POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

A Study of Evasive Behaviors in Commercial Packers

Tesi di Laurea Magistrale in
Computer science Engineering - Ingegneria Informatica

Author: **Giorgio Coccia**

Student ID: 912960
Advisor: Prof. Mario Polino
Co-advisors: Michele Carminati, Stefano Zanero
Academic Year: 2021-2022

Abstract

Sandboxes, Virtual machines, Threat intelligence , there are plenty of tools and techniques that could be used to identify and analyze malware. Malicious programs need a way of hiding the code and slowing down forensic analysis. For this purpose, Packers are software used for obfuscating and compressing the code and they are widely used as a countermeasure to forensics tools. In this study, we are not gonna focus on the obfuscation added by packers, but on the anti debugging techniques that may be added to the packed file. The final goal is to demonstrate that the intent of the packers is not only to hide the content of a file but also to prevent the analysis of the program. We first take a pool of 20 packers and we use them to pack a small and basic program. Then, we run the packed files into a framework that is able to identify the anti debugging techniques involved. The results will give an opportunity to prove that packers use multiple anti debugging tools, to identify the most common techniques used and the most uncommon and advanced ones. The output will be used to generate statistics, identify frequent packer behavior, and provide the basis for developing new unpacking and anti-analysis tools based on the findings.

Keywords: Packer , Debugger , Anti Debugging ,Anti VM

Abstract in lingua italiana

Sanbox, Macchine virtuali, Threat intelligence , ci sono una moltitudine di strumenti e tecniche che potrebbero essere utilizzati per identificare e analizzare malware. Per questo motivo, i virus hanno bisogno di un modo per nascondere il codice e rallentare l'analisi forense. Una delle soluzione adottate per risolvere questo problema sono i Packers, software utilizzati per offuscare e comprimere il codice . In questo studio, non ci concentreremo sull'offuscamento del codice operato dai packer, ma sulle tecniche anti debug che potrebbero implementare nel file compresso. L'obiettivo finale è dimostrare che l'intento dei packers non è solo nascondere il contenuto di un file, ma anche impedire l'analisi del programma. Prenderemo un pool di 20 packers e li useremo per comprimere un programma di origine da usare come test. Quindi, eseguiremo i file compressi in un framework in grado di identificare le tecniche di anti debug coinvolte. I risultati daranno l'opportunità di dimostrare che i packers utilizzano molteplici strumenti di anti debug, per identificare le tecniche più comuni utilizzate e quelle più ricercate ed avanzate. L'output sarà utilizzato per generare statistiche, identificare il comportamento dei packers e fornire la base per lo sviluppo di nuovi strumenti di unpacking ed anti-analisi basati sui risultati.

Parole chiave: Packer , Debugger , Anti Debugging ,Anti VM

Contents

1 | An introduction to the writing of scientific texts

Science, my boy, is made up of mistakes, but they are mistakes which it is useful to make, because they lead little by little to the truth.

—

1.1. Introduction

The Cisco AIR (formerly Cisco's Visual Networking Index or VNI) [4] stated that the number of overall connected devices in 2023 will be close to 29.3 billion, an increase of almost 40% compared to 18.4 billion in 2018.
This pattern demonstrates how the population is becoming increasingly computerized. We live in a world where employees work smarter, where people spend more than 4 hours a day on social media, and where laptops are the main tool for learning and working [10]. With such an increase in the number of network possible entries, it increases also the risks for private and sensible data of users: phishing attacks on personal emails, Trojans, Ransomware that could cost billions for companies.
Over the last decade, there has been an 87% [19] increase in malware infections and we predict to spend in Cybersecurity research and technologies a sum of 1.75 trillion dollars for the five-year period from 2021 to 2025 [17].
How is it possible that after all of our efforts and investments, we are still unable to detect and eliminate viruses?
The majority of dangerous programs are either zero-day malware that exploits newly found vulnerabilities or old malware that has been in use for years and has been updated or camouflaged using modern technologies and tactics. One of the most powerful overused weapon for escaping antivirus and proxy defenses are the **packers** [2] [23].
Packers were born as software for compressing and hiding the code but they are becoming a must for spreading malware. Latest studies demonstrate that the usage of packers is becoming a standard for viruses: 80% of malware uses packing software and 50% of the

malicious files are the same malware obfuscated with different packers [2]. Compiling small and portable programs is efficient, but the main reason behind this trend is that packing is an efficient way to avoid the most common and effective anti-malware tools: Threat intelligence, Static Analysis, and Machine learning.

Threat Intelligence Because we can simply access different versions of the same trojan using the polymorphism characteristic of packers, the Threat Intelligence analytic has become less effective.

Static analysis Because packers use numerous levels of encryption and the original code is decrypted at run time, static analysis is time-consuming and ineffective.

Machine learning It was proved in [8] that a machine learning system tends to connect a packed file with malware rather than distinguishing between benign and harmful packed files. Studies are still in process, and while they may be used to examine data more quickly, it is not yet a comprehensive and definite answer.

The remaining and reasonable solution is Dynamic analysis: it can be used to find the original code when decrypted in memory, analyze registries and memory dumps, or process and network Information.
If the packers also provide a defense against dynamic analysis, that means they present a barrier for every sort of well-known and frequent analysis. Whether the barrier is strong or small, it will certainly slow down and complicate the analysis.
To get this answer, we'll look into whether packers employ Anti-Debugging and Anti-VM technologies, as well as which techniques are the most popular. As a result, the study will give evidence for the application of anti-debugging while simultaneously improving the reverse of packed data.
The study's unique aspect is that it employs instrumentation to detect anti-debugging rather than more traditional methods. Instrumentation [20] is the best option for this since it can disguise the analyzing process and identify anti-analysis ways without being detected, while also implementing countermeasures to ensure safe program completion.
The following are the questions that this thesis attempts to answer:

RQ1. *Are anti-debugging tactics being used by packers?*

RQ2. *Is there a correlation between a set of anti debugging methods and packers?*

RQ3. *In our dataset of packers, what techniques are exploited?*

2 | State of the art

2.1. Related works

Anti debugging techniques and packers are widely studied in the context of Cybersecurity. The quantity and complexity of anti-debugging techniques are growing, and there are multiple well-documented manuals [9][6][14] offering strategies to fool the debugger. As can be seen in [9][6][14], new approaches have been added to the initial set of well-known techniques over a short period of time, demonstrating how rich, varied, and full of possibilities this topic is.
However, debugging the code has been for years one of the most important resources for the analysis and the unpacking of packers [3]–[9].

The majority of these methods rely on different heuristics to recognize the completion of the unpacking procedure, and therefore the correct moment to dump the contents of the process memory, and are based on dynamic execution of the sample (e.g., by an emulator or a debugger). Being able to debug a packed file has been the primary weapon for unpackers and analyzers, giving the opportunity to study the original code and publish unpacking routines that are still in use by Antivirus [18]. Finding a correlation between packers and anti-debugging would open an important question on the effectiveness of Dynamic routines used by Antivirus, Unpackers, and Debuggers, for analyzing and unpacking code.

Relation between Anti debugging and Packers

We have multiple examples of papers demonstrating an initial correlation between packers and anti-debugging, starting from [20] which is a study of anti-debugging techniques used by a large data set of malware. The authors discovered a link between the samples and the methodologies used throughout their investigation.
Because malware is frequently packed, they attempted to build a machine learning sys-

tem trained through anti dbitechniques and predict the appropriate packer family. In the results, only PEtite demonstrated a clear correspondence, but it is a good result thinking that the data set was not studied for that purpose as well as the classifier used. We employed the same instrumentation equipment and approach as in this paper [20] , but changed the scope of the study and added new features that targeted packers more specifically.

The second main source of this paper is the article written by Peter Ferrie [21] for Microsoft company. The paper's content is a study of anti-analysis strategies used by a group of packers, which were identified through debugging operations. The methodology (Instrumentation vs. Debugging), the pool of packers, and the varied techniques are the differences in our study. On the other hand, he provided the opportunity to compare the results in various ways, check previously established techniques, and add new ones. One of the most significant contributions was the theoretical explanation of new techniques, the foundation of new functionalities, and countermeasures added to our Instrumentation instrument.

The article written by Young Bi Lee et Al.[25] is the final significant source, which examines the connection between the two components. This is the piece that most closely resembles our methodology in this study; it is an analysis of commercial packer and protector anti-debugging strategies, as well as solutions for circumventing them. The tool used, according to our study, is Intel Pin tool. It was a useful resource for confirming our findings and discovering new sites for feature study. The differences are the data set of packers, limited to a few protectors in their case; the final scope, which was limited to targeted anti vm and anti debug functions, while our research is a complete overview on all the packers world; the instrumentation was used not to discover which anti analysis function is used, but to defeat already known techniques.

Theoretical manuals on Anti debugging techniques

During the experiment, a great source were three types of guide [14][9][6]. The first is a

web page that contains the most well-known strategies, which are described using c++ examples. They are brief but intuitive explanations, and they have been one of the thesis's key resources.
The second one is [6] written by Peter Ferrie, and it is a more comprehensive book that examines all possible anti-debugging behavior. This study not only describes new and advanced strategies but also provides alternatives for applying and overcoming such techniques.

Summary

As can be seen, there is a lot of information out there about the link between packers and debugging, but this is the first comprehensive analysis of a pool of 45+ techniques that have been tested using Instrumenting packed files. The method for checking the approaches, which is done via Instrumentation, as well as the vast pool of anti-debugging techniques employed, which is not restricted to simple and common techniques, but also advanced and complex ones, is the interesting component of this research. Another item to consider is that the data set used is based on 20 packers that contain the programs used in the prior studies, allowing for a comprehensive overview of the packer families and the ability to compare, confirm, and extend the findings of past studies.

2.2. Different types of analysis

2.2.1. Static analysis

As stated in the introduction, various forms of analysis may be used to examine a packed software. **Static Analysis** is the first option. Several ways for using static analysis to detect and unpack programs have already been proposed [5], but as soon as they were deployed and established a standard for every antivirus, new techniques and technologies emerged that weakened this solution (Example Packers).
The main difference between this type of analysis and the others is when it is performed. In particular, for applying a static analysis there is no need to run the program and it is applied before the execution, because of this it is called static .

The first step in using static analysis is to examine the code. This can refer to a variety of things, including plain text code (Java, Cpp,.Net, etc.), the assembly of a compiled program, or directly byte code. This approach can be done manually by skilled reverse engineers, but the most common solution is tools and automation. They are used to study the flow of a program and comprehend its purpose, allowing malicious code to be extracted and possible malware to be detected.
Examples of such solutions are ObjDump [13], a software that can deconstruct the code into machine language, DnsSpy [12], which can revert a.Net file to its original CPP code, or tools like IDA [7] or Redare, which can generate a graph that simulates the execution of the code without running it.

The second strategy is to gather as much information as possible on the entity being investigated: hashes, names, magic numbers, file types, user reports on the internet, and so on. Some of them, such as hashes or byte signatures that characterize a virus family, is employed by antivirus and firewalls as a check for the maliciousness of the code.
The problem is that obfuscation and polymorphic code can completely avoid this technique, for example, Christodorescu et al. observed in an experiment that “three commercial virus scanners could be subverted by very simple obfuscation transformations”[3] .

2.2.2. Dynamic Analysis

The second solution is Dynamic analysis, which is one of the topics of this research. Dynamic analysis is more effective than static analysis since it does not require disassembling the infected file to evaluate it. Furthermore, dynamic analysis can detect both known and undiscovered malware, and malware that is obfuscated or polymorphic cannot elude dynamic detection. Dynamic analysis, on the other hand, is time and resource expensive.

Function call monitoring, function parameter analysis, instruction tracing, and information flow tracking are some of the techniques that may be employed with dynamic analysis. All of these solutions are based on traces left during the execution that may regard functions, data stored in memory, processes, and others. The analysis can also regard the traffic in input and output while the program is executed, useful to find a possible connection to malicious domains. The most famous programs used for such information in reversing are Procmonitor, Procexp , Regshot, all of them part of a set of dynamic analysis tools provided by Sysinternals, a windows utility set of 70 programs [16].

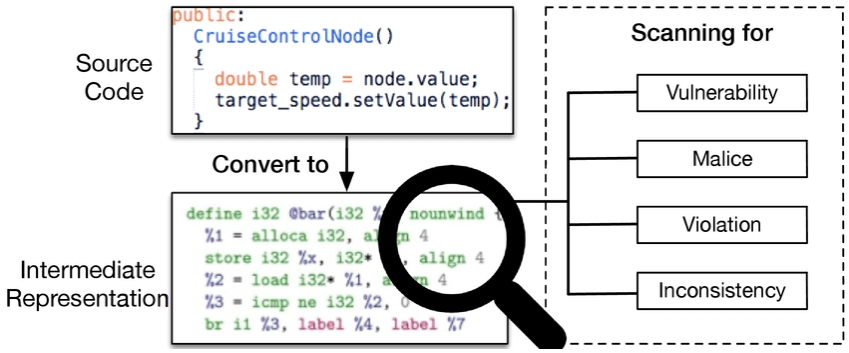

Figure 2.1: Example of static analysis performed starting from the source code. .

