarly as sees as of the UPX.


Please confirm

IDA has detected that RIP points to
an address which is not defined as code.
Would you like to directly create an instruction at RIP ?

After pressing "Yes", the code changes, and we are suddenly in a call instruction. When we stepped into it, IDA [5] told me the same message about RIP pointing to an undefined address. We cannot use the address of the call instruction (402BA6) as we did for UPX. Instead, we have to use the address pointed out by the call instruction (401B54).



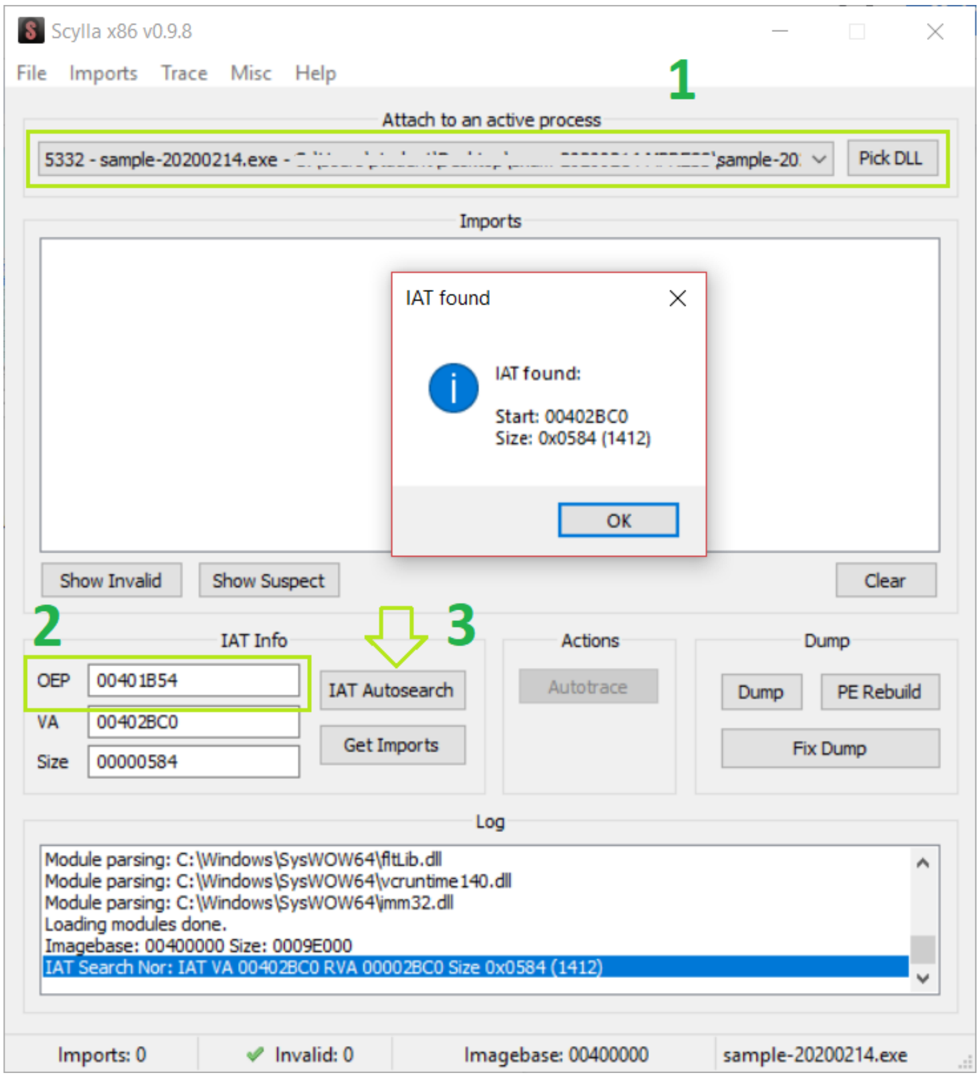
```
.MPRESS1:00401B4B push    dword ptr [ebp-20h]
.MPRESS1:00401B4E call    sub_4024A7
.MPRESS1:00401B53 int     3                         ; Trap to Debugger
.MPRESS1:00401B54 ; ---------------------------------------------------------------
.MPRESS1:00401B54 call    sub_401E28                ; CODE XREF: .MPRESS1:00402BA6↓j
.MPRESS1:00401B59 jmp     loc_4019D2
.MPRESS1:00401B5E
.MPRESS1:00401B5E ; =============== S U B R O U T I N E ===============================
.MPRESS1:00401B5E
.MPRESS1:00401B5E ; Attributes: bp-based frame
.MPRESS1:00401B5E
.MPRESS1:00401B5E sub_401B5E proc near              ; CODE XREF: sub_401CB2+49↓p
.MPRESS1:00401B5E
.MPRESS1:00401B5E arg_0= dword ptr  8
.MPRESS1:00401B5E arg_4= dword ptr  0Ch
.MPRESS1:00401B5E
.MPRESS1:00401B5E push    ebp
.MPRESS1:00401B5F mov     ebp, esp
.MPRESS1:00401B61 mov     eax, [ebp+arg_0]
.MPRESS1:00401B64 push    esi
.MPRESS1:00401B65 mov     ecx, [eax+3Ch]
.MPRESS1:00401B68 add     ecx, eax
.MPRESS1:00401B6A movzx   eax, word ptr [ecx+14h]
.MPRESS1:00401B6E lea     edx, [ecx+18h]
00000D54 0000000000401B54: .MPRESS1:00401B54 (Synchronized with RIP)
```

Finally, the address 401B54 can be used as the Original Entry Point (OEP) and unpack with Scylla [6]. Once found the OEP the unpacking process with Scylla [6] is very much the same as the one seed for the UPX which we will perform to validate its correctness. It is important to remember to restart Scylla [6] due to its bug and as seen previously we cannot close IDA [5] which was in debugging mode to let Scylla [6] use the process in execution.

In order to unpack the steps to follow are the same as seen for UPX:

1. Select the process of IDA [5] running the sample.
2. Insert in the OEP field the address found on IDA [5].
3. Click “IAT Autosearch”

Upon setting up Scylla [6] with the new values for the malware sample, it successfully finds the start address and its size.



5332 - sample-20200214.exe - [redacted] sample-20

IAT found

IAT found:

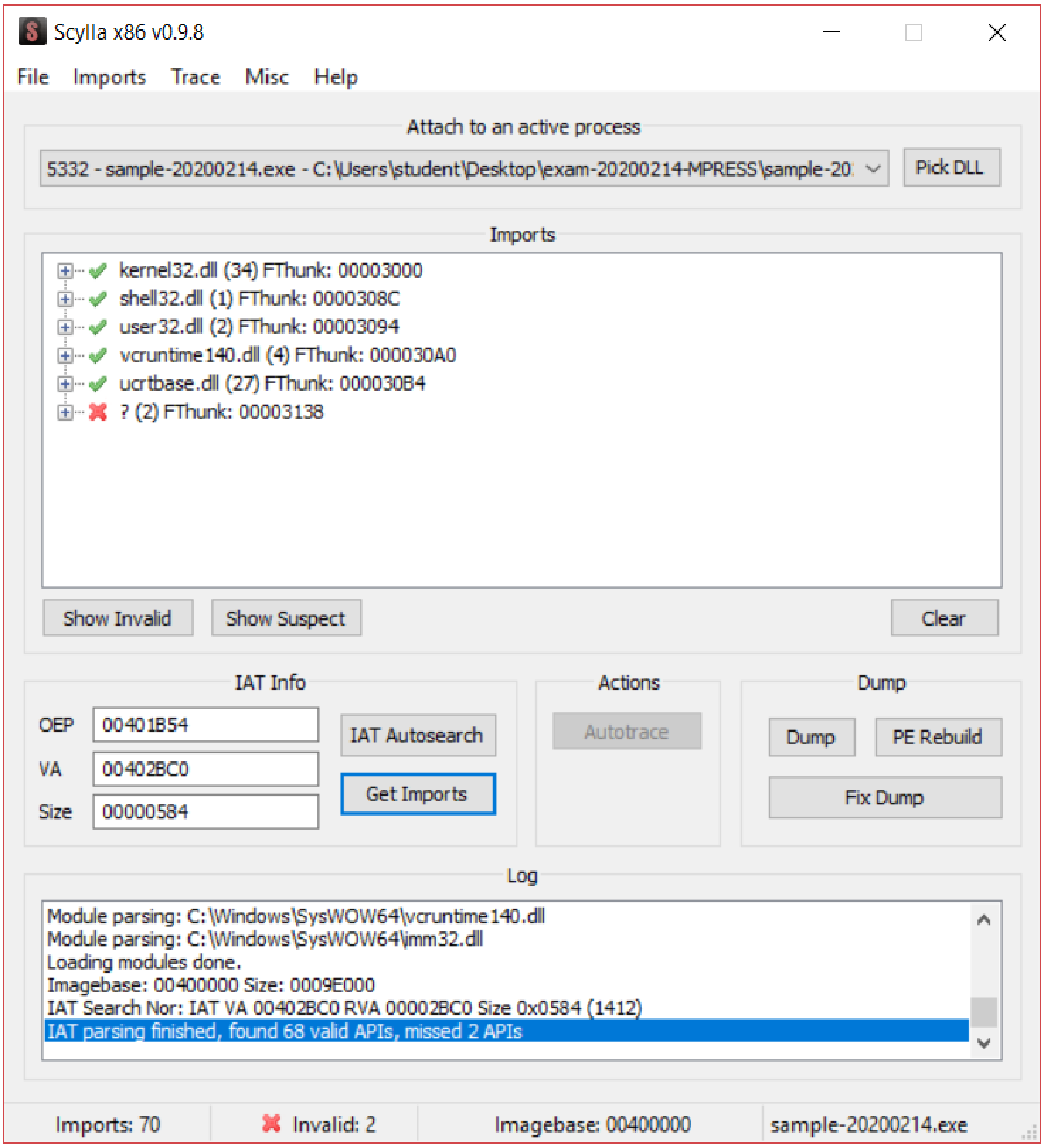
Start: 00402BC0
Size: 0x0584 (1412)

OEP 00401B54
VA 00402BC0
Size 00000584

```
Module parsing: C:\Windows\SysWOW64\fltLib.dll
Module parsing: C:\Windows\SysWOW64\vcruntime140.dll
Module parsing: C:\Windows\SysWOW64\imm32.dll
Loading modules done.
Imagebase: 00400000 Size: 0009E000
IAT Search Nor: IAT VA 00402BC0 RVA 00002BC0 Size 0x0584 (1412)
```

Imports: 0 Invalid: 0 Imagebase: 00400000 sample-20200214.exe

At this point, we obtained all the imports by simply clicking “Get Imports”.



5332 - sample-20200214.exe - C:\Users\student\Desktop\exam-20200214-MPRESS\sample-20

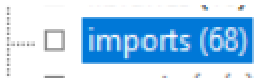
```
kernel32.dll (34) FThunk: 00003000
shell32.dll (1) FThunk: 0000308C
user32.dll (2) FThunk: 00003094
vcruntime140.dll (4) FThunk: 000030A0
ucrtbase.dll (27) FThunk: 000030B4
? (2) FThunk: 00003138
```