2.3. Process Environment Block

The PEB is a section of memory where are stored useful information and flag are used to discover the presence of a debugger. It is a Windows NT structure and it saves data that is necessary for the operation of the system and the proper flow of the process. The PEB stores Global context, starting parameters, data structures for the program image loader, the program image base address, and synchronization objects needed to enable mutual exclusion for process-wide data structures. Above all the stored data, we are interested in part of them useful for the research:

- IsDebuggedFlag

It can be found in the location fs[:30] at offset 0x2 and is set to 1 when the program is debugged.

- NTGlobalFlag

The information that the system utilizes to determine how to generate heap structures is stored in the PEB at offset 0x68 in an undocumented location. When the value at this position is 0x70, we know we're in debugger mode.

- HeapFlag

In the PEB structure, ProcessHeap is at 0x18. This heap has a header with fields that indicate the kernel whether it was created in a debugger or not. The ForceFlags and Flags fields are how they're called, and to identify a debugger the ForceFlags must be different from zero while in the Flags field we must not have the HEAP_GROWABLE (0x00000002) flag set [1].

2.4. What is a Packer

The concept of packer is the main topic in this research. A packer is software that provides obfuscation and compression to a file for privacy reasons but mostly for bypassing security analysis. A packer takes the original code and hides it by adding encryption, polymorphism, and routines that must be completed in order to reach the beginning program. As can be seen in the example (Fig. 2.1), Packers inserts superfluous variables, recreates the test using string sums, for cycles, and random variables, and makes the original code

hard to read and comprehend.
The example taken is the multi-staged virus Emotet coded in power shell script, it has been taken as example because it is in clear text and it can be directly seen how confused and meaningless is the code provided.

```
$hh='hi'+'dd'+'en';
$gIlZxbyvdhbBM=@(1..16);
$xztKds=-join ((65..90) + (97..122) | Get-Random -Count 9 | % {[char]$_
    });
$myFLENLDEQ = $ bHBcnobnyRxRNzqezNXi+"\"+$xztKds;
$uHmMCma="";
for ($i=0; $i -le $myFLENLDEQ.length-1; $i++){;
$uHmMCma+=$myFLENLDEQ[$i]+'EF';
```

Listing 2.1: Example of packed malware, written in powershell script.

Packers can create different layers of unpacking, the first layer unpacks the second layer, the second unpacks the third, and so on until we reach our code. In the article "When malware is packin' heat" [8] , packers are divided into 5 categories and we are gonna take the same categorization for better understanding:

Type I

This is the example case, where an unpacking routine is performed linearly and at the end of the program the code is in plain memory and is executed. Examples of this type are packers lightweight like Mpress and Upx.

Type II

In this second level there is not a single packing routing but multiple ones. The first layer unpacks the second one and the second layer the following one. At the end of the transactions, like in Type I, the code will be fully reconstructed and it will be executed.

Type III

This type is a development of type II; it performs sequential packing procedures but may mix patterns and go ahead and backward between them. It commonly saves the last layer for checking if the original code is unpacked, anti-debugging routines, or the packer's obfuscated code. The crucial issue is that the runtime procedure and the execution at the conclusion of the real program's process are still distinguishable and divisible.

Type IV
This kind is similar to the preceding ones, except that the packer's execution isn't confined to the unpacking procedure; it may also inject its own lines of code into the original program. The goal is to provide an extra layer of security that includes anti-debugging methods, obfuscation, and encryption.

Type V
The unpacking procedures and the execution of the original code are interleaved, which distinguishes this Type from Type V. This means that while some lines of the original code are run during unpacking, it may still be fully recovered using a complete memory dump.

Type VI
This is the type of packer that has the lower granularity level. It means that while doing the unpacking routine the packer execution can decide to unpack a single instruction of the original code and execute it.

2.5. Unpacking

As files can be packed, they can be also unpacked and reverted to their original state. This is usually possible through a recovered file generated by the packer itself, but if the file is missing or during the reversing of an unknown file, the possible solution could be using Unpackers [15]. These are software created to revert packed file, starting from the packing routines of well-known packers. Taking into mind the constraints that they may offer, they can be a viable alternative to dynamic analysis.

They frequently assume that **(I)** the entire original code is unpacked in memory at a specific point in time, **(II)** if a sample contains multiple layers of packing, these are unpacked in order and the original application code is decoded in the last layer and **(III)** the packer's execution and the original application's execution are not mangled together (i.e., the packer transfers control to the original application at a specific point in time).
As it can be noticed, these assumptions are obsolete and can be avoided by changing the packing routine or implementing more innovative solutions.

There are multiple methods they use for unpacking, the most used is an approach based on the signature. It works by comparing the code and trying to find behaviors or routines that are specific to a packer. Of course, this method is limited by the fact that the software must know the routine of the packer that obfuscated the file, or the fact that packers can change routine or get updates .

Finally, this solution cannot be relied upon because it is only necessary to modify a portion of the code or construct a private packer to overcome it.On the other hand, it is a simple, intuitive method that identifies the majority of the most common and well-known packers. The alternative is using a dynamic packer, which doesn't compare signature but is able to discover when a program is performing the packing routine and when is starting the original code once unpacked.This is possible because it recognizes when all of the original code has been unpacked and the program calls a memory location that previously had no executable code. But this is possible only for linear packers, in other most sophisticated routines there is the need of stronger algorithm and expensive in matter of time and cpu.

PEiD[24] is the most well-known and widely used; it is unpublished software that can be found in many repositories and is still widely utilized. Static analysis, dynamic analysis, and entropy for determining if a file is packed and where it is unpacking the code are all features of this application.

2.6. Debuggers

A debugger is a tool used to trace the execution of the program. It can be used to find mistakes in your own code, reverse software and have a better understanding of what the code is doing in every moment of its execution. It works by attaching himself to an existing process or directly spawning the process to analyze.

Once the process is spawned and after becoming the father of the process, the debugger waits for signals, parse them, and executes the linked routine. Operating systems provide specific libraries for dealing with this feature and provide the debugger with the necessary tools to intercept signals.
The debugger intercepts all the exceptions and traps of the debuggee program, but the most used and important is INT 3, a 1 byte instruction that can be added at the beginning of every instruction, causing a break-point exception and calling the debugging

exception handler. It is an interception that stops the analyzed program and returns the execution to the debugger. Here the debugger can obtain the state of the process' memory, its registers, and every information stored before the interruption.

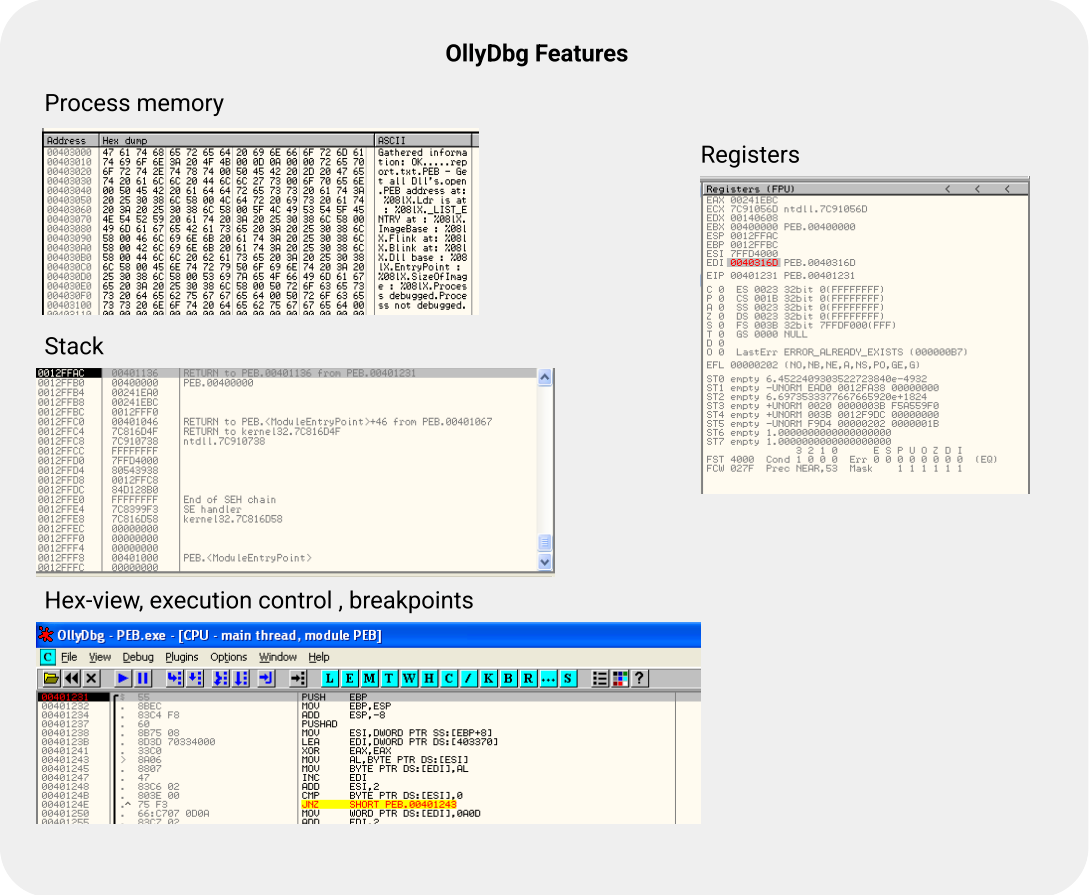

Figure 2.2: This figure represents a common graphical interface for debuggers, evidencing the main views that can be used for analyzing the execution of a program. .

The UI and key features of OllyDbg, one of the most popular debuggers with a Graphical Interface, are shown in Fig 2.2. We can see information such as the stack, register data, and the instruction where the program is stopped in the image. X64 and IDA pro were used in the studies to double-check procedures and detect problems in the code. Because there are strategies that are meant to avoid certain debuggers and because they present distinct features, the need to utilize different debuggers arose.

2.7. Instrumentation

The use of Instrumentation rather than a traditional Debugger is one of the most innovative aspects of this study. Instrumenting software means adding code to the

original executable at run time. Such a definition can be summarized as JIT compiler, which means Just In Time Compiler. The instrumentation tool intercepts the execution of the first instruction of the executable and generates ("compiles") new code for the straight line code sequence starting at this instruction. It then transfers control to the generated sequence (Fig. 2.3).

It can be decided whether to execute the original code or only use it as a reference and execute only the extra code during instrumentation. Because of this, it became a powerful tool against anti debugging technique because it is possible to specify routines for functions that are used during anti debugging and avoid or print them into the logs. Furthermore, anti-debugging techniques hunt for debugger traces, but with instrumentation, some of the debugger residues are partially gone.

In the paper "Measuring and Defeating Anti-Instrumentation-Equipped Malware" [22], using a set of procedures that hides the presence of a debugger while executing, instrumentation has been used to bypass anti-dbi techniques, demonstrating how beneficial it may be against analysis protection, and the same approach can be used for anti debugging .

The main advantages of instrumentation are that it can discover code at runtime, allowing it to discover and instrument it as the code is unpacked; it can be attached to a running process, allowing it to hide; and it can intercept exceptions, functions, and system calls, providing an infinite number of possibilities for code analysis.

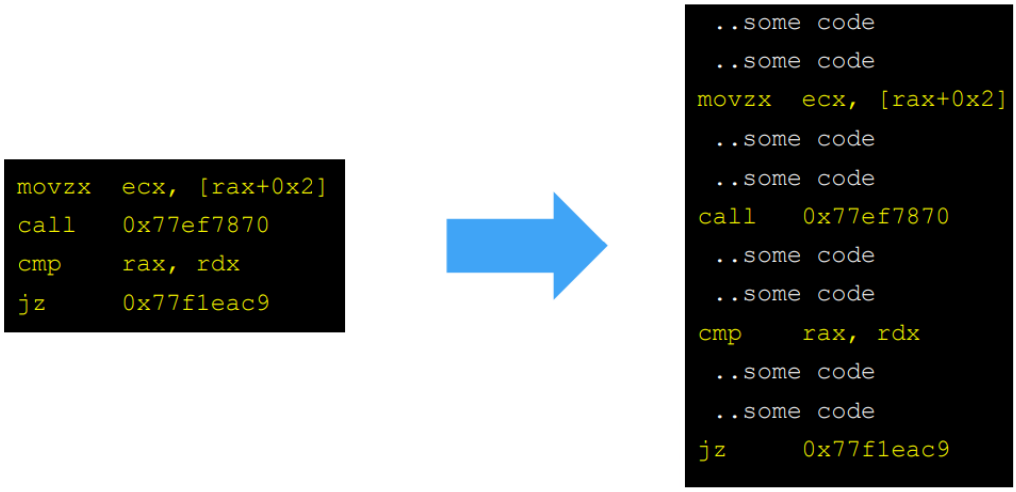

Figure 2.3: This figure represent the process of adding new lines of code to the instrumented program. .

3 | Anti Dynamic analysis techniques

3.1. Anti debugging techniques

All of the anti-debugging techniques involved and studied are based on fingerprinting: they try to find common traces during the execution of a process and the execution of a process while it is being debugged, and if one of the techniques returns a positive result, the program changes behavior or stops running. This type of approach can compare the timing of execution of a function, compare the expected memory address with the one provided while debugging, system information that involves the presence of a debugger, and also the management of exceptions and traps.
Because of this wide range of different techniques, we divided them into 6 macro categories: **Anti-debugging, Anti-VM, File, Registry, Stalling, Timing.** .

3.2. Anti-debugging

This is the most comprehensive set, containing all of the well-known and widely used functions and approaches for detecting the existence of a debugger. They differ from the other sections in that the method's origin is not in a file, timing, or registry check, but rather in functions and instructions designed specifically for debugging. :

- `IsDebuggerPresent()`
  When a process is being debugged it triggers multiple flags inside the PEB, the one searched by this function is the BeingDebugged flag. This technique works by calling the mentioned function to check if the flag is set to 1, else the process will terminate because it means that there is a debugger. The same is applied for CheckRemoteDebuggerPresent. A possible alternative is using the assembly code for avoiding hooks, it is based on 3 simple lines, the first retrieves the PEB address,

the second one takes the value of NtFlag, and finally, the flag is returned(Fig.3.1.

```
mov %eax,fs:30h;
movzx %eax,[%eax+2];
retn
```

Listing 3.1: Example of code using IsDebuggerPresentTechnique().

- `SetUnhandledExceptionFilter()`
  This is an example of Exception Handling technique. When a process is being debugged traps and exceptions are intercepted by the debugger. If an exception is triggered and there is no handler specified, the UnhandledExceptionFilter() will be called. By using SetUnahandledExceptionFilter you can modify the UnhandledExceptionFilter() with your own code: if it is called by the process it means it is not debugged, else it has been intercepted and handled by the debugger (Fig. 3.2).

```
LONG UnhandledExceptionFilter(PEXCEPTION_POINTERS pExceptionInfo)
{
    PCONTEXT ctx = pExceptionInfo->ContextRecord;
    ctx->Eip += 3; // Skip \xCC\xEB\x??
    return EXCEPTION_CONTINUE_EXECUTION;
}
bool Check()
{
    bool bDebugged = true;
    SetUnhandledExceptionFilter((LPTOP_LEVEL_EXCEPTION_FILTER)
    UnhandledExceptionFilter);
    __asm
    {
        int 3                       // CC
        jmp near being_debugged     // EB ??
    }
    bDebugged = false;
being_debugged:
    return bDebugged;
}
```

Listing 3.2: Example of SetUnhandledExceptionFilter() technique added to the code.

- `Instruction INT 3`
  The INT 3 instruction is a similar mechanism that is included in the Exception Handling group but is less detailed. This exception is intercepted debugger and is utilized for breakpoints, as indicated in paragraph 2.6. This means that if it's called from within the packed program's code and the exception isn't handled by the original process, a

debugger could be looking for breakpoints. The next section of code contains an example of the implementation (Fig. 3.3):

```
    bool IsDebugged()
{
        __try
    {
        __asm int 3;
        return true;
    }
    __except(EXCEPTION_EXECUTE_HANDLER)
    {
        return false;
    }
}
}
```

Listing 3.3: Example of Int 3 technique added to the code.

- `NtQueryInformationProcess()/NtQuerySystemInformation()`
  This function is used to retrieve information about the process: it can accept documented class as an argument and it outputs the requested information. If we choose as an argument the documented class ProcessDebugPort and we obtain as a result of the dwProcessDebugPort(It is an attribute of the class) the value 0xFFFFFFFF, it means the process is being debugged.
  The EPROCESS structure, which contains the NoDebugIneriht flag for detecting the presence of the debugger, may be obtained by specifying the argument 0x1f; the Debug Object handler, which is produced only if the process is being debugged, can be obtained by specifying the argument 0x1e. Surprisingly, all of these variations appear in our results.

- `FindWindow()`
  This function is used to retrieve windows classes, it can be exploited to check if there are debugging classes imported by the process. Examples of these indicators are classes like Regmonclass, Filemonclass, Procmon window class, and many others.

- `NtGetContextThread(CONTEXT_DEBUG_REGISTERS)`
  This function is used to retrieve the thread_context of the specified thread. If a thread is identified by the hThread parameter, it means that is typically being debugged, but the function can also operate when the thread is not being debugged.

- `Interrupt: IceBP 0xf1,Int1`

These functions are used to raise an exception. If the exception is raised and it is not handled by the program itself, it means that there is a debugger intercepting the exceptions, and it won't be handled. They all raise a different exception, but the scope of this technique is the same.

- `Memory-R: PEB->IS_DEBUGGED,Memory-R: PEB->NTGLOBALFLAG,`
  These two flags have been already discussed in this chapter. They are different from isDebuggerPresent and CheckRemoteDebuggerPresent because they can be retrieved directly by taking the PEB structure and checking the interested part of the structure. When this technique is used, it signifies that the data was acquired directly through assembly code rather than through functions.

- `NtClose()`
  An error code is issued if no debugger is present and an invalid handle is passed to the kernel32 CloseHandle() method (or directly to the ntdll NtClose() function). However, if you use a debugger, you'll get an EXCEPTION INVALID HANDLE (0xc0000008) exception. An exception handler can catch this exception, which signals the existence of a debugger.

- `NtSetInformationThread()`
  A thread can be hidden from a debugger using this function. It can be utilized by either an external or internal thread. The debugger will not receive any events when the thread is hidden, allowing it to run anti-debugging tests without being noticed. The process will crash if there are any breakpoints, and the debugger will be stacked.

3.3. Anti-VM

In this set are inserted all the functions that are used to check the presence of a sandbox or a virtual machine. As in the previous section, these techniques are all performed using functions that directly return the information that may be related to a VM or Emulation.

- `GetAdaptersInfo()`
  This function is a network-related technique that retrieves network parameters, including the machine's MAC address. Most virtual machine suppliers hard code their MAC addresses for virtual machines, and you can tell if an application is executing in a virtual environment by looking at the MAC address. This function can also be used to get the adapter name; if it's something like "VMWare" or "VirtualBox," it means the program is running on an emulating system.

- `GetComputerNameW()`
  This function is used to extract the name of the workstation; the technique is very simple and consists of comparing the computer name with well-known virtual machine default names.

- `GetCursorPosition()`
  This is a method that necessitates human engagement. The function return in x,y coordinates, as you might expect, returns the position of the mouse cursor. If the position never changes, it indicates that the program is not being used by a human, and it could be running in a sandbox, through an automatic script, or in a terminal.

- `GlobalMemoryStatusEx()`
  This function returns memory-related information. You may find out the physical and virtual memory dimensions of the system, which can be utilized to see if the physical memory is less than a certain threshold, such as 1 GB. Virtual environments require less memory because they are used for single tasks and share memory with other virtual machines and the system. Nowadays, systems typically have at least 4 GB of memory, whereas virtual environments require less memory because they are used for single tasks and share memory with other virtual machines and the system.

3.4. File

All of the techniques that utilize file streaming are discussed in this section. During the debugging process, a specific file for debugging, drivers, or libraries for encrypting may be opened. The distinction between this section and the preceding sections is that these techniques look into file descriptors directly, rather than using external functions.

- `NtCreateFile()/NtOpenFile()`
  These functions are used to open a file descriptor to a file. When a file is being debugged, it opens the file descriptor for receiving debugging alerts from the running program. If we try to open a file descriptor in exclusive mode, while it is already opened, it will raise an exception and discover the presence of a debugger or another program that is using the file. This method is employed in our results for two different purposes: the first is to open the hello world file directly. It's possible that the debugger is using the file descriptor if you try to open or create it while it's already in use, causing an exception. The second goal is to inspect the libraries and drivers that debuggers utilize. It entails attempting to open files that are only loaded when the debugger is active.

3.4.1. Registry

Some information may be kept in registries as well as files, and this section contains all of the approaches that involve the opening and reading of registries.

- `NtQueryValueKey()`
  This function returns a value entry for a registry key. It can be used to look for traces of a debugger inside the registries.

- `NtOpenyKey()`
  This function opens an existing registry key. It can be used to look for traces of a debugger inside the registries.

- `NtQueryAttributeKey()`
  This function returns an attribute entry for a registry key. It can be used to look

for traces of a debugger inside the registries.

3.5. Stalling

The delaying functions can be used for a variety of things. The first is as a simple and effective anti-sandboxing technique: sandboxes have a finite amount of time. They stop the application from running after a short or long period of time and provide the analysis findings.
The other goal is to slow down the entire debugging process by using the following functions repeatedly. The last option is samples that stall not just by utilizing the OS's sleeping features, but also by doing pointless arithmetic operations, or system calls. Even worse is software that includes time bombs that only activate on specified dates or timestamps.
In this research, we are gonna focus only on Windows functions that make the program lose time.

- `Sleep()/SleepEx()/NtDelayExecution`
  Suspends the current thread's execution until the time-out interval expires.

- `WaitForSingleObjectEx()`
  Waits until the specified object is notified, an I/O completion routine or an asynchronous procedure call (APC) is queued to the thread, or the time-out interval has passed.

3.6. Timing

The last set is composed of all the methods that measure the timing of a function. Debugging a program introduces noticeable delays in the execution of functions; this technique compares the timing of a function executed by the system to the same on top of a debugger, and if the latter is significantly slower, the program is terminated. (Fig. 3.4).

- `GetTickCount()()/GetTickCount64()`
  The amount of milliseconds that have passed since the system was launched is returned. This value is updated by the system clock ticks.

- `GetLocalTime()`
  The following time is used to get the current date and time in Coordinated Universal

Time (UTC) format.

- `RDTSC`
  Generates the rdtsc instruction, which returns the processor time stamp. The processor time stamp records the number of clock cycles since the last reset.

- `QueryPerformanceCounter()`
  Retrieves the current value of the performance counter, which is a timestamp with a high resolution (1us) that can be used to measure time intervals.

```
bool IsDebugged(DWORD64 qwNativeElapsed)
{
    ULARGE_INTEGER Start, End;
    __asm
    {
        int  2ah
        mov  Start.LowPart, eax
        mov  Start.HighPart, edx
    }
    // ... some work
    __asm
    {   int  2ah
        mov  End.LowPart, eax
        mov  End.HighPart, edx
    }
    return (End.QuadPart - Start.QuadPart) > qwNativeElapsed; }
```

Listing 3.4: Piece of code comparing the timing of a set of instructions, and the same measured without a debugger.

4 | Framework and approach of the experiment

The details of the experiment and how we arrived at our conclusions will be explained in the following chapter. The sample used will be shown in the first section, the data set acquired in the second, and the last sections will be used to describe the tool used and the adjustments made to it.

4.1. Sample used for the experiments: Hello world!

The sample used for studying the packed behavior is a hello world file.
The reason for this decision is that there was a need for a lightweight application that could reduce library imports while also evidencing the logs added by the packers. In particular, the experiment has been used 2 different versions, the first one is compiled with the visual studio x32 compiler and written in C++, while the second one has been compiled with mingw x32 and is written in C code . The reason for having two versions is that the first version was not accepted by a small portion of packers because they didn't support visual studio libraries. On the other hand, it was the version that was compatible with most of the packers.

The second version had the problem that mingw added by default TLS callback. Normally, a program begins with the main function, but with TLS callbacks, you can add lines of code to run before the main function is called. Adding this feature, packers detected the program compiled with mingw as already packed or simply didn't support tls callback, failing the packing process.

Another reason is that outputting a simple hello world string could be used as an indicator to verify if the software is not broken, is functioning well, or if it can be debugged and output the hello world string after being packed. (Fig. 4.1.

```
#include <iostream>
int main()
{
  std::cout << "Hello, World!" << std::endl;
  return 0;
}
```

Listing 4.1: Code of the original sample used for experiments.

4.2. Data set of packers

The data set for this study consisted of 20 commercial packers who were chosen based on a number of criteria. The first criterion was that the packer is capable of packing 32-bit executables into Portable Executable files. Because Brioscia can only instrument 32-bit exe files, this is a must.