OEP 00401B54
VA 00402BC0
Size 00000584

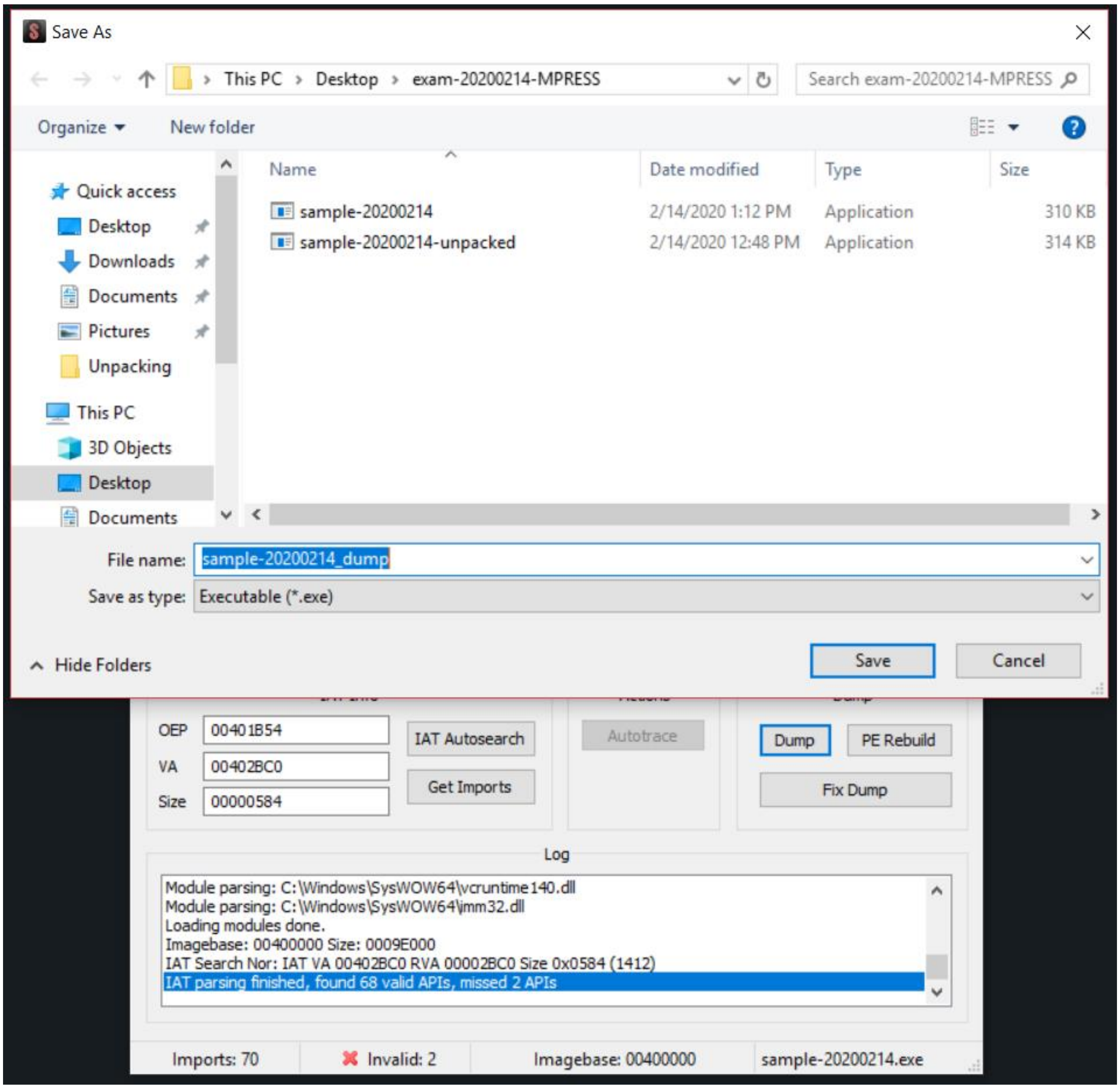
```
Module parsing: C:\Windows\SysWOW64\vcruntime140.dll
Module parsing: C:\Windows\SysWOW64\imm32.dll
Loading modules done.
Imagebase: 00400000 Size: 0009E000
IAT Search Nor: IAT VA 00402BC0 RVA 00002BC0 Size 0x0584 (1412)
IAT parsing finished, found 68 valid APIs, missed 2 APIs
```

Imports: 70 | Invalid: 2 | Imagebase: 00400000 | sample-20200214.exe

Scylla [6] shows that 68 imports are found since we own the unpacked binary, we can double-checked if the found imports matched the original number of imports.


imports (68)

In fact, PEStudio [3] had told us that the unpacked binary had 68 imports, so our process was successful. Now to create the unpacked binary, we first create a dump by clicking on "Dump" and saved the new file.



Name	Date modified	Type	Size
sample-20200214	2/14/2020 1:12 PM	Application	310 KB
sample-20200214-unpacked	2/14/2020 12:48 PM	Application	314 KB

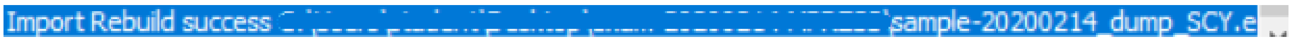
File name: sample-20200214_dump
Save as type: Executable (*.exe)

OEP 00401B54
VA 00402BC0
Size 00000584

```
Module parsing: C:\Windows\SysWOW64\vcruntime140.dll
Module parsing: C:\Windows\SysWOW64\imm32.dll
Loading modules done.
Imagebase: 00400000 Size: 0009E000
IAT Search Nor: IAT VA 00402BC0 RVA 00002BC0 Size 0x0584 (1412)
IAT parsing finished, found 68 valid APIs, missed 2 APIs
```

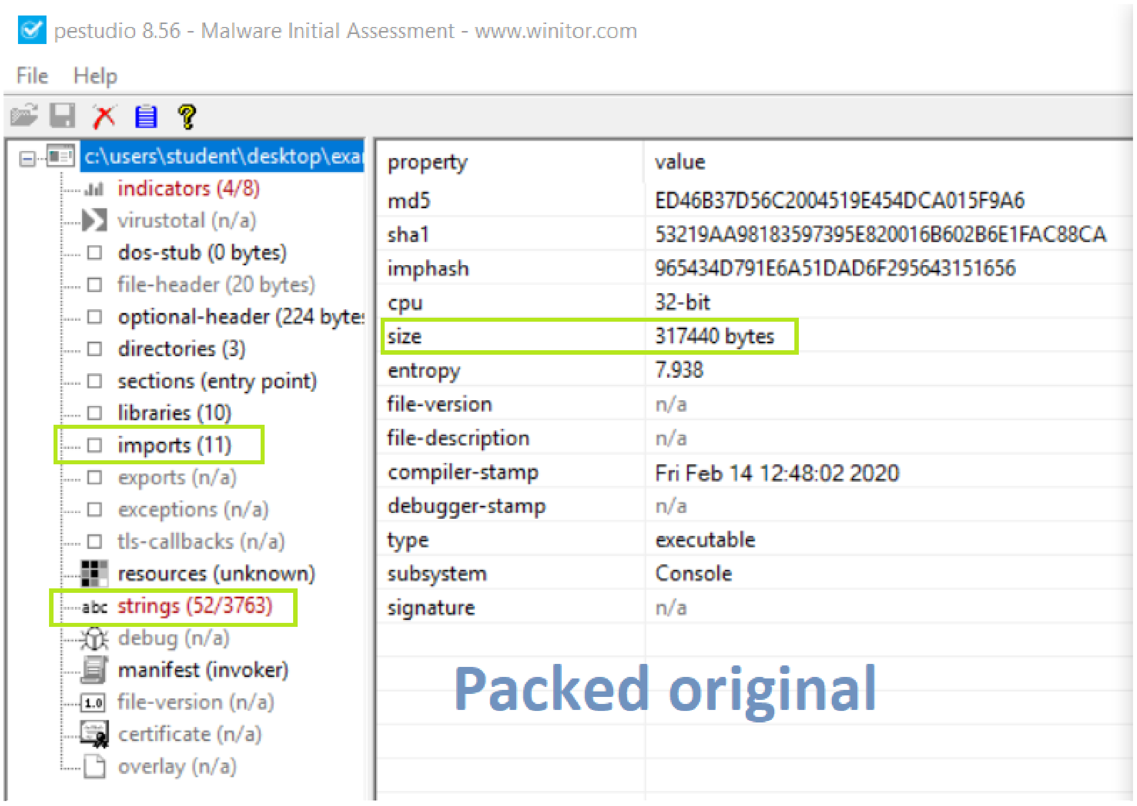
Imports: 70 | Invalid: 2 | Imagebase: 00400000 | sample-20200214.exe

Now we click “Fix Dump” and select the dump file previously created in order to unpack it and create the unpacked binary.



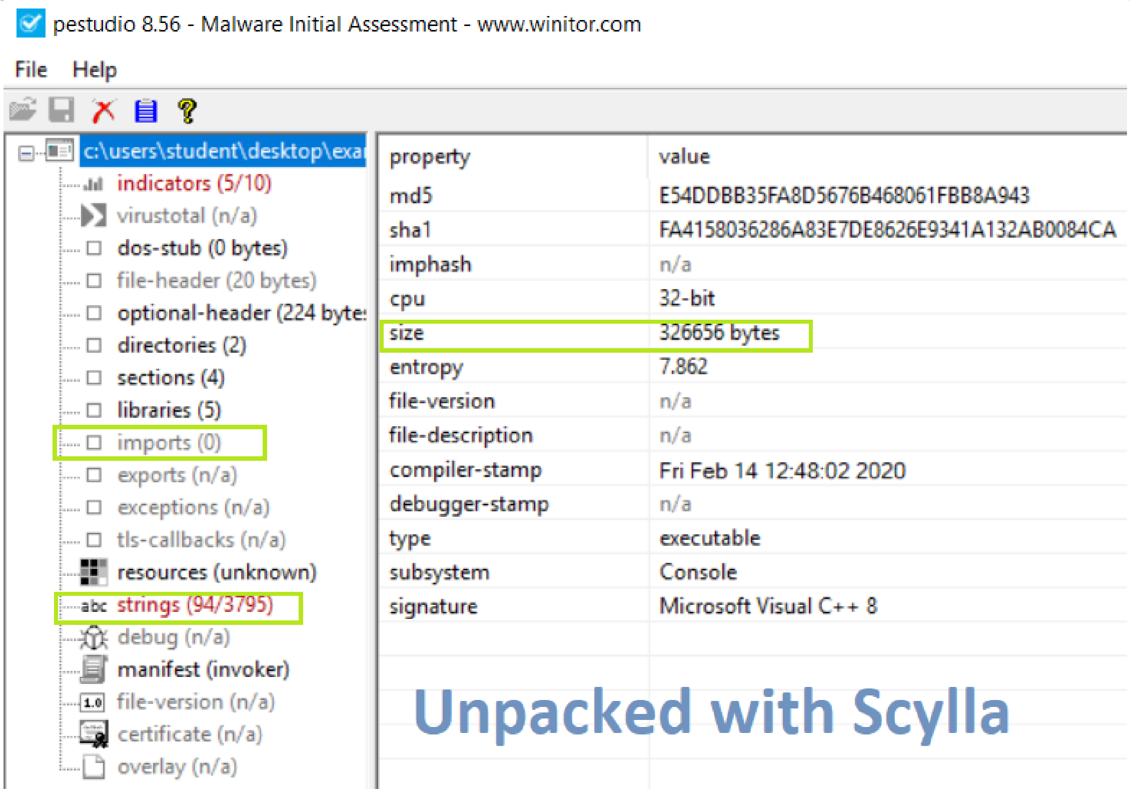
```
Import Rebuild success [redacted]sample-20200214_dump_SCY.e
```

To make further check we can load the newly created binary in PEStudio [3] and compared it with the original packed binary to see their differences.


pestudio 8.56 - Malware Initial Assessment - www.winitor.com
File Help
Packed original

- c:\users\student\desktop\exa
- indicators (4/8)
- virustotal (n/a)
- dos-stub (0 bytes)
- file-header (20 bytes)
- optional-header (224 byte
- directories (3)
- sections (entry point)
- libraries (10)
- imports (11)
- exports (n/a)
- exceptions (n/a)
- tls-callbacks (n/a)
- resources (unknown)
- strings (52/3763)
- debug (n/a)
- manifest (invoker)
- file-version (n/a)
- certificate (n/a)
- overlay (n/a)

property	value
md5	ED46B37D56C2004519E454DCA015F9A6
sha1	53219AA98183597395E820016B602B6E1FAC88CA
imphash	965434D791E6A51DAD6F295643151656
cpu	32-bit
size	317440 bytes
entropy	7.938
file-version	n/a
file-description	n/a
compiler-stamp	Fri Feb 14 12:48:02 2020
debugger-stamp	n/a
type	executable
subsystem	Console
signature	n/a


pestudio 8.56 - Malware Initial Assessment - www.winitor.com
File Help
Unpacked with Scylla

- c:\users\student\desktop\exa
- indicators (5/10)
- virustotal (n/a)
- dos-stub (0 bytes)
- file-header (20 bytes)
- optional-header (224 byte
- directories (2)
- sections (4)
- libraries (5)
- imports (0)
- exports (n/a)
- exceptions (n/a)
- tls-callbacks (n/a)
- resources (unknown)
- strings (94/3795)
- debug (n/a)
- manifest (invoker)
- file-version (n/a)
- certificate (n/a)
- overlay (n/a)

property	value
md5	E54DDBB35FA8D5676B468061FBB8A943
sha1	FA4158036286A83E7DE8626E9341A132AB0084CA
imphash	n/a
cpu	32-bit
size	326656 bytes
entropy	7.862
file-version	n/a
file-description	n/a
compiler-stamp	Fri Feb 14 12:48:02 2020
debugger-stamp	n/a
type	executable
subsystem	Console
signature	Microsoft Visual C++ 8

Finally, we could see the differences between the original packed binary and the unpacked binary that we created. Much like UPX, even here, the same value changes, even if the change is not significant. The unpacked version with Scylla [6] has 0 imports, which was the same as UPX. It is not a problem as long as Scylla [6] has found all the imports during the process.

This concludes the manual unpacking of binaries packed of MPRESS.