The availability of the packer is the second criterion, which is more practical. Because most of the packers did not have their own website where they could be obtained, they had to be downloaded from third-party file-sharing services. The motive is that many packers have been there for years and were built in the early years of 2010, but they are based on outdated architectures and have never been updated.
Armadillo, for example, is one of the most interesting and well-documented software among packers who may employ anti-debugging techniques, but it has been completely removed from standard download sources and is no longer available through third-party sources.

Being free is the third guideline. The majority of packers are now given for a cost, and as a result, the research's limitation is that many of the packers utilized are trial or free research, which may change and have fewer possibilities than the original version.

The last requirement is being a packers used in the referred researches [22][20][6]. The motivation is that in this way it is possible to compare results from different sources and confirm or not the results of other articles(Table 4.2).

Packer	Version	Trial
rlc	3.0.3.18	✗
pecompact	3.0.3.18	✗
mpress	2.19	✗
petite	2.4	✗
obsidium	1.7.4	✓
vmprotect	3.5.1	✗
mew11	1.2	✗
aspack	2.43	✓
pelock	2.11	✓
yoda's protector	1.03	✗
enigma vm	9.7	✓
telock	0.98	✗
themida	3.1.1.0	✗
exe32	545	✗
alternate	2.440	✓
asprotect	2018.2	✓
enigma protector	7.0	✓
upx	3.96	✗
PcGuard		✓
kkrunchy	0.23	✗

Listing 4.2: Table of the packers composing the data set. It is reported the version of the sample used and if they were a trial version.

4.3. Description of the framework MBare

4.3.1. Original Code

Mbare is a framework written in python on top of brioscia. It is a framework that has been used in earlier studies to undertake experiments on malware and harmful files [22][20]. The experiments are designed as a client-server architecture, with the client sending samples to the server (which is a local virtual machine) for analysis, and then collecting the findings once they are available. The goal is to replicate a Sanbox where the packed data (possibly malware) may be inspected without damaging the real OS, ensuring isolation and security in a virtualized environment (Fig. 4.1).

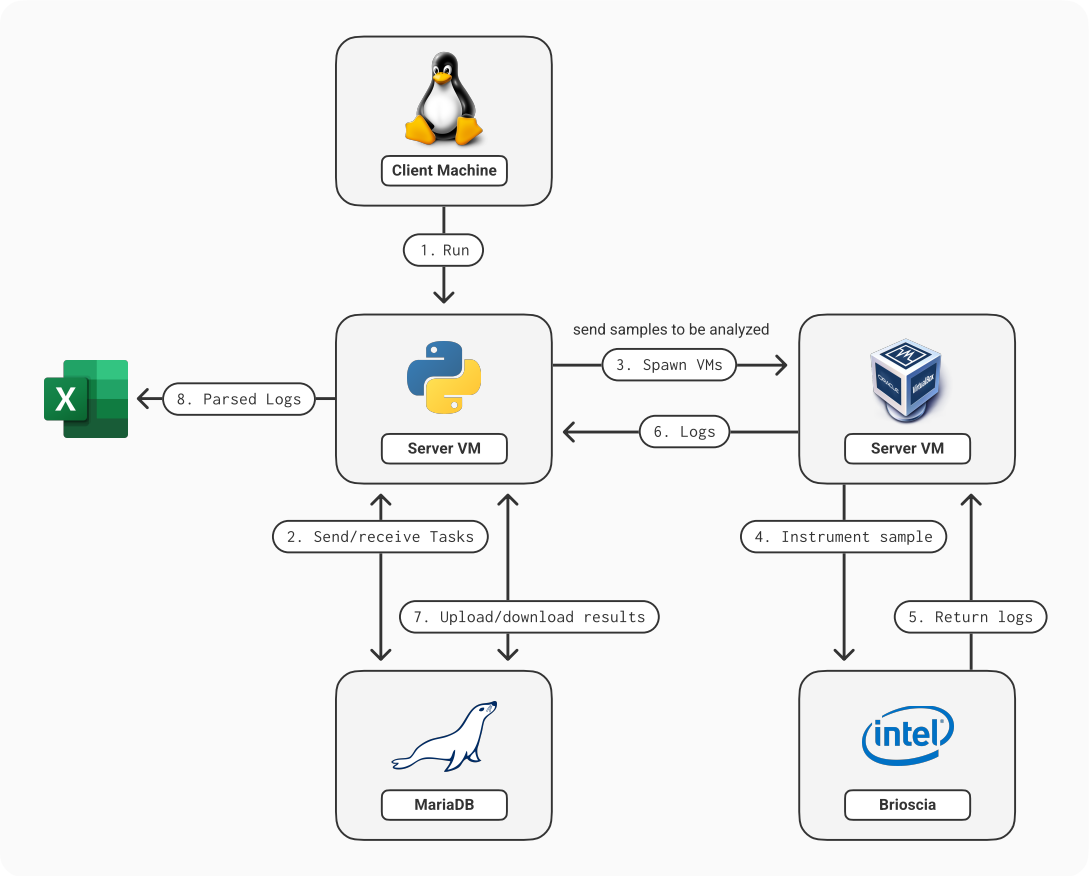

Figure 4.1: Scheme of the entire structure that compose the framework MBare.

Framework Data Flow

The flow done by the MBare framework from the launch of the client application to the retrieval of the parsed logs of the examined sample is described in the following list. The list is arranged in chronological order and describes a typical run for analyzing a packed sample.:

1) The client begins putting tasks in a database (Maria DB) and specifies the packed sample to be evaluated in the new job.

2) When the client is launched, it collects one task at a time from the DB and launches a virtual machine (Vbox) that is configured to listen for and respond to new requests from the client.

3) The client delivers a sample file to the server using an http request.

4)The server consumes the sample into brioscia as soon as it receives the request and waits for the results.
Because the experiments were done on a local system with memory and CPU limits, the framework cannot manage numerous tasks at once. It can work in a multi-threaded mode, but it was not possible to use in such conditions.

5) When the logs are complete, the server delivers them to the client

6) Mbare saves the received logs into the database after collecting the results. The logs connected to anti-debugging machines, which are the output of the instrumentation tool, would be obtained at this point. The issue is that it also contains logs for normal calls and noise generated by the program's execution. As a result, the next step is to process the logs into a parser that selects, counts and generates an excel file including all possible debugging techniques.

7) The client takes a new task and repeats the procedure after resetting the virtual machine's snapshot.

4.3.2. Updates performed to the original code

The support for Virtual Box is the first change made to the original code. Originally, the framework was designed to work with VMWare, but we had to modify a section of code to allow for the spawning and use of VBox Virtual Machines for practical reasons.
The final parser is where the second change is made. The goal of this study is to examine and compile statistics from packer records. To achieve this goal, we created a Python parser that reduced noise from the first logs as well as processed and counted anti-debugging tactics, storing them into an excel file. This parser takes the logs as input and checks them against a list. The list may be seen in fig 4.1, and it contains phrases that are used in registries, files, and path directories, with the goal of reducing false positives. The list has been expanded from the previous version, with the addition of names of debugger drivers that produced different results in the final document, as well as new filenames (Tab. 4.1).

Blacklist
"EXTREM","FILEM","FILEVXG","ICEEXT","NDBGMSG.VXD","NTICE" "REGSYS,"REGVXG","RING0","SICE","SIWVID","TRW" "SPCOMMAND","SYSER","SYSERBOOT" "SYSERDBGMSG","SYSERLANGUAGE", "nkvwovuotd.exe" "cng", "computername", "activecomputername", "Session Manager" "vbox", "virtualbox", "vmware", "vmx", "parallels", "bochs" "wine", "wireshark", "ollydbg", "processhacker", "pin", "tcpview" "autoruns", "filemon", "procmon", "regmon", "procexp", "hookexplorer" "sysinspector", "petools", "dumppcap", "python", "cuckoo", "debugger" "debug", "vmm", "ida", "qemu", "sandbox", "virus", "sample", "malware" "innotek", "dfserv", "vmhgfs", "prl_cc", "prl_tools", "vmsrvc", "vmusrvc" "xen", "softice", "vbox", "virtualbox", "vmware", "vmx", "parallels", "bochs", "wine" "vmm", "qemu", "innotek", "vmhgfs", "prl_cc", "prl_tools", "vmsrvc", "vmusrvc" "xen", "sandbox", "wireshark", "ollydbg", "processhacker", "pin", "tcpview" "autoruns", "filemon", "procmon", "regmon", "procexp", "hookexplorer" "sysinspector", "petools", "dumppcap", "python", "cuckoo", "debugger" "debug", "ida", "virus", "sample", "malware", "dfserv", "softice"

Table 4.1: Table containg the blacklist used during the parsing procedure. In green there are the new added entries.

After parsing the list, the parser compares the logs of the unpacked and packed versions of the program, discarding those that are identical. It is feasible to receive only the new logs added by the packer in this way.
The parser's second job is to generate an output summary that may be used to archive all of the approaches utilized by packers. The result is an excel file that lists all of the techniques found at the row level and the packers that used them at the column level.

Indicators added:

- `NTICE,REGSYS,SYSER..`
  The first part of the table is dedicated to driver names that may be opened by the packed sample for anti-debugging purposes.

- `nkvwovuotd.exe`
  It is the name of the packed executable. The first reason for this is that certain techniques attempt to open the file to see if it has already been opened by another

application that may be examining it, returning an exception. The second reason is because the registry key **registry\machine\software\microsoft\windows nt\\currentversion\\image file execution options\ProgramName.exe** where it may be stored the NtGlobalFlag value (a flag that is set to 1 while debugging a program).

- `NtOpenFile(\\device\\cng)`
  Functions for encrypting strings and code can be found in the cng library. When this file is opened , it may be possible that the code has been hidden using a cryptography algorithm. It is not a foolproof method, but it can be used as a guide.

- `Session Manager`
  The Indicator Session Manager is part of the path **\HKLM\ System\CurrentControlSet\Control\ Session Manager**, which is another way to check the value of the GlobalFlag [21].

- `NtOpenKey/Ex(\\??\control\computername)`
  Inside the registry path **\\registry\machine\system\currentcontrolset\ control\computername** it can be found the name of the computer running the program. This name can be compared against names that are often used in virtual machines and sandboxes, and if there is a match, the program will be terminated.

4.4. Brioscia Intel pin Tool

4.4.1. Original Code

Here is described the instrumentation architecture used and its implementation.
The instrumentation tool used is based on Intel Pin Instrumentation [11], it is a Intel library implemented to give all the necessary tools for doing instrumentation in C++ language. The code has been developed for studying anti debugging techniques and it is used in "Systematical study" [20] and the implementation of Arancino [22], an anti anti dbi tool.
The original Intel Pin version has crucial characteristics that lead to the packer study. The PinDemonium cpp file is the program's starting point; it contains the program's main function and is responsible for the configuration of the running instrumentation.

There are various options available, but the following are the most important:

The exception handler's function is to intercept exceptions, parse them, and schedule a routine to handle them. Because some anti-debugging strategies create interrupts for testing the presence of the debugger, this class is quite significant. It works by hooking exceptions and examining the exception code, printing it in the logs, and starting a function to safeguard the code in the presence of a debugger if the pin shield is configured. The pin shield is another important part of the initial configuration, as mentioned it is used for fixing the environment and avoiding anti debugging technique, a good and easy example is fixing the eip register. A few anti-debugging methods check the eip address to see if the next scheduled instruction is the same as it would be without the debugger. Before entering the exception routine, the shield merely alters the value of the eip address, evading detection of the anti-db technique.

The setting of controlling the son of the starting process is just as critical as this last portion because many packers split the unpacking routine and the code execution. In this approach, the same hooks are grabbed for all of the son processes, increasing the instrumentation tool's coverage power. The instrumentation procedures are the last but most important part of the program. System call and regular hook functions are the two types of hook functions.They required a separate set of functions since the OS handles them differently, but the way they function is the same. If the software detects the name of a registered function, it can start a routine before and after it is executed.
The handlers normally print the technique in the log paper, but they can also give additional information, such as registers,function arguments, and return address, among other things.

4.4.2. Updates performed to the original code

The tool already had all of the necessary information for anti-debugging experiments, but the version was out of date and relied on outdated libraries.
As a consequence, during the first run it caused thousand of errors and couldn't compile at all. It has been necessary about 2 months to totally debug and correct all the errors, reaching a compatible and working version.
The fixing procedure was carried on by identifying the error that had the major number of occurrences, finding a solution, implementing it, and repeating the process. In this way, the number of errors started becoming less and giving the opportunity to patch faster the entire code. In the end, the fix done are lesser fix, but it required deep analysis of the entire code and multiple tries to find out the cause of the problems.

The first version of Brioscia detected different techniques but it was not intended for studying packers. After reading and inserting the papers [14][6], it became more evident that packers can implement their own type of techniques. Starting from papers [21] it was possible to implement new detection functions in Pin that gave multiple hits in packers and demonstrated the usage of anti debugging. This helped also to prove that despite easy techniques and overused, they preferred to detect tricks less famous but effective, which can be detected by skilled reversing engineering skills but would definitely work on less advanced systems.
The list of new hooks detected in brioscia can be seen in Fig. 4.2 and it is explained in the list below.

```
this->functionsMap.insert(std::pair<std::string, int>("Process32First", PROCESS32FIRST_INDEX));
this->functionsMap.insert(std::pair<std::string, int>("Process32Next", PROCESS32NEXT_INDEX));
this->functionsMap.insert(std::pair<std::string, int>("DebugActiveProcessStop", DEBUGACTIVEPROCESSSTOP_INDEX));
this->functionsMap.insert(std::pair<std::string, int>("GetVersion", GETVERSION_INDEX));
this->functionsMap.insert(std::pair<std::string, int>("OpenProcess", OPENPROCESS_INDEX));
this->functionsMap.insert(std::pair<std::string, int>("GetProcessHeap", GETPROCESSHEAP_INDEX));
this->functionsMap.insert(std::pair<std::string, int>("GetWindowThreadProcessId", GETWINDOWTHREADPROCESSID_INDEX));
this->functionsMap.insert(std::pair<std::string, int>("GetShellWindow",GETSHELLWINDOW_INDEX));
this->functionsMap.insert(std::pair<std::string, int>("CreateToolhelp32Snapshot", CREATETOOLHELP32SNAPSHOT_INDEX));
this->functionsMap.insert(std::pair<std::string, int>("SuspendThread", SUSPENDTHREAD_INDEX));
this->functionsMap.insert(std::pair<std::string, int>("BlockInput", BLOCKINPUT_INDEX));
```

Figure 4.2: Piece of code where it can be seen the hooks configured for the new functions.

DebugActiveProcessStop()

This method is an alternative to DebugActiveProcess in that it can start a frash clone of a debugged process. The issue is that a process can only be debugged by one debugger at a time; if it has already been debugged, an error will occur. The function DebugActive-ProcessStop() uses the identical procedure, but instead of launching a process, it tries to stop one that has already been debugged.

GetVersion()

This function is used to ensure that the descriptor table layout matches the operating system platform, discovering if a system is being emulated.

OpenProcess()

When a process is being debugged it acquires full control of the process CSRSS.EXE, which is a system process. If a different program tries to open the same process it will cause a program, and demonstrate that the file is probably being debugged. This technique was studied by Piotr Bania in the year 2005. He released a document demonstrating how this technique could affect and detect OllyDbg and WinDbg, both taking the full privileges

on csrss.exe at their start.

GetProcessHeap()

This function can retrieve 2 important flags, the NTFlags which is set to 2 by default, and the ForceFlag, which is set to 0 by default. These values can be compared to the usual combination used while the process is being debugged.

BlockInput()

This function is used to block keyboard and mouse events. The purpose of using this call for anti debugging is to prevent the debugger from inserting new commands from the command line or totally blocking debuggers that have a GUI.

CreateToolHelp32Snapshot/ Process32Next/Process32First()

The process ID of explorer.exe may be obtained using a coombination of the following functions: CreateToolHelp32Snapshot/ Process32Next/Process32First() (but also OpenProcess()). This id may be compared to the running process's parent to see if it was started by explorer.exe or by another program, like a debugger or Instrumenting program.

CreateToolHelp32Snapshot accepts a flag and a pid as input; if the pid is 0 or null, it produces a snapshot of all processes. The snapshot contains all of the process handlers, and it is an iterable structure. It's enough to read through the entire list until you find the process you're looking for.

This feature allowed the application to determine if the packed program was looking for the parent process, pin.exe, or environment processes such as virtual box or Procmon. It has also been added a feature for avoiding iteration and stopping the software in the enumeration activity: if the list of processes in the snapshot comes to an end, the iterable class Process32Next returns false. If we return false at the first call of Process32Next(), the anti debugging check will halt at the beginning of the list and it will be unable to acquire the list of processes Fig. 4.3.

```
void AntiDebuggingFunctionHooks::CreatToolHelp32SnapshotHook(W::DWORD input1,W::DWORD input2) {
    if (input2 == '0' || input2==NULL) {
        MYTEST("[ANTI-DEBUGGING] CreateToolHelp32Snapshot ( pid= 0)");
    }
    else {
        MYTEST("[ANTI-DEBUGGING] CreateToolHelp32Snapshot ( pid > 0)");
    }
    MYINFO("[ANTI-DEBUGGING] CreateToolHelp32Snapshot(%d, %d)", input1, input2);
}

void AntiDebuggingFunctionHooks::Process32NextHook(W::LPPROCESSENTRY32 input, ADDRINT* input2) {
    MYINFO("[ANTI-DEBUGGING] Process32Next(%s,%d)",input->szExeFile,input->th32ProcessID);
    MYTEST("[ANTI-DEBUGGING] Process32Next");
     *((BOOL*)input2)=FALSE;

}
```

Figure 4.3: Hooks for CreateToolHelp32Snapshot() and Process32Next().

SuspendThread()

This function is usually connected to the technique analyzed before (Section 4.4.2). If the pid is not the same as explorer.exe, the program may decide to suspend the father process Thread, increasing the possibility that the program is using the previous anti debugging check.

Jmp Headers technique

This technique has been observed in mew11 by looking at the logs and looking at the address of the executed functions (Fig 4.4).

This method makes use of the peculiar structure of a PE file. Starting at virtual address 0x400000, the PE loads the code of the executing application into memory. The headers take up the first portion of memory, which is loaded in read-only mode. The code and other parts normally begin at 0x401000 and operate in both writing and reading modes. This method involves placing executable code in headers and jumping to it at the start of the process. The debugger is unable to interact with the code or write to the memory as a result of this. The common structure indicated before may be observed in Fig. 4.4. The implementation is basic but effective: it checks for a jmp to the protected region at the start of every instrumented function and outputs it in the results.

```
// trigger the instrumentation routine for each instruction
void Instruction(INS ins, void* v) {
    ADDRINT curEip = INS_Address(ins);
    if (proc_info->getFirstINSaddress() == curEip) {
        Config::reached_entry_point = TRUE;
    }
    std::string line = INS_Disassemble(ins).c_str();
    if (line >"jmp 0x400000" && line < "jmp 0x401000") {
        MYTEST("[ANTI-DEBUGGING] JMP HEADERS technique");
    }
    if (config->ANTIEVASION_MODE) {
        thider->avoidEvasion(ins);
    }
    if (Config::reached_entry_point && !entryPointSet) {
        fprintf(Config::getInstance()->getTestFile(),
            "------------EntryPoint----------\n");
        fflush(Config::getInstance()->getTestFile());
        entryPointSet = TRUE;
    }
}
```

Figure 4.4: Implementation of the jmp headers technique inside Pin.

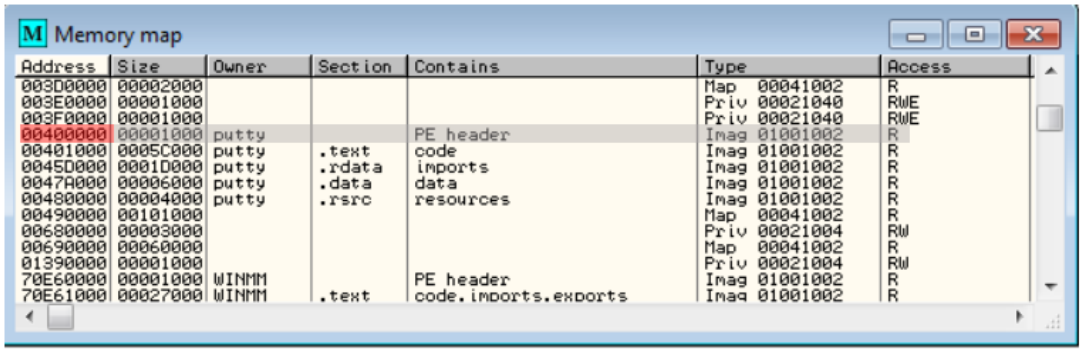

Figure 4.5: Example of common structure of a PE file, evidencing the starting address of the Headers section . In this case we took the program Putty as an example and debugged it with x64dbg.

Guard Pages Exception

This technique regards the EXCEPTION_GUARD_PAGE (0x80000001) and it is expressly used to check if the program is running under the control of OllyDB debugger. This technique consists in registering an exception handler for the guard page exception, allocating a writable/readable memory, and inserting a C3 instruction (RET) on it.

After doing this, it is needed to change the protection of the allocated memory to Page Guard. When this function is called, it will raise an exception and if it doesn't trigger the handler, it means that it has been intercepted by a debugger.
The Guard Pages are inaccessible allocated memory used for defending the code from buffer overflow attempts and heap attacks. Because of this, trying to access a part of this memory will trigger the exception and try to end the program. This technique is implemented in brioscia by intercepting the exception, checking if the exception code is the guard page one, and then writing the result inside the logs.

5 | Analysis of the results

In this chapter, we will discuss the findings obtained from the experiments. This is the research's main focus, and it will demonstrate through the data the use of anti-debugging by debuggers and how deeply it is engaged in their usage in packers. The results are grouped into categories (Anti db, Anti vm, File, Registry, and Time-Stalling), each focusing on a different component of the running program in order to detect a debugger. In section 3.1 it can be found the description of the categorization applied and the explanation of the motivation.
At the end of this chapter, the results divided by category will be joined and it will provide an estimation of the effective packers implementing at least an anti dynamic analysis technique (Section 5.6) .
For every section it will provide a distinction of the techniques analyzed in **False Positive, Uncertain and Secure**, and statistics about the obtained logs. This option between the Certain and Uncertain technique was picked after determining whether the presence of a function in the logs indicates the employment of an anti-debugging approach, or if it may be used for other purposes and there are no additional clues to aid the decision. False positives are examples in which the number of logs produced too much noise in the findings and plainly did not represent the truth of the facts; as a result, we deemed these functions unsuitable and excluded them from the final results.

The subdivision in a different category is represented by symbols as follow:
- ○ **False Positive**
- ◐ **Uncertain**
- ● **Certain**

5.1. Anti Debugging results

ANTIDEBUG	
CheckRemoteDebuggerPresent,	●
FindWindow(classname: filemonclass, procmon, regmonclass),	●
FindWindow(windowname: null, classname: shell_traywnd),	●
GetWindowThreadProcessId,	◐
Instruction: 0xf1 - IceBP,	●
Instruction: INT 1,	●
Instruction: INT 3,	●
Instruction: POPF/D - TRAP FLAG SET,	●
IsDebuggerPresent,	●
JMP HEADERS technique,	●
Memory-R: PEB->IS_DEBUGGED,	●
Memory-R: PEB->NTGLOBALFLAG,	●
NtClose(INVALID_HANDLE),	●
NtGetContextThread(CONTEXT_DEBUG_REGISTERS),	●
NtQueryInformationProcess(0x07),	●
NtQueryInformationProcess(0x1e),	●
NtQuerySystemInformation(0x23),	●
NtSetInformationThread(0x11),	●
Process32Next,	◐
SetUnhandledExceptionFilter,	○
SuspendThread,	◐
BlockInput,	●

Table 5.1: Categorization of the Anti Debugging techniques. ○False Positive, ◐Uncertain Technique,●Certain technique.

Table 5.1 shows all the functions found in the logs related to an anti-debugging technique and the assigned category. The majority of the techniques are certain to be used as an anti-debug trick, as they are functions or assembly lines specifically designed to collect debugging information. The only uncertain functions detected are **SuspendThread, Process32First, Process32Next,GetWindowThreadProcessId**. These are functions that can be used to check the running process and the father process. A single call to one of these routines may be a false positive, but numerous calls can be linked to show that a running anti dbi trick is active. This is because their presence does not guarantee that the program is doing anti-debugging; they may be part of one of these methods, but they necessitate a more thorough examination of the logs and the running program 6.2.1 A pattern that can be extrapolated from the table 5.1 is that eight of the strategies are

based on looking at registers and flags in the PEB structure, making them the most used among all the samples of packers. Techniques based on exception handling are ranked second, with six different techniques belonging to the same family. The rest of the techniques are using different sources and don't evidence any particular pattern.
This finding demonstrates that, although being one of the oldest techniques, the PEB structure and debugging flags methods remain the most popular, indicating a slow verticalization towards modern techniques.