6 - Advanced Technique for Packer Identification

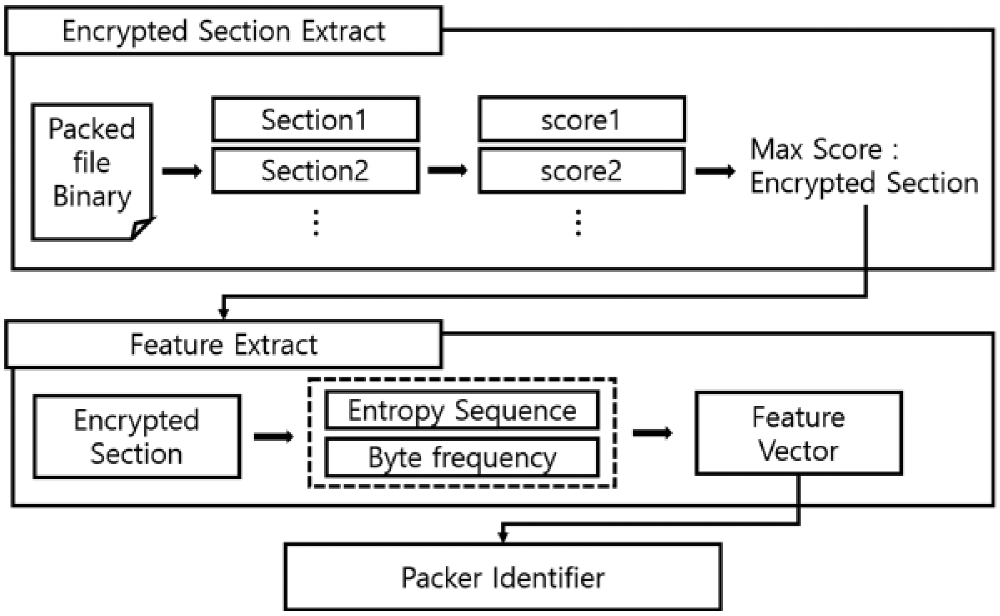
As we have seen, identifying whether a file is packed or not is very important and it is even more important to know beforehand which packer used to pack the malware. Malware writers often modify information regarding the packer within the executable to reduce the identification of the used packer by changing certain parameters such as the header or section names. In such a case the techniques shown earlier in section 5.4 may not be particularly effective and the amount of information obtained would be particularly reduced.

An advanced and innovative technic to identify the packer is proposed of the paper "Packer identification method based on byte sequences," which achieves the result by extracting some features from the packed fire executable. In particular this paper proposes a *"packer identification method based on features extracted only from the code sections of binary executable files. Because the characteristics of code sections cannot be easily changed by malware developers [7]"*. The value of entropy and the frequency of bytes in the code section is used to extract these features.

"The Encrypted Section Extraction process identifies each section based on the header information of a packed executable file. After identifying sections in an executable file, the entropy scores of each section are calculated, and the section with the highest score is selected as an encrypted section. In the Feature Extraction process, features are extracted from the binary sequences of the encrypted section [7]".

After extracting the features, they are analyzed with machine learning algorithms to analyze and identify the type of packer used and finally the results are evaluated to calculate the accuracy of the identification divided for each different packer used.

An overall process overview can be seen in the following image:



[Packer identification method based on byte sequences [7]]

The first step of the process is to identify the section encrypted by the packer and this is done by calculating the entropy. Usually, PE files contain the name of the encrypted section as the table below shows. But this information is easily modified by malware writes to make the identification more challenging reason why the entropy is calculated in order to detect the encrypted section.

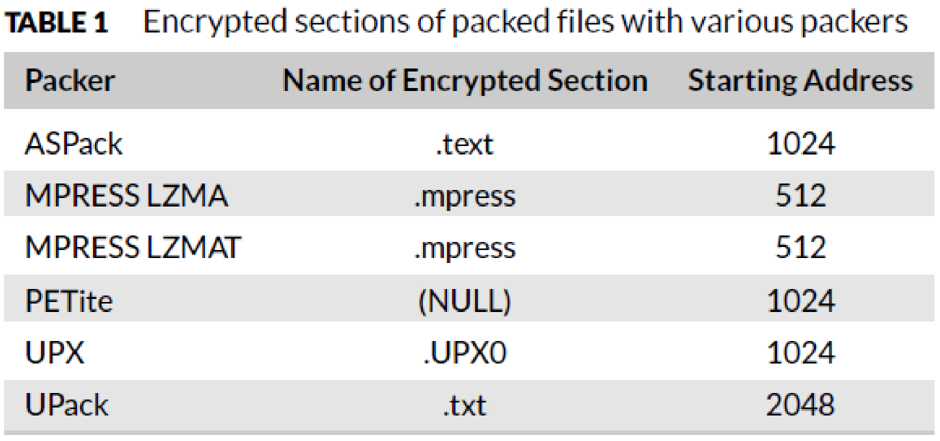


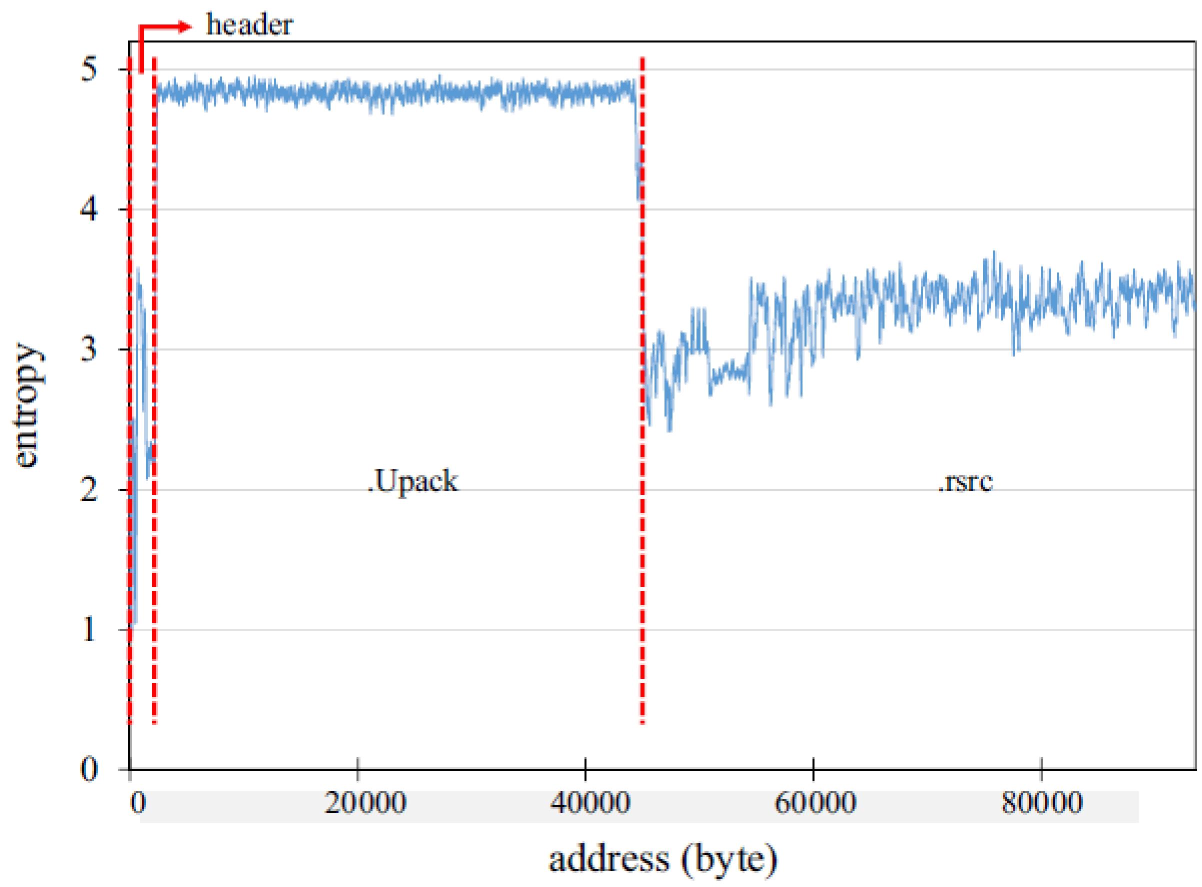
TABLE 1 Encrypted sections of packed files with various packers

Packer	Name of Encrypted Section	Starting Address
ASPack	.text	1024
MPRESS LZMA	.mpress	512
MPRESS LZMAT	.mpress	512
PETite	(NULL)	1024
UPX	.UPX0	1024
UPack	.txt	2048

[Packer identification method based on byte sequences [7]]

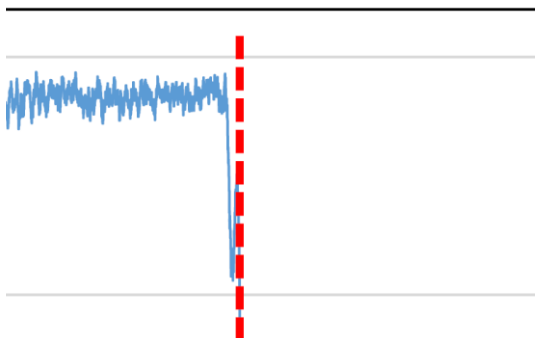
"Entropy is defined as a degree of disorder or uncertainty in a system [7]". A high number of entropies is a clear indication of encryption, and it is computed for each section of the PE

file and the encrypted section will have a higher entropy compared to other section of the executable as sown in the image below.



[Packer identification method based on byte sequences [7]]

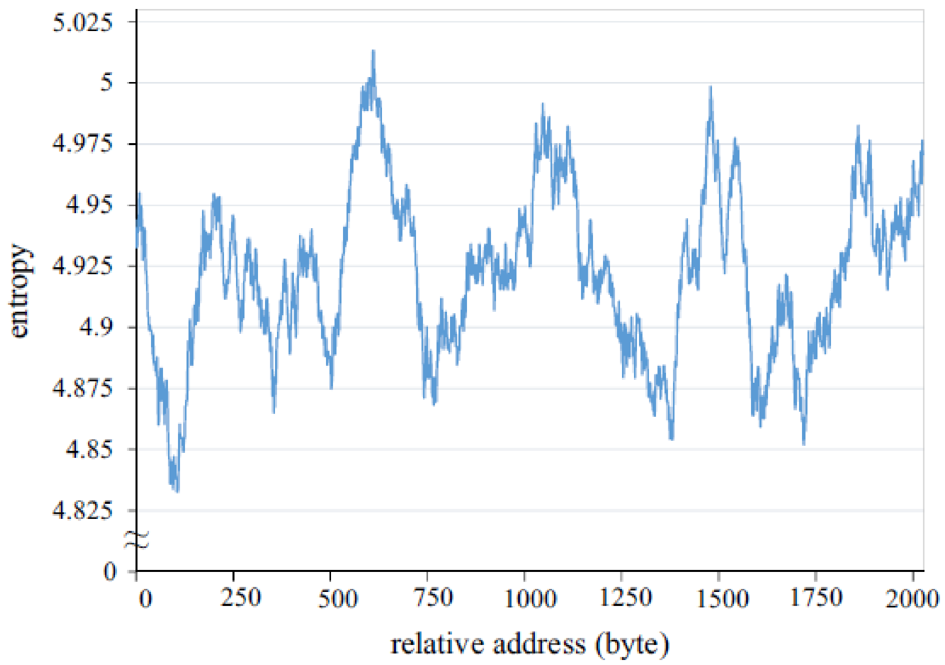
These values can be misled the final result due to the fact that "b*inary executable files can have null paddings at the end of a section, and these null paddings can affect the entropy score of the section. To mitigate the effects of null paddings, our scoring method calculates the entropy scores of byte sequences in n- byte sliding windows for a single section* [7]".



[Packer identification method based on byte sequences [7]]

In the proposed paper [7] the entropy is not only used to identify if an executable is packed or not but also to identify the packer itself because "*different packing algorithms are used by different packers; the statistical values of entropy values may also be different. Then, different statistical values can be used as features to identify different packers* [7]".

Analyzing the byte sequence of an encrypted section and extracting features can reveal the packer as the encrypted section is a list of random byte values and different packer uses different encryption algorithm, they will generate different entropy sequence. To such purpose the encrypted section is analyzed byte-by-byte. "*The entropy score sequence is defined as a list of entropy values of byte sequences within the n -byte sliding window* [7]". The following image shows an example of an executable packed with ASPack [29] and show and shows a unique pattern of entropy.



[Packer identification method based on byte sequences [7]]

The paper also analyzed different packing algorithms and was able to find concrete values to distinguish the employed packer. Thus, these features shown in the image below can be used to distinguish different packers in use.