5.1.1. Logs and statistics for Anti Debugging functions

function/packers	Alter	mpress	pecomp	obsid	enigpr	rlp	vmp	telock	pelock	pcguard
CheckRemoteDebuggerPresent,									✓	
FindWindow(classname: filemonclass, procmon, regmonclass),										
FindWindow(windowname: null, classname: shell_traywnd),										
GetWindowThreadProcessId,				✓	✓				✓	✓
Instruction: 0xf1 - IceBP,								✓		
Instruction: INT 1,				✓						
Instruction: INT 3,										
Instruction: POPF/D - TRAP FLAG SET,							✓	✓		
IsDebuggerPresent,					✓				✓	✓
JMP HEADERS technique,										
Memory-R: PEB->IS_DEBUGGED,							✓			✓
Memory-R: PEB->NTGLOBALFLAG,										
NtClose(INVALID_HANDLE),							✓			
NtGetContextThread(CONTEXT_DEBUG_REGISTERS),									✓	
NtQueryInformationProcess(0x07),							✓			
NtQueryInformationProcess(0x1e),							✓			
NtQuerySystemInformation(0x23),							✓			✓
NtSetInformationThread(0x11),					✓		✓			
Process32Next,										
SetUnhandledExceptionFilter,	✓	✓			✓	✓				
SuspendThread,										
BlockInput										

function/packers	kkrun	upx	exe32	enigvm	yp	them	mew11	asprot	petite	aspack
CheckRemoteDebuggerPresent,						✓				
FindWindow(classname: filemonclass, procmon, regmonclass),						✓				
FindWindow(windowname: null, classname: shell_traywnd),					✓					
GetWindowThreadProcessId,						✓		✓		
Instruction: 0xf1 - IceBP,										
Instruction: INT 1,										
Instruction: INT 3,					✓					
Instruction: POPF/D - TRAP FLAG SET,										
IsDebuggerPresent,			✓		✓	✓		✓		
JMP HEADERS technique,							✓			
Memory-R: PEB->IS_DEBUGGED,										
Memory-R: PEB->NTGLOBALFLAG,						✓				
NtClose(INVALID_HANDLE),										
NtGetContextThread(CONTEXT_DEBUG_REGISTERS),						✓				
NtQueryInformationProcess(0x07),					✓	✓				
NtQueryInformationProcess(0x1e),						✓				
NtQuerySystemInformation(0x23),										
NtSetInformationThread(0x11),						✓				
Process32Next,					✓					
SetUnhandledExceptionFilter,	✓	✓	✓		✓		✓		✓	✓
SuspendThread,					✓					
BlockInput					✓					

Table 5.2: Anti debugging logs' results.

The table 5.2 displays the logs obtained after running Brioscia and parsing the results; it contains the anti debugging techniques discovered on the row level, as well as the packer that implements the technique on the column level. As we can notice, there are multiple hits in various techniques, and it demonstrates the first proof of the link between packers and anti debugging. This table itself it's relevant because we can focus on every packer and evidence of which types of techniques are involved. This can be taken as the starting point for other research on a single specific packer, as well as used for new unpacking routines and programs involving packers. Starting from the table 5.2 , it was possible to obtain statistics about packers and the usage of anti debugging, like the number of techniques involved by each packer or the number of occurrences of the same function.

AntiDb function	Total
SetUnhandledExceptionFilter	11
IsDebuggerPresent	7
GetWindowThreadProcessId()	6
NtQueryInformationProcess(0x07)	3
NtSetInformationThread(0x11)	3
NtQuerySystemInformation(0x23)	2
POPF/D - TRAP FLAG SET	2
PEB->IS_DEBUGGED	2
NtGetContextThread(DEBUG_REGISTERS)	2
NtQueryInformationProcess(0x1e)	2
CheckRemoteDebugger	2
FindWindow(file_regmon)	1
FindWindow(shell_traywnd)	1
IceBP	1
INT 1	1
INT 3	1
JMP HEADERS	1
PEB->NTGLOBALFLAG	1
NtClose()	1
Process32Next	1
SuspendThread	1
BlockInput	1

Table 5.3: Number of times a technique appears in the packers Data Set.

SetUnhanledExceptionFilter owns the top position in table 5.3, with eleven results. The truth is that this technique has been classified as a False Positive, and the high number of occurrences, when compared to the results of other procedures, was the initial motivation for digging deeper and determining the cause (Section 6.1.1). We can see that **IsDebuggerPresent** follows in position, which is an expected result for a solution that is easy to create and effective against inexperienced reverse engineers. Every technique's number of occurrences reduces as the number position in the table decreases, but the variety increases, demonstrating that the packers applied numerous ways to avoid a debugger. We could also add to the result of this technique, the results of **CheckRemoteDebuggerPresent**, because they are very similar, reaching a total number of 9 occurrences, almost half of the packers studied.
A special mention goes to **FindWindow**, used in this case by **Themida and Yoda's protector** to check the presence of Procmon (Sysinternal debugging library) and the shell process.

	Alternate	mpress2	pecompact	obsidium	enigpr	rlp	vmp	telock	pelock	pcguard	✓
●+ ◐	0	0	0	2	3	0	7	2	4	4	6
●	0	0	0	1	2	0	7	2	3	3	6

	kkrunchy	upx	exe32	enigvm	yp	themida	mew11	asprot	petite	aspack	✓
●+ ◐	0	0	1	0	8	9	1	2	0	0	5
●	0	0	1	0	5	8	1	1	0	0	4

Table 5.4: Packers exploiting Anti Debugging techniques.
✓ indicates the number of packers exploiting at least one technique.
◐Uncertain techniques. ●Certain techniques.

The final table, 5.4, provides a summary based on the information in the first section. It shows a first overview of the number of packers implementing at least one anti debugging technique, in the case we consider both uncertain and working cases, or only the cases that are 100% sure to be anti debugging.
The table demonstrates that the number of packers implementing a technique is at least half of the total data set. The total number of packers varies between 10 and 11 depending on whether you take an optimistic or pessimistic approach.
In this case, the packers which demonstrated to use the most number of techniques are **Themida, Yoda's protector,virtual machine protector and Pc guard**, but stand alone cases like mew11 and telock will be demonstrated to use a less but effective number of techniques (Chap 6).

5.2. Anti VM results

This section is dedicated to the results linked to Anti VM techniques, from the categorization to the effective results parsed in the logs.

ANTIVM	
GetVersion	◐
GetAdaptersInfo	◐
GetComputerNameA	◐
GetComputerNameW	◐
GetCursorPosition	●
GetDiskFreeSpace	◐
GlobalMemoryStatusEx	●
CPUID(eax=0x00000001)	○
IN(0x564d5868, 0x00005658),Instruction: IN(0x68584d56, 0x77875856)	●
Instruction: SLDT,Instruction: SLDT	●
NtQuerySystemInformation(PHYSICAL_MEMORY_INFO)	○
NtQuerySystemInformation(Process:vbox)	●
NtQuerySystemInformation(SYSTEM_PROCESS_INFORMATION)	●

Table 5.5: Categorization of the Anti VM techniques.
○False Positive, ◐Uncertain Technique,●Certain technique.

The anti virtual machine techniques are more recognizable in case of false positives or actual hits. The motivation is that the software tries to look for information linked to emulated surroundings, this sort of activity is absolutely absent in the original code, and there is no other justification for explaining the appearance of these methods. Some of these are labeled as uncertain functions only because they are not born or used only for anti VM purposes.

An example are **GetComputerName, GetAdaptersInfo, GetDiskFreespace, CPUID** . These functions retrieve information on the host machine, but they can be used to compare common virtual machine settings and names (Example the default computer name, low static and dynamic memory etc..). As a result, we opted to classify them as uncertain situations rather than real techniques, but their single appearance in an example that prints the string "Hello world!" is odd.

On the other hand, there are techniques labeled for sure as anti debugging, like **GetCursorPosition** which appears in packed programs that don't need the information related to the cursor position, or **In(x,y)** that is a well-known technique to check the signature

of Vmware or Virtual box.
The lone false positive in this section is **NtQuerySystemInformation(PHYSICAL MEMORYINFO)**, which is caused by the high number of occurrences and challenging capability to discriminate between genuine case and false positive, as well as **CPUID**, both discussed in Chapter 6.

5.2.1. Logs and statistics for Anti Debugging functions

A,N,T,I,V,M	Alter	mpress	pecomp	obsid	enigpr	rlp	vmp	telock	pelock	pcguard
GetVersion					✓					
GetAdaptersInf										✓
GetComputerNameA					✓					
GetComputerNameW					✓					
GetCursorPosition				✓	✓					
GetDiskFreeSpace					✓					
GlobalMemoryStatusEx					✓				✓	✓
Instruction: CPUID(eax=0x00000001)	✓	✓		✓	✓	✓	✓			
Instruction: IN(X,Y)										
Instruction: SLDT,Instruction: SLDT										
NtQuerySystemInformation(PHYSICAL_MEMORY_INFO)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
NtQuerySystemInformation(Process:vbox)										
NtQuerySystemInformation(SYSTEM_PROCESS_INFORMATION)									✓	

ANTIVM	kkrun	upx	exe32	enigvm	yp	them	mew11	asprot	petite	aspack
GetVersion				✓	✓	✓				
GetAdaptersInfo										
GetComputerNameA										
GetComputerNameW										
GetCursorPosition										
GetDiskFreeSpace				✓				✓		
GlobalMemoryStatusEx						✓		✓		
Instruction: CPUID(eax=0x00000001)	✓	✓	✓				✓			✓
Instruction: IN(X,Y)						✓				
Instruction: SLDT,Instruction: SLDT										
NtQuerySystemInformation(PHYSICAL_MEMORY_INFO)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
NtQuerySystemInformation(Process:vbox)					✓					
NtQuerySystemInformation(SYSTEM_PROCESS_INFORMATION)					✓					

Table 5.6: Anti VM logs' results.

The table 5.6 shows multiple hits,especially for packers like **Themida**, **Enigma protector** or **PcGuard**. On the other hand, it confirms that packers like **upx,mpress or Alternate** have no hits or only false positives. This is consistent with the previous section on anti-debugging, where the same packers failed to use any meaningful approach, suggesting that they are primarily interested in using compression and obfuscation as a means of evading dynamic analysis.

ANTIVM	Total
NtQuerySystemInformation(PHYSICAL_MEMORY_INFO)	20
CPUID(eax=0x00000001)	11
GlobalMemoryStatusEx	5
GetVersion	4
GetDiskFreeSpace	3
NtQuerySystemInformation(Process:vbox)	2
GetCursorPosition	2
GetAdaptersInfo	1
GetComputerNameA	1
GetComputerNameW	1
IN(0x564d5868, 0x00005658),Instruction: IN(0x68584d56, 0x77875856)	1
Instruction: SLDT,Instruction: SLDT	1
NtQuerySystemInformation(SYSTEM_PROCESS_INFORMATION)	1

Table 5.7: Number of times a technique appears in the packers Data Set

The table 5.7 shows several techniques: there are techniques based on the emulated hardware, techniques looking at the processes, and other techniques that looks for common virtual machines configuration. Excluding the false positives, at the top of the list, there is CPUID. Despite the fact that the huge number of instances practically qualifies it as a false positive, its use as an anti-VM strategy remains an effective and quick check that numerous packers can utilize. We decided to leave it out of the findings since it generates too much noise, and we believe that if some packer is using anti-VM, it should implement it using other additional methods. Following, the most used ones are techniques related to the emulated hardware (**GlobalMemoryStatusEx** and **GetDiskFreeSpace**) , followed by common figuration of a virtual machine (**GetVersion**) and finally human interactions functions (**GetCursorPosition**). Last but not least, there are single cases that are specifically designed to avoid emulation at the end of the list.
NtQuerySystemInformation(vbox) deserves special attention since it is a clear evidence of anti-VM and collects all of the packer's attempts to locate Vbox processes, but it will be discussed in Section 6.

	Alternate	mpress2	pecompact	obsidium	enigpr	rlp	vmp	telock	pelock	pcguard	✓
●+ ◐	0	0	0	1	6	0	0	0	2	2	4
●	0	0	0	1	2	0	0	0	2	1	4

	kkrunchy	upx	exe32	enigvm	yp	themida	mew11	asprot	petite	aspack	✓
●+ ◐	0	0	0	2	3	3	0	2	0	0	4
●	0	0	0	0	2	2	0	1	0	0	3

Table 5.8: Packers exploiting Anti VM techniques.
✓ indicates the number of packers exploiting at least one technique.
◐Uncertain techniques. ●Certain techniques.

The table 5.8 describes the final results regarding anti VM, confirming that almost half of the packers, certainly adopt some anti VM technique. Looking at the results, and comparing them to the results in section 5.1, we can notice that the packers that involve a secure anti VM technique, are included in the packers that use at least one anti-debugging technique.This indicates that packers who choose to not only compress but also protect the exe, are planned to shield from all the possible technologies of dynamic analysis.