TABLE 2 Statistical values of entropy scores

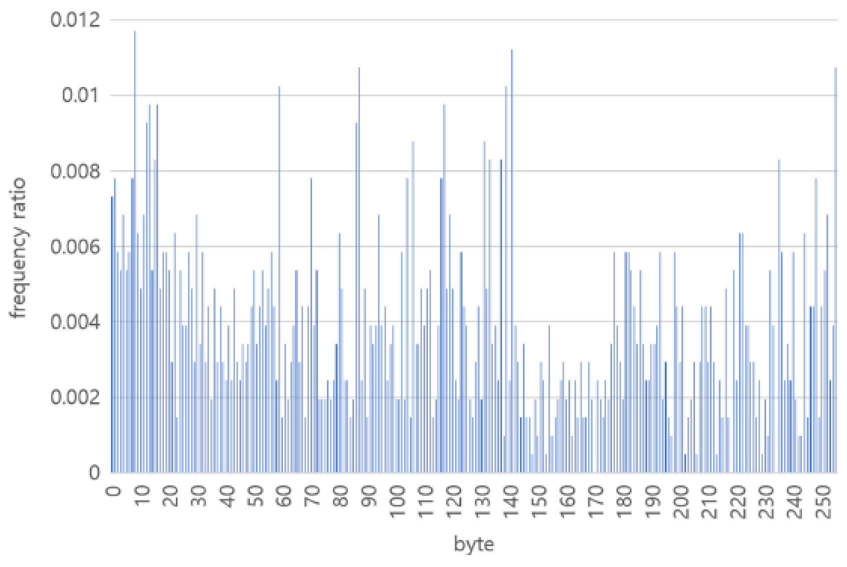
Packer	std	average	max	min
ASPack	0.04324	4.94923	5.04480	4.82456
MPRESS LZMA	0.06839	4.85957	4.98586	4.66865
MPRESS LZMAT	0.11468	4.67481	4.86365	4.35886
PETite	0.06291	4.91712	5.02858	4.72991
UPack	0.40392	4.19520	4.65096	3.03325
UPX	0.08173	4.79295	4.93875	4.54831
Normal files	0.29876	3.57985	4.12918	2.94382

[Packer identification method based on byte sequences [7]]

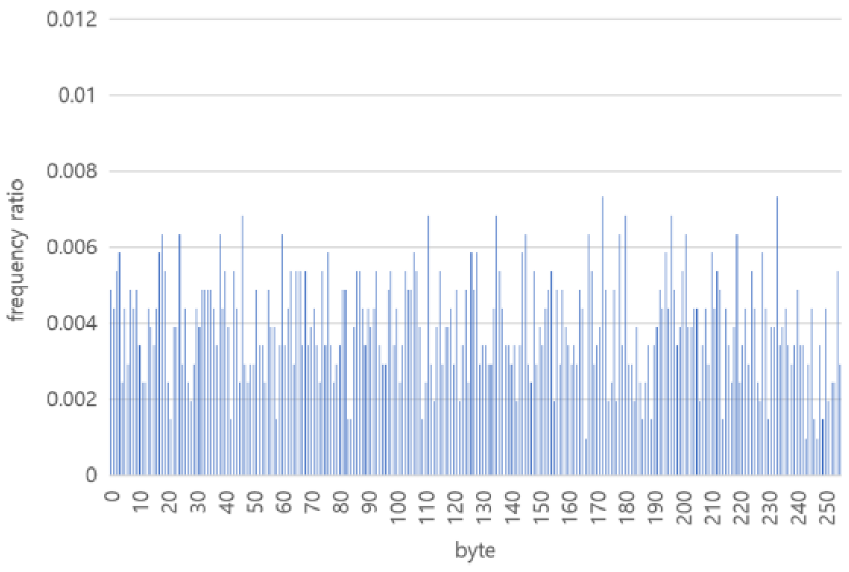
Until now we have seen one of the two proposed technique, feature extraction form entropy. Now let's see the second technique which is the analysis of the byte sequence and how it can reveal the packer.

The strategy is to analyze all the bytes of the sections, the byte values between 0x00 and 0xFF and calculating the number of occurrences of each byte to generate the frequencies. One of the problems during this phase is that the byte 0x00 is also used as padding at the end of each section so all the bytes 0x00 that are at the end of a section will be ignored to generate an accurate result.

In the paper [7] two particular packers have been studied, UPX and ASPack and they generated respectively the following frequency graph:



[Packer identification method based on byte sequences [7]]



[Packer identification method based on byte sequences [7]]

From the frequency analysis we can clearly distinguish the two packers in use and incorporate such information during the analysis process. Such information often is crucial as it may save time and speed up the analysis phase which is often time consuming for unknows or custom packers.

7 - Advanced Packers

So far, we have seen and analyzed very simple packers, it is important to describe the behaviour of more advanced packers to understand how far they can go. For this reason, I have chosen a very advanced packer to protect Windows software, Themida. Themida is a commercial packer, it implements state-of-the-art security measures to make it difficult and complicated to analyze.

The goal, in fact, is to avoid decompilation, and Themida, for this, falls well within the category of software protector. The problem that Themida effectively solves is that the techniques employed by classic packers are all mostly known, and they can be unpacked. The executable is first decompiled and broken down into portions, each of which is protected, and then reassembly and a second phase of protection of the entire file is performed, resulting in the final executable.

All processes are managed by an architecture based on virtual micro-machines (*"a virtual machine program that serves to isolate an untrusted computing operation* [8]"), each dedicated to real-time encoding and decoding of the pieces into which the executable is decomposed. Code virtualization, in Themida, can be managed in a granular manner, selecting the type of virtualization based on parameters such as execution speed (which is more or less affected depending on the technology chosen) and the size of the executable. And based on these parameters that the complexity of decoding and analysis of the processed files varies.

Below are few features of Themida that are noteworthy:

1. Anti-API scanners techniques that avoid reconstruction of original import table.
2. Anti-memory dumpers techniques.
3. Anti-debugger techniques that detect/fool any kind of debugger.
4. Multiple polymorphic layers with more than 50.000 permutations.

The Themida packer is a highly advanced and intricate tool used to protect software from reverse engineering and analysis. It incorporates multiple anti-analysis and evasion techniques, making it challenging to analyze using simple methods.

Another very famous packer among malware writers' worth mentioning is Armadillo. When Armadillo is run it creates two separate processes where the first process acts as a debugger to the second to catch exceptions during execution. When the program attempts to execute the encrypted code, an exception is triggered because the required code is not present in the memory in its decrypted form. This exception is caught by the debugger process, which is monitoring the execution of the program. The debugger process then takes control and decrypts the required page of memory, making the decrypted code available for execution. This decrypted code is loaded into the memory of the program, replacing the encrypted version. As a result, the program can now execute the previously protected code.

To successfully understand and analyze complex packers like Themida and Armadillo, we need to employ more advanced strategies and techniques. In the next chapters, we will explore these advanced approaches, enabling us to effectively analyze and comprehend the inner workings of complex packers.

7.1 - Advanced Packing Techniques

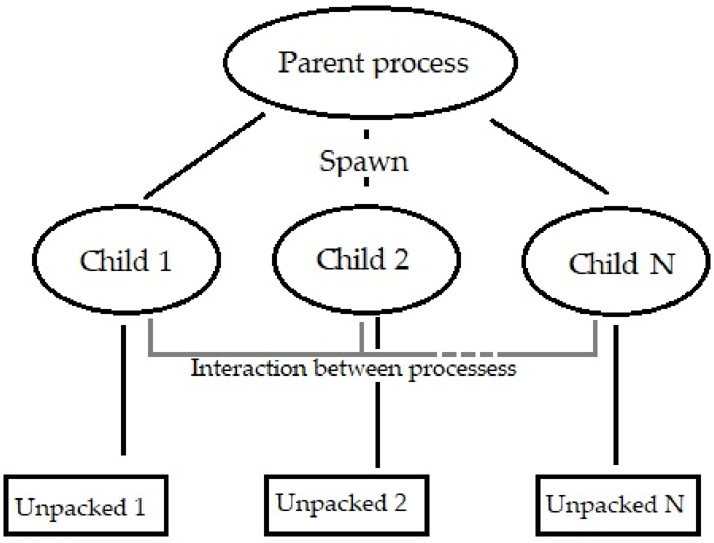
As can be seen, such a complex packer that implements several anti-analysis and evasion techniques cannot be analyzed in a simple way as seen in the previous examples and they require different strategies. The following chapters will discuss advanced strategies for analyzing and understanding the implementation of packed malware with advanced packing strategies.

"More complex packers often involve several layers of unpacking routines, in which the first layer unpacks the second one, which in turn unpacks another routine, until the original code is reconstructed at the end of the chain. Others employ several parallel processes, they interleave the execution of unpacking code with the original code, or even incrementally unpack and re-pack the code on-demand before and after its execution [93]*"*.

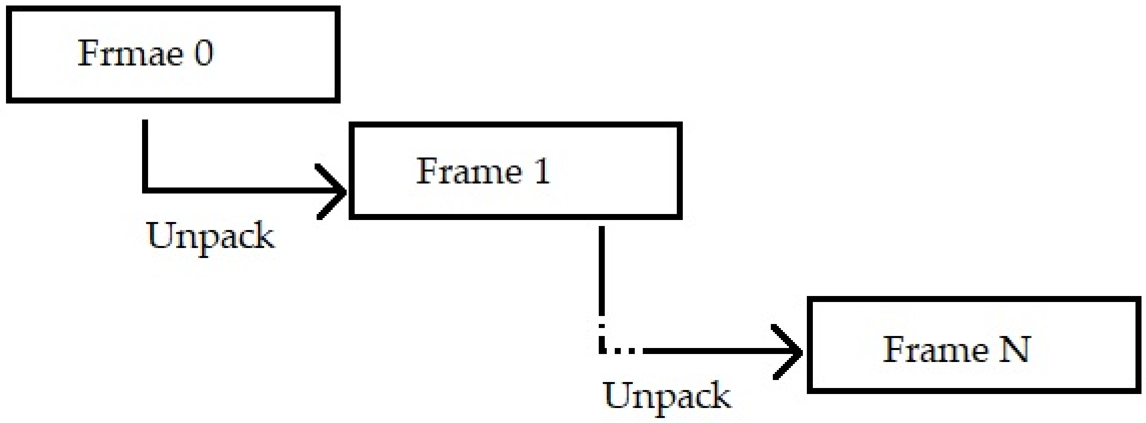
Following a subset of advanced unpacking techniques presented in the paper "SoK: Deep Packer Inspection: A Longitudinal Study of the Complexity of Run-Time Packers":

1. Parallel Unpackers
2. Unpacking Frames
3. Code Visibility

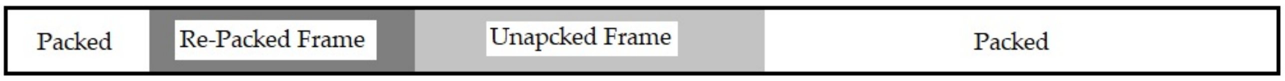
Parallel Unpackers is a strategy where "*packers employ several processes in order to unpack the original code* [9]". These processes can interact with one another to perform various checks such as synchronization or time measurement to avoid detection and analysis.



Unpacking Frame *"is a region of memory in which we observe a sequence of a memory write followed by a memory execution. Traditional run-time packers have one unpacking frame for each layer, because the code is fully unpacked in one layer before the next layers are unprotected. We call these packers single-frame unpackers. However, more complex cases exist in which the code of one layer is reconstructed and executed one piece at a time. These cases involve multiple frames per layer and are called multi-frame packers in our terminology [9]"*.



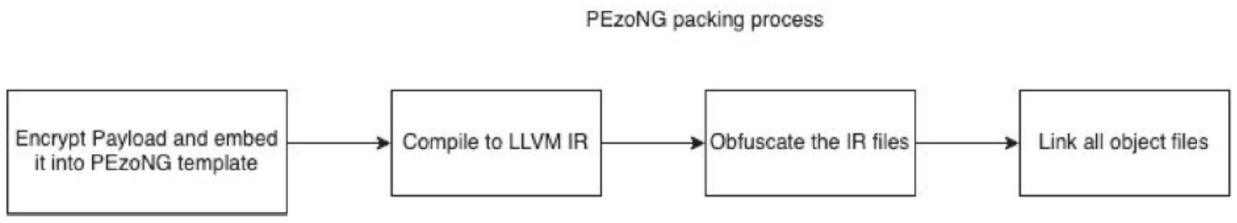
Code Visibility is an "*advanced multi-frame examples exist that selectively unpack only the portion of code that is actually executed. This approach is used as a mechanism to prevent analysts and tools from easily acquiring a memory dump of the entire content of the binary [9]*". There are different types of unpacking mechanism such as: Full -Code Unpacking which is a routine that unpack the entire code and executes it. Incremental Unpacking is an approach where the code in unpacked, if necessary, before being executed but not all the code are unpacked. Shifting-Decode Frames is a strategy where only one frame at a time is visible and unpacked and the previews unpacked frame are repacked to reduce the visibility.



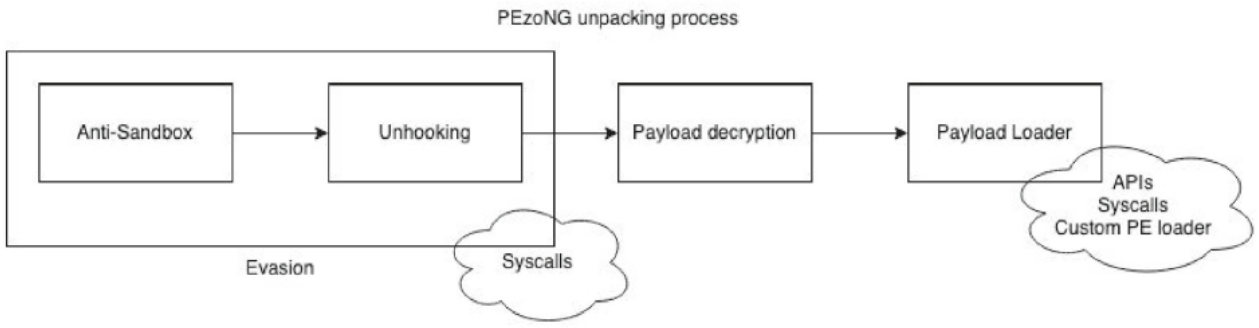
7.2 - Deep Dive into PEzoNG and Advanced Malware Packer

PEzoNG is a packer for Windows that implements various evasion techniques and anti-analysis mechanism. *"PEzoNG is built with modularity in mind and allows to add new features in a simple way by adding new modules that could implement different techniques with a fine grained detail [10]"*. In this way it is possible to add new techniques and implement additional feature to add to the packer and develop new evasion strategies. PEzoNG is a polymorphic packer which means that its signature changes from computation to computation which makes Anti Viruses that checks for trivial signature ineffective.

The image below shows an overview of the packing and unpacking overview of PEzoNG:



[Advanced Packer For Automated Evasion On Windows [10]]



[Advanced Packer For Automated Evasion On Windows [10]]

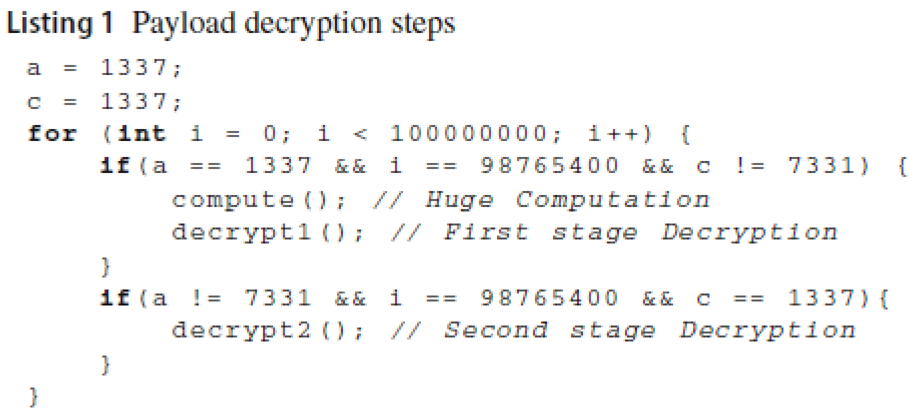
PEzoNG includes the following modules:

1. Encryption
2. APIs
3. Syscall
4. Evasion
5. PE Loader
6. Shellcode Injeciton

Encryption/Decryption Strategy

PEzoNG uses a 256bit key and a python script t encrypt the malware and embeds the key in the header and different encryption algorithm can be selected. Also, the decryption happens in two stages as the encryption in order to avoid detection by Anti Viruses. As Anti Viruses can choose to run always one branch of if-else statements by changing the condition implementing the decryption accordingly can avoid the extraction of the malware. Antiviruses also tries to avoid performing high computation in order to maintain in order to maintain a smooth user experience.

The code below shows an example of a two-stage decryption with high computation:


Listing 1 Payload decryption steps

```
a = 1337;
c = 1337;
for (int i = 0; i < 100000000; i++) {
    if(a == 1337 && i == 98765400 && c != 7331) {
        compute(); // Huge Computation
        decrypt1(); // First stage Decryption
    }
    if(a != 7331 && i == 98765400 && c == 1337){
        decrypt2(); // Second stage Decryption
    }
}
```

[Advanced Packer For Automated Evasion On Windows [10]]

The two if-else statement with the addition of a huge computation inside one of the branches prevent AVs from evaluating all the branches and the successful decryption of the malicious code.

APIs

As malware packer uses several Windows APIs, many of them can be potentially flagged by security software as malicious and prevent a successful infection as their names are present in the IAT. To avoid such PEzoNG "*implemented an automatic functioncall obfuscation method which allows to dynamically resolveany Windows API address at run-time without ever specifying the API name* [10]". The strategy consists in computing the hash value of the API names, the hash of a specific API then is checked with a computed hash of the function in the Process "Environment Block" that contains the names of the needed APIs.

"Since all the APIs used by PEzoNG belongs to two dynamic libraries ntdll.dll and kernelbase.dll which are always loaded by Windows into every process address space, all of them can be correctly

resolved at run-time [10]". In this way the API names never appear in the PE file, and it avoids using "LoadLybrary" and "GetProcAddress" which are indicator of packing.

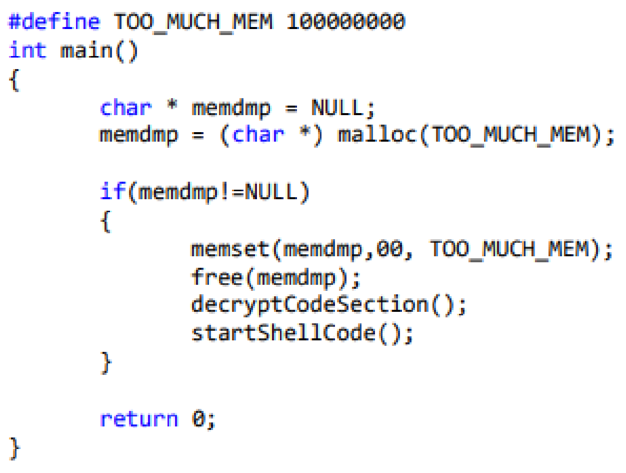
Systemcalls and Evations

PEzoNG, for some operations, invoke directly the system calls without relying on APIs with the help of "*Syswhispers2* [11]". For example, the API call VirtualAlloc uses the system call NtAllocateVirtualMemory and using directly the NT syscall avoid security software from user space hooking. This is possible due to the fact that "*in user-space Zw* functions (kernel-level functions) are at the same address of the corresponding Nt* function (user-level function)* [10]". Reordering by address the Zw* function will provide the system call number which is needed to perform the call.

Anti-sandbox

PEzoNG make use of "*Offer you have to refuse* [12]" method. This method consists of using large number of resources such as memory or CPU, and as AVs have to analyze huge number of files, they cannot use cush amount of time and resources for a single file as it may lead to Denial of Service on the AV.

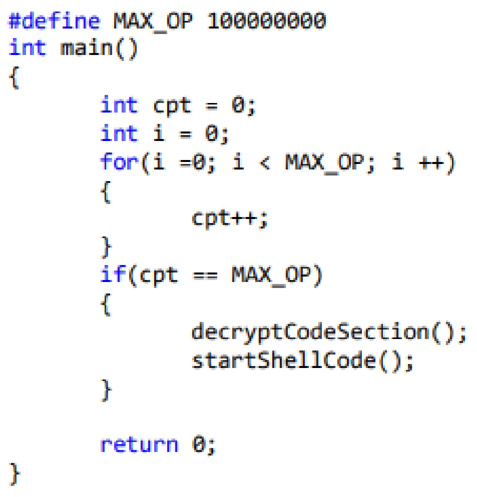
The following code are an example for memory and CPU usage:



```
#define TOO_MUCH_MEM 100000000
int main()
{
        char * memdmp = NULL;
        memdmp = (char *) malloc(TOO_MUCH_MEM);

        if(memdmp!=NULL)
        {
                memset(memdmp,00, TOO_MUCH_MEM);
                free(memdmp);
                decryptCodeSection();
                startShellCode();
        }

        return 0;
}
```



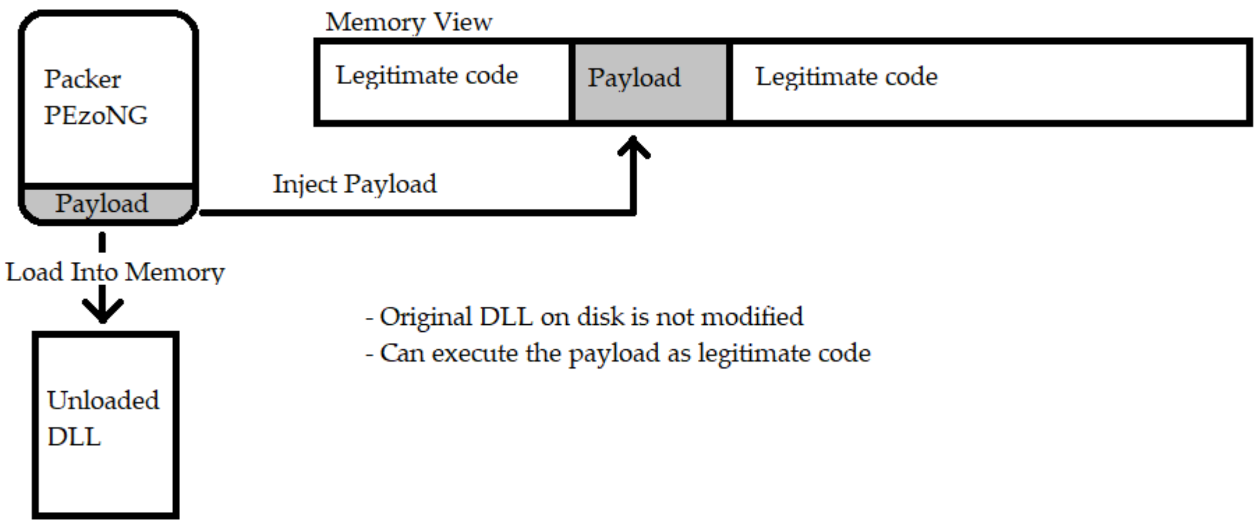
```
#define MAX_OP 100000000
int main()
{
        int cpt = 0;
        int i = 0;
        for(i =0; i < MAX_OP; i ++)
        {
                cpt++;
        }
        if(cpt == MAX_OP)
        {
                decryptCodeSection();
                startShellCode();
        }

        return 0;
}
```

[Bypass Antivirus Dynamic Analysis [12]*]*

Main Loader and Shellcode Injection

Once the malicious code has been decrypted the main loader prepare it for the execution. In the event of a shellcode injection the packer employs the "Phantom DLL Hollowing [34]" technique. "*Phantom Dll Hollowing looks for a dll on disk that has not already been loaded into memory and that is large enough to host the malicious payload. Once a feasible Dll is found, the loader opens the Dll using a transacted NTFS (TxF)* [34]". Phantom DLL Hollowing maps the victim DLL into the memory and inject the payload keeping intact the rest of the library.



With this technique PEzoNG make the payload stealthier and harder to detect as the payload is treated as a legitimate code.

To understand better this process, it is essential to introduce what Transacted NTFS is. Transacted NTFS is a feature of Windows that groups several operations in one single unit. If one of these operations fails all the others are considerate unsuccessful. In the case of PEzoNG it uses the Transacted NTFS feature to open the target DLL and inject the shellcode in the memory without modifying it in the disk and then the changes can be rolled-back without affecting the original file in the disk. In this way the injection is made only in the memory and not save in the disk.

Results

The following image shows the employed strategies against a few antiviruses if they can detect if the binary is malicious or not.

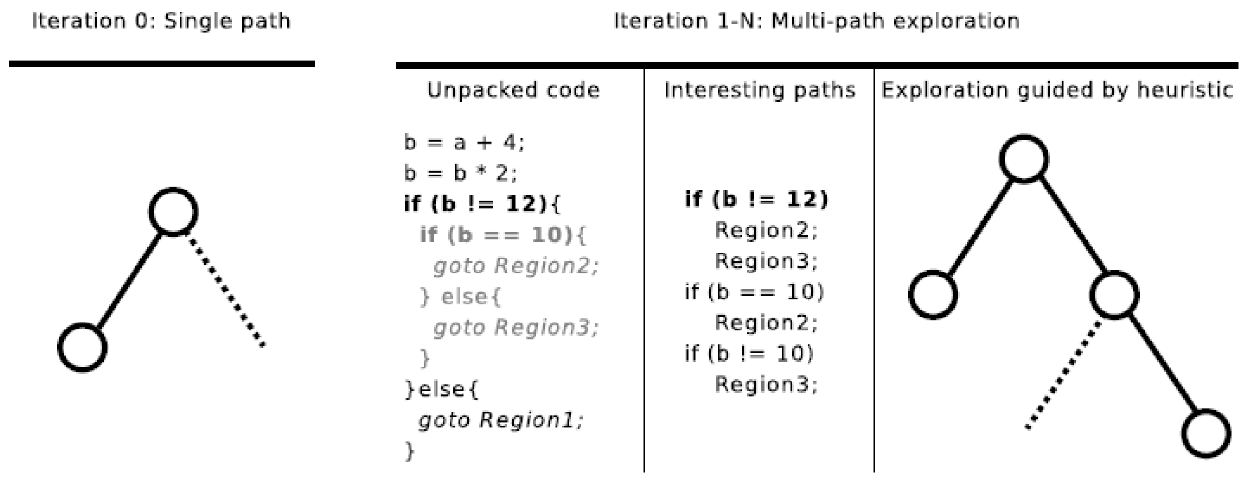
Antivirus	Raw	Encryption	Syscall	Dll Hollowing	Anti-sandbox	Unhooking	All(=+Obfuscation)
Defender	✓	×	×	×	×	×	×
AVG	✓	×	×	×	×	×	×
BitDefender	✓	✓	✓	✓	×	×	×
MalwareBytes	✓	✓	✓	✓	×	×	×
Norton 360	✓	✓	✓	✓	✓	×	×
ESET Int. Sec.	✓	✓	✓	✓	×	×	×
McAfee	✓	✓	✓	✓	×	×	×

[Bypass Antivirus Dynamic Analysis [12]]

As we can see the PEzoNG was undetected by the antiviruses when employing unhooking technique and a combination of all of them. In other cases, a subset of antiviruses was able to detect the executable as malicious. We can state that PEzoNG is able to successfully pack malware and pass undetected by most of the antiviruses in commerce.