5.3. File results

FILE	
NtCreateFile(\\??\\global\\procmondebuglogger)	●
NtCreateFile(\\??\\ntice),	●
NtCreateFile(\\??\\sice)	●
NtCreateFile(\\??\\siwvidstart),	●
NtCreateFile(\\??\\spcommand)	●
NtCreateFile(\\??\\syser),	●
NtCreateFile(\\??\\syserboot)	●
NtCreateFile(\\??\\syserdbgmsg)	●
NtCreateFile(\\??\\global\\procmondebuglogger)	●

Table 5.9: Categorization of the File techniques.
○False Positive, ◐Uncertain Technique,●Certain technique.

In the file section, all of the categories are deemed to be working anti debugging approaches, as shown in the table 5.9. The motivation is that the libraries opened with create files are all part of Sysinternals (a collection of utilities for managing registries, processes, and dynamic data about programs), or they are all libraries commonly used by debuggers. It's also worth noting that some of the packers go straight for Procmon, the most often used software for performing dynamic analysis at runtime.
The last piece of information worth to mention is that all the files are related to anti-debugging functions rather than anti vm, giving additional proof of the concepts described in section 5.1.

5.3.1. Logs and statistics for Files management

FILE	Alter	mpress	pecomp	obsid	enigpr	rlp	vmp	telock	pelock	pcguard
NtCreateFile(\\??\\global\\procmondebuglogger)										✓
NtCreateFile(\\??\\ntice)				✓					✓	
NtCreateFile(\\??\\sice)				✓						
NtCreateFile(\\??\\siwvidstart)									✓	
NtCreateFile(\\??\\spcommand)									✓	
NtCreateFile(\\??\\syser)									✓	
NtCreateFile(\\??\\syserboot)									✓	
NtCreateFile(\\??\\syserdbgmsg)									✓	
NtCreateFile(\\??\\global\\procmondebuglogger)										✓

ANTIVM	kkrun	upx	exe32	enigvm	yp	them	mew11	asprot	petite	aspack
NtCreateFile(\\??\\global\\procmondebuglogger)										
NtCreateFile(\\??\\ntice)					✓					
NtCreateFile(\\??\\sice)					✓					
NtCreateFile(\\??\\siwvidstart)										
NtCreateFile(\\??\\spcommand)										
NtCreateFile(\\??\\syser)										
NtCreateFile(\\??\\syserboot)										
NtCreateFile(\\??\\syserdbgmsg)										
NtCreateFile(\\??\\global\\procmondebuglogger)										

Table 5.10: File logs' results.

In this study case, there are fewer results compared to the previous sections. Studying the results in the table 5.10, the only packers looking for Indicators in the files are **Obsidium, Pelock, Pcguard, and Yoda's Protector**. This discovery is consistent with the prior ones, as they are all main characters in the preceding tables.
It must also be demonstrated how **Pelock** attempts to open the file descriptor for practically every well-known debugger library, making him the most advanced in this part.

FILE	Total
NtCreateFile(\\??\\ntice),	3
NtCreateFile(\\??\\sice)	2
NtCreateFile(\\??\\siwvidstart),	1
NtCreateFile(\\??\\global\\procmondebuglogger)	1
NtCreateFile(\\??\\spcommand)	1
NtCreateFile(\\??\\syser),	1
NtCreateFile(\\??\\syserboot)	1
NtCreateFile(\\??\\syserdbgmsg)	1
NtCreateFile(\\??\\global\\procmondebuglogger)	1

Table 5.11: Number of times a technique appears in the packers Data Set.

Only **Pelock** looks for all the libraries used by a debugger, but the other packers only look for one of them, as shown in Table 5.9. It's possible that when the other packer reaches one of the libraries specified, they stop this type of check, whereas **Pelock** enumerates them all.

	Alternate	mpress2	pecompact	obsidium	enigpr	rlp	vmp	telock	pelock	pcguard	✓
●+ ◐	0	0	0	2	0	0	0	0	6	2	3
●	0	0	0	2	0	0	0	0	6	2	3

	kkrunchy	upx	exe32	enigvm	yp	themida	mew11	asprot	petite	aspack	✓
●+ ◐	0	0	0	0	2	0	0	0	0	0	1
●	0	0	0	0	2	0	0	0	0	0	1

Table 5.12: Packers exploiting File techniques.
✓ indicates the number of packers exploiting at least one technique.
◐Uncertain techniques. ●Certain techniques.

At the end of this section, the results that imply checks in the File usage are 4 out of 20. It is a good result because obtaining this type of log from the original hello world file is totally unexpected and provides decisive information for the additional feauture given by packers (Table 5.11 and 5.12). We predicted fewer findings in this part because our pool of methods does not include a big number of techniques utilizing File, and we parsed just the logs that had a high possibility of being engaged with anti debugging out of the massive number of files descriptors.

5.4. Registry results

REGISTRY	
NtOpenKey/Ex(\\registry\machine\hardware\acpi\dsdt\vbox)	●
NtOpenKey/Ex(\\registry\??\currentversion\image file execution options\ProgramName.exe)	●
NtOpenKey/Ex(\\registry\machine\software\wine\wine\config)	●
NtOpenKey/Ex(\\registry\??\control\computername\activecomputername)	●
NtQueryValueKey(\registry\??\system, videobiosversion)	●
NtQueryValueKey(\\registry\??\centralprocessor\0, identifier) sz = intel64 family 6)	◐
NtQueryValueKey(\registry\??\activecomputername, computername) sz = msedgewin10)	●
NtQueryValueKey(\\registry\??\system, systembiosversion) = vbox - 1)	●
NtQueryValueKey(\\registry\??\system, videobiosversion) = oracle vm virtualbox bios)	●
NtQueryValueKey(\\registry\??\0000, driverdesc) = virtualbox graphics adapter (wddm)	●
NtQueryValueKey(\\registry\??\disk\enum, 0) = ide\\diskvbox_harddisk)	●

Table 5.13: Categorization of the Registry techniques.
○False Positive, ◐Uncertain Technique,●Certain technique.

Values useful for anti-debugging and anti-VM can be found in the registries. The table 5.13 demonstrates that the majority of the checks are for detecting the presence of a Virtual Box machine, but it also includes the **ProgramName.exe** function, which is one of the techniques for looking for the NT debugging flag, and an inspection for the process being debugged by wine. The control on the CPU version is the only category that is designated as Uncertain (Intel 64). It is classified like this because programs can check its value for compatibility issues, but it is also utilized by packers who run a more stringent check on anti-VM, ensuring that it is with high probability used as anti-VM. Adding it as certain or doubtful at the conclusion of the findings didn't impact the final output, but it's meaningful to show how it might be simply considered as an anti-VM strategy.

5.4.1. Logs and statistics for Registries management

REGISTRY	Alter	mpress	pecomp	obsid	enigpr	rlp	vmp	telock	pelock	pcguard
NtOpenKey/Ex(\\registry\machine\hardware\acpi\dsdt\vbox)										
NtOpenKey/Ex(\\registry\??\currentversion\image file execution options\ProgramName.exe)										
NtOpenKey/Ex(\\registry\machine\software\wine\wine\config										✓
NtOpenKey/Ex(\\registry\??\control\computername\activecomputername)					✓				✓	✓
NtQueryValueKey(\registry\??\system, videobiosversion)										✓
NtQueryValueKey(\\registry\??\centralprocessor\0, identifier) sz = intel64 family 6)										✓
NtQueryValueKey(\registry\??\activecomputername, computername) sz = msedgewin10)					✓				✓	✓
NtQueryValueKey(\\registry\??\system, systembiosversion) = vbox - 1)										✓
NtQueryValueKey(\\registry\??\system, videobiosversion) = oracle vm virtualbox bios)										✓
NtQueryValueKey(\\registry\??\0000, driverdesc) = virtualbox graphics adapter (wddm)										
NtQueryValueKey(\\registry\??\disk\enum, 0) = ide\\diskvbox_harddisk)										✓

REGISTRY	kkrun	upx	exe32	enigvm	yp	them	mew11	asprot	petite	aspack
NtOpenKey/Ex(\\registry\machine\hardware\acpi\dsdt\vbox)						✓				
NtOpenKey/Ex(\\registry\??\currentversion\image file execution options\ProgramName.exe)				✓		✓				
NtOpenKey/Ex(\\registry\machine\software\wine\wine\config)										
NtOpenKey/Ex(\\registry\??\control\computername\activecomputername)						✓		✓		
NtQueryValueKey(\registry\??\system, videobiosversion)										
NtQueryValueKey(\\registry\??\centralprocessor\0, identifier) sz = intel64 family 6)										
NtQueryValueKey(\registry\??\activecomputername, computername) sz = msedgewin10)						✓		✓		
NtQueryValueKey(\\registry\??\system, systembiosversion) = vbox - 1)						✓				
NtQueryValueKey(\\registry\??\system, videobiosversion) = oracle vm virtualbox bios)						✓				
NtQueryValueKey(\\registry\??\0000, driverdesc) = virtualbox graphics adapter (wddm)						✓				
NtQueryValueKey(\\registry\??\disk\enum, 0) = ide\\diskvbox_harddisk)										

Table 5.14: Registry logs' results.

The packers who implemented the most number of anti VM checks in this section are **PCGuard and Themida**. In particular, Themida looks for evidence that is linked to Virtual box:
- It checks for the pc name to be msedgewin10, which is the default name for a windows virtual machine.
- It checks for the videobiosversion, in particular, "Oracle VM virtual Box" which is the value while using the Virtual Box virtual machine.
- It checks for the systembios version, which is "Vbox - 1" for a Virtual Box machine.
- It checks for the virtual box graphic adapter wddm.
PcGuard performs the same checks, with the addition of an inspection for the presence of the windows debugger wine and the virtual box disk.

REGISTRY	
NtOpenKey/Ex(\\registry\??\control\computername\activecomputername)	5
NtQueryValueKey(\registry\??\activecomputername, computername) sz = msedgewin105	2
NtOpenKey/Ex(\\registry\??\currentversion\image file execution options\ProgramName.exe)	2
NtOpenKey/Ex(\\registry\machine\hardware\acpi\dsdt\vbox)	1
NtOpenKey/Ex(\\registry\machine\software\wine\wine\config)	1
NtQueryValueKey(\registry\??\system, videobiosversion)	1
NtQueryValueKey(\\registry\??\centralprocessor\0, identifier) sz = intel64 family 6)	1
NtQueryValueKey(\\registry\??\system, systembiosversion) = vbox - 1)	1
NtQueryValueKey(\\registry\??\system, videobiosversion) = oracle vm virtualbox bios)	1
NtQueryValueKey(\\registry\??\0000, driverdesc) = virtualbox graphics adapter (wddm)	1
NtQueryValueKey(\\registry\??\disk\enum, 0) = ide\\diskvbox_harddisk)	1

Table 5.15: Number of times a technique appears in the packers Data Set.

The most used technique in registries control is the check performed for the active computer name. This registry as a result can easily spot the presence of VMbox and VMware or the default settings of an emulated environment. The second place is occupied by the NT flag discovery in the Image File Execution directory, and following there are all

the virtual box environment checks. The table 5.15 doesn't show any particular pattern, despite the large use of the machine computer name compared to the other remaining findings.

	Alternate	mpress2	pecompact	obsidium	enigpr	rlp	vmp	telock	pelock	pcguard	✓
●+ ◐	0	0	0	0	2	0	0	0	2	8	3
●	0	0	0	0	2	0	0	0	2	7	3

	kkrunchy	upx	exe32	enigvm	yp	themida	mew11	asprot	petite	aspack	✓
●+ ◐	0	0	0	1	0	7	2	0	0	0	3
●	0	0	0	1	0	7	2	0	0	0	3

Table 5.16: Packers exploiting Registry techniques.
✓ indicates the number of packers exploiting at least one technique.
◐Uncertain techniques. ●Certain techniques.

5.5. Stalling and Timing results

STALLING	
NtDelayExecution()	●
SetTimer()	●
Sleep/SleepEx()	●
waitForSingleObject/Ex()	●
TIMING	
GetLocalTime	◐
GetTickCount	◐
GetTickCount64	○
Instruction: RDTSC/D	●
QueryPerformanceCounter	○
timeGetTime	◐

Table 5.17: Packers exploiting Registry techniques.
○False Positive, ◐Uncertain Technique,●Certain technique.

In this section, it will be treated Stalling and Timing. They both are related to the time spent on the execution of a program and they both include a small number of functions, making it possible to discuss them together.
In this category, the stalling part doesn't show any uncertain technique. All the functions used are repeated several times into the logs of the packers and are not stand-alone functions. Such a behavior perfectly fits the final scope of a stalling process, slowing down the dynamic analysis and avoiding sandboxes that are planned to have a limited amount of time for the execution.
On the other hand, the Timing sections show completely the opposite behavior. All the functions are uncertain or false positives because are commonly used during the normal execution of a program. The uncertain functions are not labeled as False Positives because they perfectly match the results packers that already used multiple anti-VM and anti-debugging techniques, making it reasonable to think that they are also involved in measuring the timing of functions for anti-debugging purposes.