8 - A Possible solution for Advanced Packers Analysis

Multiple branch observation is a technique described in the paper "RAMBO: Run-time packer Analysis with multiple-branch observation [13]" that tries to solve code partitioning problems efficiently. This technic is particularly effective when malware implements code frames shifting techniques. *"These packers only reveal one code region at a time, decrypting only the code covered by a single execution path [13]"*. This technique can also be used for advanced unpacking mechanisms implemented. The paper also discusses about the efficiency of the multiple path exploration the protected regions as it may generate overhead.

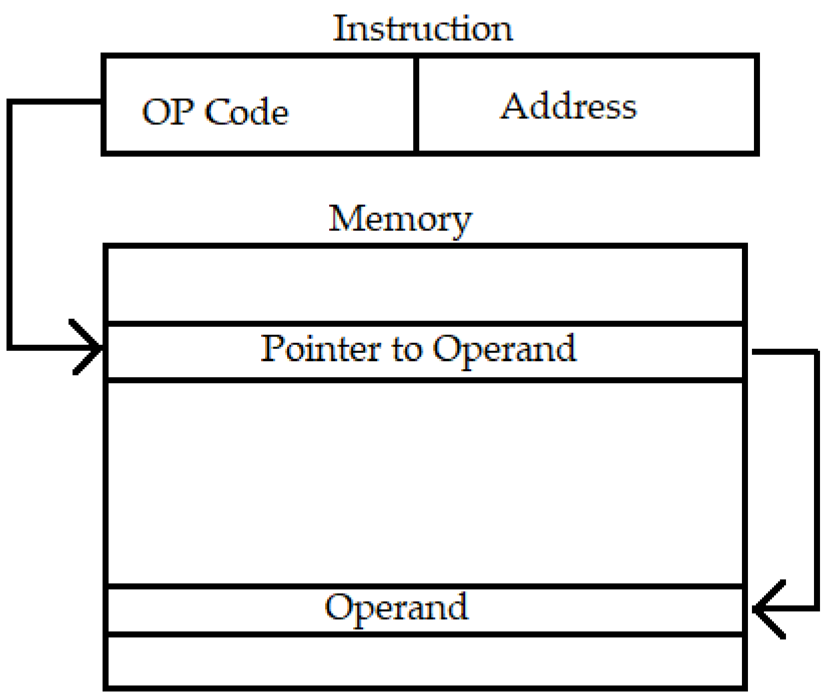

Iteration 0: Single path
Iteration 1-N: Multi-path exploration
Unpacked code
Interesting paths
Exploration guided by heuristic

```
b = a + 4;
b = b * 2;
if (b != 12){
  if (b == 10){
    goto Region2;
  } else{
    goto Region3;
  }
}else{
  goto Region1;
}
```

```
if (b != 12)
    Region2;
    Region3;
if (b == 10)
    Region2;
if (b != 10)
    Region3;
```

"*This approach leverages dynamic trait analysis, symbolic execution and process snapshotting in order to explore execution path in depth first order* [13]". The strategy mentioned is executed in two steps. In the first step we execute the malware sample in order to visit a single path then we analyze the packer to identify the region where the code has been unpacked. In the second step we analyze the extracted code in order to find the information on how to execute a different path for a new portion of the code to be unpacked. We iterate these two steps until the packer finishes to extract all the portions of the code.

"*Symbolic execution allows to evaluate a program over a set of symbolic inputs instead of concrete values. A constraint solver can evaluate the symbolic expression that must be satisfied to follow a given path, providing an appropriate set of values for each input variable* [13]". Through the symbolic execution we can execute all the paths of the malware. But the problem of accessing the memory address remains as packers utilize indirect memory access (as shown in the image below) as an anti-analysis measure to hinder static and dynamic analysis techniques. By dynamically generating memory addresses or using complex memory access patterns, packers make it challenging for analysts to understand and track the packer's behaviour.



Sometimes it is necessary to stop the execution in a given time for example before a conditional jump or a change of a value, for this purpose it is really useful performing system snapshots. By stopping the execution at given time it will be possible to redirect un an unexecuted branch of the program. This approach has many limitations as the operating system must save different data structure and may incur in error restoring the previews state.

The last approach mentioned in the paper which is applied for Shifting-Decode-Frames is to identify the stub, in charge of encrypting and decrypting portion of codes. The strategy proposed is to apply the multiple path exploration only on the protected code avoiding the stub and the anti-analysis routines which are out of our interest. To achieve this, we must identify the first decrypted portion of the code and from analyzing it we may find hints how to explore the next path and decrypt the next portion of the code.

The paper proposes optimization that simplify the multiple path observation to unpack binaries. For example, one of the optimizations proposed is the inconsistent multipart exploration which consist in forcing, which is consistent with the last tainted jump, to void crushes before completing examining the code and, inconsistent with the previews path restriction. This strategy overcome the challenge when packers modify the way how a program executed and make it difficult to explore its branches as standard symbolic execution certain path may not be executed. In addition, to reduce the size and the overhead of the exploration of the various paths Multiple Branch Observation limit the explored code only to regions that may contain the original code of the malware.

To further reduce the overhead and the complexity is to consider the local and the global consistency it is necessary to apply logical and global consistency. "*This means that we respect the consistency within the regions that contain the original code of the binary, but we allow the variables in this region to adopt values that are inconsistent with the rest of the code* [13]". The global consistency maintains coherence and compatibility across different parts of the code. It ensures that the findings and conclusions derived from analyzing various sections of the software align with each other and form a coherent understanding of the overall behaviour of the program.

These optimizations allow complex malware to be analyzed in a reasonable time and limited to only the portions of interest. Doing so reduces the analysis overhead to have greater accuracy in understanding the behaviour and extraction of encrypted portions from the packer. The multiple branch observation if used on malware packed with Themida as described above would greatly facilitate its analysis that would otherwise be extremely complex and time-consuming. additionally limiting the analysis to only the interested portions and thus avoiding analyzing code that deals with for example, code that does not unpack subsequent portions would further speed up the process.

9 - Classification of Packed Files

9.1 - Introduction

To solve the problem of malware classification and its variant a system named Malwise [14] has been developed. This system classifies malware "*using a fast application-level emulator to reverse the code packing transformation, and two flowgraph matching algorithms to perform classification. An exact flowgraph matching algorithm is employed that uses string-based signatures and is able to detect malware with near real-time performance [14]*".

The innovative technique that Malwise uses is the use of structuring and decompilation to generate malware signature. Structuring a malware consists in rearranging the code in a more understandable manner which helps to understand patterns and logical flow of it. Decompilation is used to convert the low-level code to a pseudo higher level for a better understanding of it and analysis.

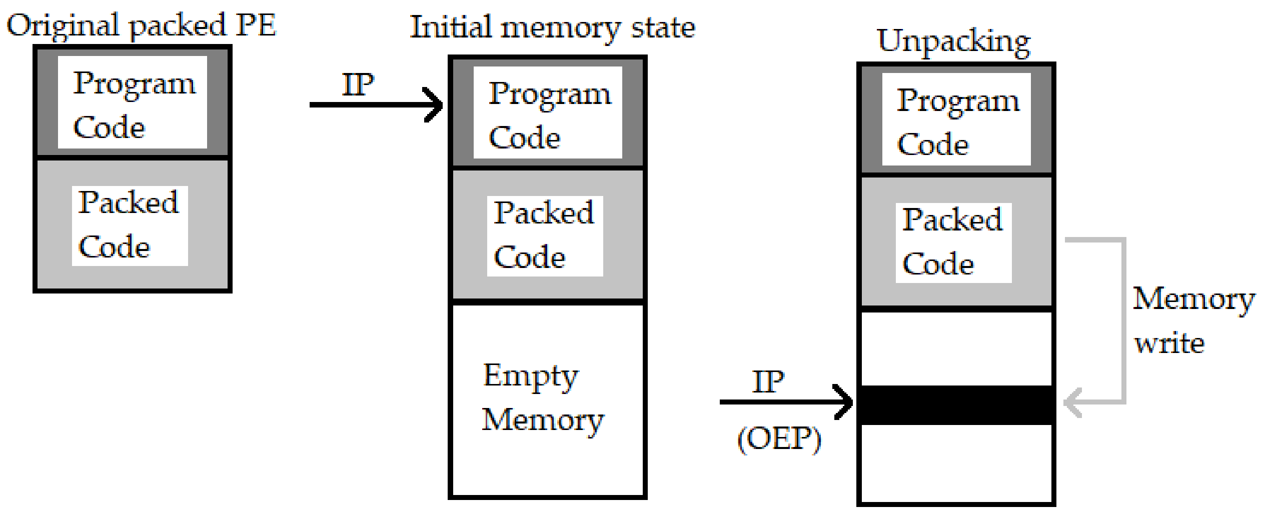
The approach of Malwise to generate the signature is divided in two parts and they consist of "Exact Matching", Malwise look for the exact match of a control flow graph from a malware database to classify the input binary, and "Approximate Matching" Malwise search for similarity between the input flow graph and the database to classify binaries respecting a given threshold.

9.2 - Detecting Packing

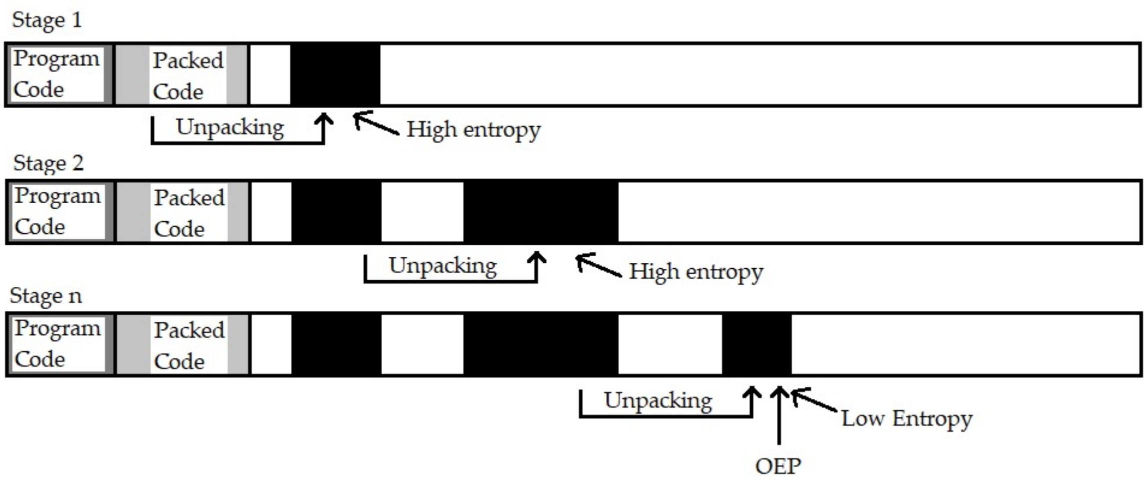
The first step is to understand if the malware employs packing techniques. This is done by computing the entropy of the input file as a packed file most like have high entropy compared to a non-packed one. "*If the binary is identified as being packed, then the dynamic analysis to perform automated unpacking proceeds. If the binary is not packed, then the static analysis commences immediately [14]*".

9.3 - Detect Code Extraction (Unpacking Completion)

Malwise detects the OEP by tracking the memory writes [15]. The idea behind this technique discussed in Renovo [15] is that regardless the employed packing technique eventually the original code should be preset in the memory to be executed and the instruction pointer must jump to the address (OEP) of the extracted program to execute it at runtime.



Malwise improved this concept in the event of multi-layer packing and can identify precisely in which state the OEP will be revealed. The idea is that *"if there exist high entropy packed data that have not been used by the packer during execution, then they remain to be unpacked. To determine if a particular stage of unpacking represents the OEP, the entropy of new or unread memory in the process image is examined. Newly written memory is indicated by the shadow memory for the current stage being unpacked. Unread memory is maintained globally, in a shadow memory for all stages. If the entropy of the analyzed data is low, then it is presumed that no more compressed or encrypted data are left to be unpacked. This heuristically indicates completion of unpacking [14]".*

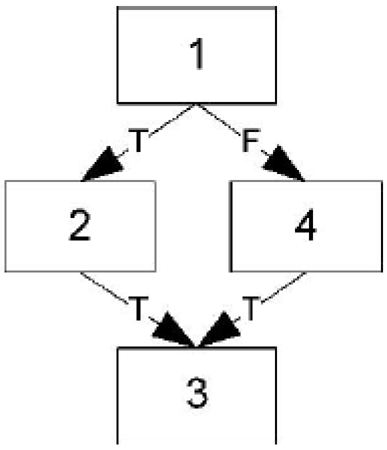


9.4 - Static Analysis

The objective of the static analysis is to extract characteristics from the unpacked binary. The characteristics are then transformed in a higher-level representation that are suitable from string matching (signature). For such a purpose Exacts Flow Graph and Approximate Flow Graph matching are employed.

9.5 - Exact Flow Graph Matching

Once the characteristics of the input binary has been extracted a sigher level of control flow graph is created and its signature is also generated. The signature is then compared with the database of malware signatures (consisted of graph edges and ordered nodes) in search of isomorphism. The figure below shows and example of control flow graph and the signature of it.



```
(1 -> 2), (1 -> 4)
(2 -> 3), ()
(), ()
(4 -> 3), ()
```

[*An Effective and Efficient Classification System for Packed and Polymorphic Malware* [14]]

9.6 - Approximate Flow Graph

Using the approximate flow graph is possible to identify a greater number of malware variants compared to the previews matching technique. In this process structuring is used to generate signatures to compare the similarity with the signatures in the database. *"Structuring is the process of recovering high-level structured control flow from a control flow graph. The intuition behind using structuring as a signature is that similarities between malware variants are reflected by variants sharing similar high-level structured control flow* [14]*"*.

The reason of performing structuring of code is because malware variants have similarities at a higher-level representation of their structure and control flow. Structuring produces a string of characters representing the higher-level representation of the malware code block. Following an example of the signature generation:

```
int main() {    B
   int x = 5;   B
   int y = 10;  B

   if (x < y){  I
      printf("x is less than y.\n");   B
   } else {     E
      printf("x is not less than y.\n");      B
   }

   return 0;    R
}
```

Signature: BBBI{I{B}E{B}}R

Malwise uses a database containing the signature generate from flow graphs of known malware in string format. To classify an input binary the signature is then compared with the database in search of similarities. *"For exact matching, the assignment is based on string equality. For approximate matching, a greedy assignment is made for the best approximate matching string where the similarity ratio is above 0.9* [14]*"*.

9.7 - Similarity

The greedy match of 0.9 threshold is used to identify potential exact matches and in the final classification a threshold of 0.6 is used to identify malware variants. Both thresholds are used in the result for malware identification and a lower threshold is used for a broader match for flexibility and capture even slightly modified versions of malware.

The problem that arises from this approach is the match with several malwares. For such reason the similarity check process is divided in two stages. The stage is the search for uniqueness, the input malware is compared with the database to check if it's a unique malware or has been already seen previously by comparing its extracted features. In the second step the remaining duplicated are compared with a lower threshold of 0.6 in search of similarities and malware variants. Only the exact matches or the matches above 0.6 are returned as a result of the searching process.

9.8 - Evaluation

The following tables shows the accuracy of Malwise in identifying the OEP of binaries (HOSTNAME.EXE and CALC.EXE) packed with different packers.

Name	Revealed code and data	Number of stages to real OEP	Stages unpacked	% of instr. to real OEP unpacked
upx	13107	1	1	100.00
rlpack	6947	1	1	100.00
mew	4808	1	1	100.00
fsg	12348	1	1	100.00
npack	10890	1	1	100.00
expressor	59212	1	1	100.00
packman	10313	2	1	99.99
pe compact	18039	4	3	99.98
acprotect	99900	46	39	98.81
winupack	41250	2	1	98.80
telock	3177	19	15	93.45
yoda's protector	3492	6	2	85.81
aspack	2453	6	1	43.41
pepsin	err	23	err	err

hostname.exe

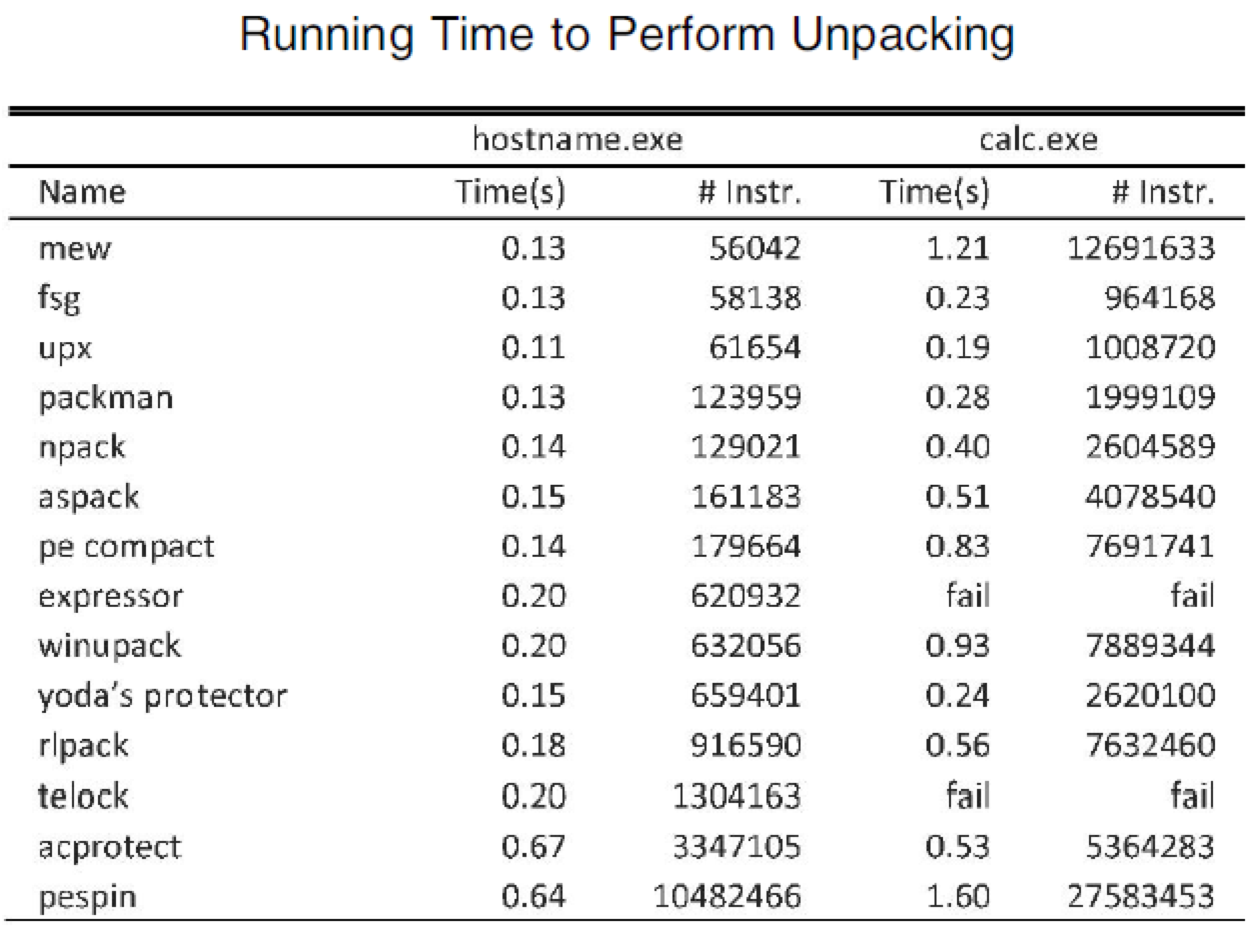
Name	Revealed code and data	Number of stages to real OEP	Stages unpacked	% of instr. to real OEP unpacked
upx	125308	1	1	100.00
rlpack	114395	1	1	100.00
mew	152822	2	2	100.00
fsg	122936	1	1	100.00
npack	169581	1	1	100.00
expressor	fail	fail	fail	fail
packman	188657	2	1	99.99
pe compact	145239	4	3	99.99
acprotect	251152	209	159	96.51
winupack	143477	2	1	95.84
telock	fail	fail	fail	fail
yoda's protector	112673	6	3	95.82
aspack	227751	4	2	99.90
pespin	err	23	err	err

calc.exe

[*An Effective and Efficient Classification System for Packed and Polymorphic Malware* [14]]

From the table we can see that Malwise has a great accuracy in finding the OEP even in cases of several layer of several layer of packing. Malwise failed the unpacking of CALC.EXE only when packed with “napck”, “expressor”, “telock”, and “pespin”. It failed 4 out of 14 packer which is an impressive result, and it was able in most of the cases to reveal more than 98% of the packed instructions.

The performance evaluation shows that thanks to the immersive speed of Malwise, as shown below, is suitable to be integrated into desktop anti malware systems.



Running Time to Perform Unpacking

	hostname.exe		calc.exe	
Name	Time(s)	# Instr.	Time(s)	# Instr.
mew	0.13	56042	1.21	12691633
fsg	0.13	58138	0.23	964168
upx	0.11	61654	0.19	1008720
packman	0.13	123959	0.28	1999109
npack	0.14	129021	0.40	2604589
aspack	0.15	161183	0.51	4078540
pe compact	0.14	179664	0.83	7691741
expressor	0.20	620932	fail	fail
winupack	0.20	632056	0.93	7889344
yoda’s protector	0.15	659401	0.24	2620100
rlpack	0.18	916590	0.56	7632460
telock	0.20	1304163	fail	fail
acprotect	0.67	3347105	0.53	5364283
pespin	0.64	10482466	1.60	27583453

[*An Effective and Efficient Classification System for Packed and Polymorphic Malware* [14]]

10 - Packers on Embedded Systems

10.1 - Embedded Systems and IoT

With the emergence of IoT and Embedded Systems, malicious users have adapted to exploit these technologies for malicious purposes. In this regard, it is important how much and how packing impacts them.

The difference between Embedded System and IoT lies in connectivity. That is, an IoT device is connected to the Internet e.g., a washing machine or refrigerator that can offer real-time information. On the other hand, an embedded system is like a minicomputer build for a specific use case, equipped with minimal computing power, and it is not connected to the internet.

Normally an embedded system does not have an Internet connection, but an IoT embedder does, so an embedded system can be considered a subset of IoT. So, these types of devices can be used by malicious users to carry out further attacks or perform packing and unpacking of malware.

Theoretically it is possible to carry out cyber-attacks from a compromised IoT device, its only limitation would be the available device resources such as processing power and memory. More specifically if we talk about packing "*While packing, in general, helps reduce the size of the stored binary, the unpacking/decryption time increases the total instructions executed and leads to increased power consumption* [56]".

10.2 - Performance Evaluation

The study experimented with two packers, AES Encryption and UPX, with the addition of a simple anti-debugging feature, time delay, "*when single-stepping a process through a debugger the wall clock differences between any two points will grow dramatically when compared to a native execution. Time based anti-debugging software can perform two time checks and infer that it is running under such conditions if unusual difference is detected* [56]".

The experiment was conducted against the embedded board, which consists of Intel XScale PXA255 400MHz CPU, 128MB SDRAM, 32MB Flash ROM, Embedded Linux (Kernel 2.4.19) and g++ was used to compile the executables. Its result proved that UPX several times faster that AES because of the AES decryption overhead also "*UPX is better than AES-encryption packing for reducing the startup time and invocation time on embedded systems with limited resources* [56]".

Regarding power consumption (W=V·I·t=Volt·Ampere·milliseconds), it was not possible to keep the electric current stable so there is a margin of error of ±2%. UPX showed very low power consumption compared with AES. In this regard, AES can be optimized with power consumption reduction in mind.

The two programs binary that have been choose for these experiments are MD5 hashing algorithm and AES encryption algorithms. Their execution time and power consumption have been measured in different states such as with no packing, with UPX packing and AES packing. The table below shows the data from the experiment and the difference in the measurements.

TABLE III. POWER CONSUMPTION OF MD5

	Original executable	Packing with Secure UPX	Packing with AES
Avg. Execution time (ms)	55.450	59.836	1139.100
Avg. Voltage (V)	5	5	5
Avg. Current (amp)	1.526 (1.310)	1.520 (1.310)	1.530 (1.320)
Total Power Cons. (W)	423.084	454.754	8714.115
Increase ratio		7.5%	2059.7%

TABLE IV. POWER CONSUMPTION OF AES

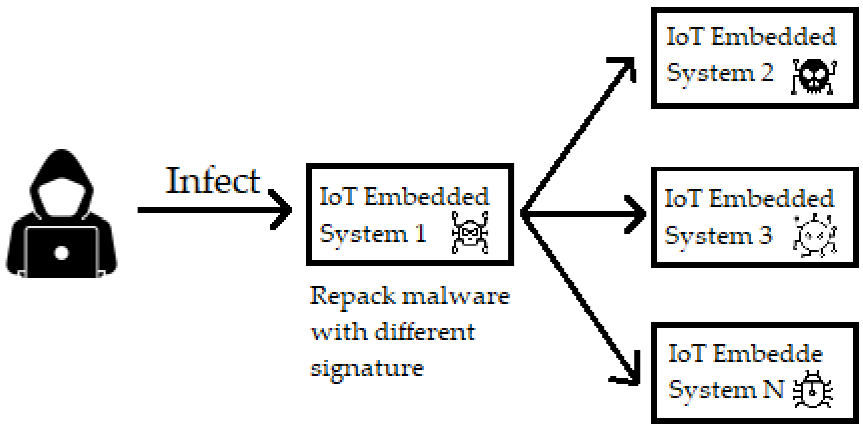
	Original executable	Packing with Secure UPX	Packing with AES
Avg. Execution time (ms)	50.730	53.786	853.700
Avg. Voltage (V)	5	5	5
Avg. Current (A)	1.463 (1.270)	1.470 (1.270)	1.466 (1.260)
Total Power Cons. (W)	371.090	395.327	6257.621
Increase ratio		6.5%	1686.3%

[*Design and Performance Evaluation of Binary Code Packing for Protecting Embedded Software against Reverse Engineering* [51]]

10.3 - IoT Embedded System for malware distribution

IoT Embedded Systems can be infected or used for malware distribution. Through the results obtained, it is possible to see how fast the execution of programs packed specially with UPX are. On average between the two programs (MD5 and AES), the average execution time is about 0.45sec. despite having significantly limited computation power. By compromising a device an attacker can use it to make lateral movement and infect other devices in the network.

This works particularly well in the case of polymorphic malware where each time after compromising a device it is possible to re-pack the malware so that it has a different signature to infect subsequent devices. This would increase the chances of infection and reduce detection by security systems.



10.4 – Packing Limitations

Given the reduced computational power, the packing algorithm cannot be the same as in standard computing systems. That is, the packer in an Embedded IoT system cannot implement all the evasion features and advanced anti-debugging techniques such as parallel execution and unpacking through different child processes or threads. This would drastically increase execution times. So, in this regard, malware and packers for these systems have to be crafted in a very specific way keeping in mind power consumption and execution time.

11 - Conclusions

In this thesis we studied about malwares, in particular about malware packing and unpacking. We studied the basics of packing and unpacking techniques with a practical demonstration on how to unpack UPX and MPRESS packers. The virtual analysis environment was also described in order to replicate the demonstration and guidelines has been provided to safely test malware without risking infecting the host machine.

Furthermore, more advanced packing techniques has been studied to understand their behaviour and evasion techniques. The study of such packing, packer identification and unpacking techniques shows that although malware implement evasion techniques that are hard and time consuming to analyze, with the proposed advanced analysis techniques it is possible, with a moderate percentage of success rate, to analyze, identify and extract packed code.

Such operations by automatic analysis systems are also highly time consuming, so they are not always possible to completely apply and execute in a real-world scenario. With the advancement of technology and with the development with more efficient techniques, that will reduce the identification and analysis time, it will be possible to create a substantial database to detect packers and analyze them in a reasonable time to identify malware. Only then it will be possible to apply such systems for commercial purpose or in a relatively large network where countless files must be analyzed.

A - Bibliography

[1] Nirav Bhojani, *Malware Analysis*, Research Gate, Ethical Hacking At: Nirma University, October 2014, DOI: 10.13140/2.1.4750.6889.

[2] VMWare Workstation Pro, https://www.vmware.com/uk/products/workstation-pro.html, VMWare.

[3] PEStudio, https://pestudio.en.lo4d.com/windows, Marc Ochsenmeier.

[4] Detect It Easy, https://github.com/horsicq/Detect-It-Easy, Hors.

[5] Ida Free, https://hex-rays.com/ida-free/, hex-rays

[6] Scylla, https://github.com/NtQuery/Scylla, NtQuery.

[7] ByeongHo Jung1 Seong Il Bae, Chang Choi, Eul Gyu Im, *Packer identification method based on byte sequences,* John Wiley & Sons, 23 October 2018, DOI: 10.1002/cpe.5082.

[8] Stephen J. Bigelow, https://www.techtarget.com/searchsecurity/definition/micro-VM-micro-virtual-machine, TechTarget, December 2021.

[9] Packers Xabier Ugarte-Pedrero, Davide Balzarottiy, Igor Santos, Pablo G. Bringas, *SoK: Deep Packer Inspection: A Longitudinal Study of the Complexity of Run-Time*, in 2015 IEEE Symposium on Security and Privacy, 20 July 2015, DOI: 10.1109/SP.2015.46.

[10] Giorgio Bernardinetti · Dimitri Di Cristofaro · Giuseppe Bianchi, *PEzoNG: Advanced Packer For Automated Evasion On Windows*, in Springer Nature, Journal of Computer Virology and Hacking Techniques, 7 February 2022, https://doi.org/10.1007/s11416-022-00417-2.

[11] Jackson T., *Syswhispers2*, https://github.com/jthuraisamy/

[12] Emeric Nasi, *Bypass Antivirus Dynamic Analysis*, Sevagas, 08/2014.

[13] Xabier Ugarte-Pedrero, Davide Balzarotti, Igor Santos, and Pablo G. Bringas, *RAMBO: Run-time packer Analysis with Multiple Branch Observation*, Springer, Cham, 12 June 2016, https://doi.org/10.1007/978-3-319-40667-1_10.

[14] Silvio Cesare, Student Member, IEEE, Yang Xiang, Senior Member, IEEE, and Wanlei Zhou, Senior Member, *Malwise—An Effective and Efficient Classification System for Packed and Polymorphic Malware*, in IEEE transactions on computers, 19 March 2012, DOI: 10.1109/TC.2012.65.

[15] Min Gyung Kang, Pongsin Poosankam, and Heng Yin, *Renovo: A Hidden Code Extractor for Packed Executables*, WORM'07, November 2, 2007, ACM 978-1-59593-886-2/07/0011.

[16] Tom Brosch, Maik, Morgenstern, *Runtime Packer: The Hidden Problem*, Black Hat USA, January 2006.

[17] Symbolic execution, https://en.wikipedia.org/wiki/Symbolic_execution

[18] Satyajit Daulaguphu, Exciting Journey Towards Import Address Table (IAT) of an Executable, https://tech-zealots.com/malware-analysis/journey-towards-import-address-table-of-an-executable-file/

[19] Themida, https://www.oreans.com/Themida.php, Oreans, Software Security and Licensing Solutions.

[20] Arie Olshtein, *Following the Scent of Trickgate: 6-Year-Old Packer Used to Deploy the Most Wanted Malware*, Check Point Research, January 30, 2023.

[21] https://reverseengineering.stackexchange.com/questions/2142/what-is-import-reconstruction-and-why-is-it-necessary, Stack Exchange.

[22] *Packed Malware Basics*, Arridae Infosec Pvt. Ltd., Jan 08, 2020.

[23] Metamorphic Code, https://en.wikipedia.org/wiki/Metamorphic_code

[24] Debugger flow control: Hardware breakpoints vs software breakpoints http://www.nynaeve.net/?p=80

[25] Import table vs Import Address Table https://reverseengineering.stackexchange.com/questions/16870/import-table-vs-import-address-table

[26] Riccardo Meggiato, *Themida, il tool che protegge gli eseguibili e piace ai malware*, cybersecurity360, 25 Gen 2022.

[27] Commando VM, https://github.com/mandiant/commando-vm

[28] Symbolic Execution, https://en.wikipedia.org/wiki/Symbolic_execution, Wikipedia.

[29] Kurt Baker, Malware Analysis, crowdstrike, April 17, 2023.

[30] PE Format, https://learn.microsoft.com/en-us/windows/win32/debug/pe-format, Microsoft, 03-24-2023.

[31] Saurabh Dept., *Advance Malware Analysis Using Static and Dynamic Methodology*, IEEE, 19 December 2019, DOI: 10.1109/ICACAT.2018.8933769.

[32] ASPack, http://www.aspack.com/, StarForce Technologies Inc.

[33] Yuxin Gao, Zexin Lu, Yuqing Luo, *Survey on malware anti-analysis Yuxin Gao*, IEEE, 15 January 2015, DOI: 10.1109/ICICIP.2014.7010353.

[34] Forrest Orr, *Phantom DLL Hollowing*, Forrest-orr.net, Oct 19, 2020.

[35] Binlin Cheng, Jiang Ming, Jianmin Fu, Guojun Peng, Ting Chen, Xiaosong Zhang, Jean-Yves Marion, *Towards Paving the Way for Large-Scale Windows Malware Analysis: Generic Binary Unpacking with Orders-of-Magnitude Performance Boost*, 2018 ACM SIGSAC Conference, 15 October 2018, https://doi.org/10.1145/3243734.3243771.

[36] Min-Jae Kim, Jin-Young Lee, Hye-Young Chang, SeongJe Cho, Yongsu Park, Minkyu Park, Philip A. Wilsey, *Design and Performance Evaluation of Binary Code*

Packing for Protecting Embedded Software against Reverse Engineering, IEEE, 07 June 2010, DOI: 10.1109/ISORC.2010.23.

[37] Alessandro Mantovani, Simone Aonzo, Xabier Ugarte-Pedrero, Alessio Merlo, Davide Balzarotti, *Prevalence and Impact of Low-Entropy Packing Schemes in the Malware Ecosystem*, Network and Distributed System Security (NDSS) Symposium, 23-26 February 2020, https://doi.org/10.14722/ndss.2020.24297.

[38] Wei Yan, Zheng Zhang, Nirwan Ansari, *Revealing Packed Malware*, IEEE, 07 October 2008, DOI: 10.1109/MSP.2008.126.

[39] Yuxin Gao, Zexin Lu, and Yuqing Luo, *Survey on malware anti-analysis*, Fifth International Conference on Intelligence Control and Information Processing, IEEE, 20 Aug 2014, DOI: 10.1109/ICICIP.2014.7010353.

[40] Donghwi Shin, Chaetae Im, Hyuncheol Jeong, Seungjoo Kim, Dongho Won, *The New Signature Generation Method Based on an Unpacking Algorithm and Procedure for a Packer Detection*, International Journal of Advanced Science and Technology, 7 February 2011.

[41] Robert Lyda, James Hamrock, *Using Entropy Analysis to Find Encrypted and Packed Malware*, in IEEE Security & Privacy, DOI: 10.1109/MSP.2007.48.

[42] Trivikram Muralidharan, Aviad Cohen, Noa Gerson, Nir Nissim, *File Packing from the Malware Perspective: Techniques, Analysis Approaches, and Directions for Enhancements*, ACM Computing, 03 December 2022, https://doi.org/10.1145/3530810.

[43] Li Sun, Steven Versteeg, Serdar Boztaş, Trevor Yann, *Pattern Recognition Techniques for the Classification of Malware Packers*, Springer, Berlin, Heidelberg, Australasian Conference on Information Security and Privacy (ACISP) 2010, https://doi.org/10.1007/978-3-642-14081-5_23.

[44] Xingwei Li, Zheng Shan, Fudong Liu, Yihang Chen, Yifan Hou, A Consistently-Executing Graph-Based Approach for Malware Packer Identification, IEEE, 23 April 2019, DOI: 10.1109/ACCESS.2019.2910268.

[45] Anitta Patience Namanya, Andrea Cullen, Irfan U. Awan, Jules Pagna Disso, *The World of Malware: An Overview*, in 2018 IEEE 6th International Conference on Future Internet of Things and Cloud (FiCloud), 11 September 2018, DOI: 10.1109/FiCloud.2018.00067.

[46] Philip O'Kane; Sakir Sezer, Kieran McLaughlin, *Obfuscation: The Hidden Malware*, in IEEE Security & Privacy, 04 August 2011, DOI: 10.1109/MSP.2011.98.

[47] Grégoire Jacob, Paolo Milani Comparetti, Matthias Neugschwandtner, Christopher Kruegel & Giovanni Vigna, *A Static, Packer-Agnostic Filter to Detect Similar Malware Samples*, International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA) 2012, https://doi.org/10.1007/978-3-642-37300-8_6.

[48] Kesav Kancherla, Srinivas Mukkamala, *Image visualization based malware detection*, in 2013 IEEE Symposium on Computational Intelligence in Cyber Security (CICS), 16 September 2013, DOI: 10.1109/CICYBS.2013.6597204.

[49] Veeramani R, Nitin Rai, *Windows API based Malware Detection and Framework Analysis*, ACSAC '20: Annual Computer Security Applications Conference, 08 December 2020, https://doi.org/10.1145/3427228.3427242.

[50] Jana Šťastná, Martin Tomášek, *Exploring Malware Behaviour for Improvement of Malware Signatures*, in 2015 IEEE 13th International Scientific Conference on Informatics, 11 January 2016, DOI: 10.1109/Informatics.2015.7377846.

[51] Fanglu Guo, Peter Ferrie, Tzi-cker Chiueh, *A Study of the Packer Problem and Its Solutions*, International Workshop on Recent Advances in Intrusion Detection (RAID) 2018, Springer, Berlin, Heidelberg, https://doi.org/10.1007/978-3-540-87403-4_6.

[52] Binlin Cheng ,Jiang Ming, Jianmin Fu, Guojun Peng, Ting Chen, Xiaosong Zhang, Jean-Yves Marion, *Towards Paving the Way for Large-Scale Windows Malware Analysis: Generic Binary Unpacking with Orders-of-Magnitude Performance Boost*, CCS '18: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, 15 October 2018, https://doi.org/10.1145/3243734.3243771

[53] Kevin A. Roundy, Barton P. Miller, *Binary-code obfuscations in prevalent packer tools*, ACM Computing Surveys, 11 July 2013, https://doi.org/10.1145/2522968.2522972.

[54] Joan Calvet, Fanny Lalonde Levesque, Erwann Traourouder, François Menet, José M. Fernandez, Jean-Yves Marion, *WaveAtlas: Surfing Through the Landscape of Current Malware Packers*, Virus Bulletin Conference, 30 Sept 2015.

[55] Dhruwajita Devi, Sukumar Nandi, *Detection of packed malware*, SecurIT '12: Proceedings of the First International Conference on Security of Internet of Things, August 2012, https://doi.org/10.1145/2490428.2490431.

[56] Min-Jae Kim, Jin-Young Lee, Hye-Young Chang, SeongJe Cho, Yongsu Park, Minkyu Park, Philip A. Wilsey, *Design and Performance Evaluation of Binary Code Packing for Protecting Embedded Software against Reverse Engineering*, IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, 07 June 2010, DOI: 10.1109/ISORC.2010.23.

[57] IoT Embedded System, https://emteria.com/blog/iot-embedded-system#:~:text=The%20difference%20between%20an%20embedded,of%20devices%20and%20use%20cases, Emetria, 5 Jul 2022.