5.5.1. Logs and statistics for Registries management

STALLING	Alter	mpress	pecomp	obsid	enigpr	rlp	vmp	telock	pelock	pcguard
NtDelayExecution()					✓					✓
SetTimer()					✓				✓	✓
Sleep/SleepEx()					✓					
waitForSingleObject/Ex()					✓				✓	✓
TIMING										
GetLocalTime					✓				✓	
GetTickCount				✓	✓				✓	✓
GetTickCount64	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Instruction: RDTSC/D				✓			✓			✓
QueryPerformanceCounter	✓	✓	✓		✓	✓				✓
timeGetTime										

STALLING	kkrun	upx	exe32	enigvm	yp	them	mew11	asprot	petite	aspack
NtDelayExecution()						✓				
SetTimer()						✓		✓		
Sleep/SleepEx()						✓				
waitForSingleObject/Ex()				✓		✓		✓		
TIMING										
GetLocalTime								✓		
GetTickCount				✓	✓	✓		✓		
GetTickCount64	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Instruction: RDTSC/D						✓				
QueryPerformanceCounter	✓	✓	✓	✓			✓			✓
timeGetTime						✓				

Table 5.18: Stalling/Timing logs results.

The stalling section of table 5.18 reveals that two packers, **Enigma protector and Themida**, make extensive use of stalling tactics.
In the timing section , removed the noise produced by false positives, we have a similar results, where the same packers that adopt stalling, adopted also a timing procedure.

STALLING	Total
waitForSingleObject/Ex()	6
SetTimer()	5
NtDelayExecution()	3
Sleep/SleepEx()	2

TIMING	Total
GetTickCount64	20
QueryPerformanceCounter	12
GetTickCount	8
Instruction: RDTSC/D	4
GetLocalTime	3
timeGetTime	1

Table 5.19: Parameters needed for things that are not needed anymore themselves.

The last table, 5.20, demonstrates how the number of packers involved in timing and delaying is nine out of twenty, whereas excluding uncertain techniques we obtain a result of eight out of twenty. Both of the results are close to the half of packers, a result which is aligned with the other sections.

	Alternate	mpress2	pecompact	obsidium	enigpr	rlp	vmp	telock	pelock	pcguard	✓
●+ ◐	0	0	0	2	6	0	1	0	4	5	5
●	0	0	0	1	4	0	1	0	2	4	5

	kkrunchy	upx	exe32	enigvm	yp	themida	mew11	asprot	petite	aspack	✓
●+ ◐	0	0	0	2	1	7	0	4	0	0	4
●	0	0	0	1	0	5	0	2	0	0	3

Table 5.20: Packers exploiting Stalling\Timing techniques.
✓ indicates the number of packers exploiting at least one technique.
◐Uncertain techniques. ●Certain techniques.

5.6. Final results

The next Tables (5.21 and 5.22) summarize all the findings of the previous section. The scope is to give an estimation of how many packers are implementing at least an anti-debugging or anti VM technique.
The first table contains all the results obtained by selecting both certain and uncertain techniques, while the second table contains only the certain technique. The scope is to demonstrate that packers are using anti dynamic analysis methods, but also that despite the categorization, the results are consistent.
Finding radically different results in the two examples would indicate that the uncertain techniques were most likely false positives or that the categorization was incorrect.

Instead, if identical findings are obtained, it will be proved that the uncertain techniques were most likely utilized for anti-debugging objectives, as they are used by packers who are already employing more certain and reliable techniques. In our example, we got the best result we could, proving that 12 out of 20 packers are performing anti-analysis, regardless of whether we add or remove functions that may be utilized for other purposes. The final table proves the usage of anti-debugging by more than half of the data set, demonstrating that packers are commonly adding anti-debugging beyond compression and obfuscation. We also demonstrated that all the uncertain techniques are probably used for anti-analysis purposes, giving more information about which and how many functions are used by the packers in our data set.

Packers implementing Anti Dynamic Analysis ●+ ◐

	Alternate	mpress2	pecompact	obsidium	enigpr	rlp	vmp	telock	pelock	pcguard
ANTIDEBUG	0	0	0	2	3	0	7	2	4	4
ANTIVM	0	0	0	1	6	0	0	0	2	2
FILE	0	0	0	2	0	0	0	0	6	2
REGISTRY	0	0	0	0	2	0	0	0	2	8
STALL/TIME	0	0	0	2	6	0	1	0	4	5
RESULT	✗	✗	✗	✓	✓	✗	✓	✓	✓	✓

	kkrunchy	upx	exe32	enigvm	yp	themida	mew11	asprot	petite	aspack
ANTIDEBUG	0	0	1	0	8	9	1	2	0	0
ANTIVM	0	0	0	2	3	3	0	2	0	0
FILE	0	0	0	0	2	0	0	0	0	0
REGISTRY	0	0	0	1	0	7	2	0	0	0
STALL/TIME	0	0	0	2	1	7	0	4	0	0
RESULT	✗	✗	✓	✓	✓	✓	✓	✓	✗	✗

Table 5.21: This table collects for each packer the number of certain and uncertain techniques divided by category, and output if they implements at least one technique.
✓ Indicates if the packer exploits at least one technique, ✗ if not.

Packers implementing Anti Dynamic Analysis ●

	Alternate	mpress2	pecompact	obsidium	enigpr	rlp	vmp	telock	pelock	pcguard
ANTIDEBUG	0	0	0	1	2	0	7	2	3	3
ANTIVM	0	0	0	1	2	0	0	0	2	1
FILE	0	0	0	2	0	0	0	0	6	2
REGISTRY	0	0	0	0	2	0	0	0	2	7
STALL/TIME	0	0	0	1	4	0	1	0	2	4
RESULT	✗	✗	✗	✓	✓	✗	✓	✓	✓	✓

	kkrunchy	upx	exe32	enigvm	yp	themida	mew11	asprot	petite	aspack
ANTIDEBUG	0	0	1	0	5	8	1	1	0	0
ANTIVM	0	0	0	0	2	2	0	1	0	0
FILE	0	0	0	0	2	0	0	0	0	0
REGISTRY	0	0	0	1	0	7	2	0	0	0
STALL/TIME	0	0	0	1	0	5	0	2	0	0
RESULT	✗	✗	✓	✓	✓	✓	✓	✓	✗	✗

Table 5.22: This table collects for each packer the number of certain techniques divided by category, and output if they implements at least one technique.
✓ Indicates if the packer exploits at least one technique, ✗ if not.

✓ Total packers ≡ 12 ✗ Total packers ≡ 8

6 | False Positives and relevant findings

The reasons and research that led to the classification of some functions as False positives will be described in this chapter. Because admitting strategies that are not certain would introduce noise and unbalance the final number of packers involving anti-debugging, this section was crucial for the output in the end results.
The other goal of this chapter, in terms of false positives, is to examine some specific and advanced strategies that were discovered during the experimental phase. They are functions that, despite being listed in the end logs, need more room and have a prominent presence in the involving packer's final logs.

6.1. False Positives

6.1.1. SetUnhandledExceptionFilter()

The first function analyzed and considered as a False Positive is **SetUnhandledExceptionFilter**. A first look into the logs showed that 11 packers out of 20 involved this type of technique. It may appear to produce a result that is consistent with the research's ultimate outcome, but the reality is that it was employed by packers who failed to produce any results throughout the study cases and who appeared to have little interest in anti-debugging.
Going back to the original logs to determine if the function was already utilized by the hello world example was the first index that helped us prove our point. Looking at the logs it can be seen that this type of function is already added at compiling time by the GNU and Visual studio compiler:

```
(*:0:kernel32.dll:setunhandledexceptionfilter)->(*:1:main:)->
(*:1:main:)->(*:0:ucrtbased.dll:initterm)

(*:0:kernelbase.dll:setunhandledexceptionfilter)->
(*:1:main:)->(*:1:main:)->(*:0:ucrtbased.dll:initterm)

(*:0:kernel32.dll:setunhandledexceptionfilter)->(*:1:main:unnamedimageentrypoint)->
(*:1:main:unnamedimageentrypoint)->(*:0:ucrtbased.dll:initterm)

(*:0:kernelbase.dll:setunhandledexceptionfilter)->(*:1:main:unnamedimageentrypoint)->
(*:1:main:unnamedimageentrypoint)->(*:0:ucrtbased.dll:initterm)
```

Listing 6.1: Call tree produced by the SetUnhandledExceptionFilter functions in the original unpacked programs. The logs are printed using different colors to highlight the logs that evidenced similar call trees. .

The other results are extremely similar to the function called in Fig 6.2, and they are not eliminated by the parser because the call tree is shortened or the section name changes, which is another argument that may prove the function's origin.
The second factor that influenced the selection was that the technique relies on throwing an unhandled exception, and none of the packers had an unhandled exception like division by zero or others, but just conventional exceptions that were already connected to a default handler.
Being quite all the logs produced similar, the third and last point to prove the thesis has been analyzing UPX which is a packer that didn't use any other anti-debugging function and trying to find it is possible to unpack it through a debugger. The reason behind the choice of the packer is that UPX it's easy to unpack and , being all the logs quite the same, we could extend the result to the other packers.
As expected, the result showed that it was possible to reach the original code through IDA pro after being unpacked, and reading the "Hello World!" string loaded in memory, as can be shown in figure 6.1. This demonstrates that the packed file could be debugged until reaching the original code and reading the test String.

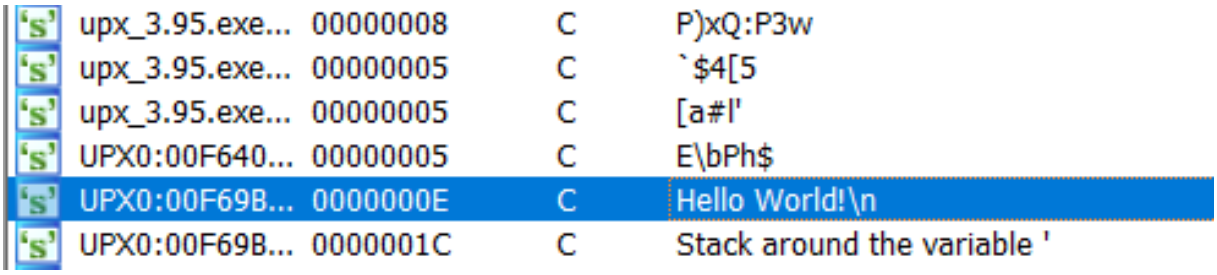

Figure 6.1: Hello World string stored in memory, after the unpacking procedure is ended.

.

6.1.2. NtQuerySystemInformation(PHYSICALMEMORYINFO)

The following function was added to the False Positives for a variety of reasons, the first of which is that it was implemented by all packers. This leads to the conclusion that it might be used for other purposes other than anti-debugging.
The fact that this function is the endpoint of a large pool of other functions linked to normal execution supports this hypothesis. To demonstrate it, we examined the logs for this type of call and discovered roughly a hundred calls involving numerous and distinct methods and call trees.
As a result of this, it was decided that the rule was too naive to be used as an anti-debugging signal.

```
->(*:0:combase.dll:coincrementmtausage)->(*:0:combase.dll:coincrementmtausage)
->(*:0:coremessaging.dll:coreuiconfigureuserintegration)->(*:0:coremessaging.dll:msgrelease)
->(*:0:coremessaging.dll:coreuiconfigureuserintegration)->(*:0:coremessaging.dll:userintegration)
->(*:0:coremessaging.dll:msgrelease)->(*:0:coremessaging.dll:coreuifailfastoom)
->(*:0:rpcrt4.dll:rpcstringfreew)->(*:0:rpcrt4.dll:rpcstringfreew)
```

Listing 6.2: Small part of the large number of functions that make the use of NtQuerySystemInformation(PHYSICALMEMORYINFO) .

6.1.3. CPUID

The CPUID technique is effective and simple to verify; all that is required is to call the assembly function and check one of the bits of the results. This made deciding whether or not to include it in the False Positive set extremely difficult. A trend discovered in the logs proved to be the deciding factor. Looking at them, it's clear that CPUID followed three distinct patterns in the brioscia output:

1) (*:1:main:unnamedimageentrypoint)-> (*:1:main:unnamedimageentrypoint)-> (*:1:main:unnamedimageentrypoint)->(*:1:main:unnamedimageentrypoint)

This first pattern is implemented also by the original program Hello World.exe. It is characterized by a sequence of 4 identical calls building the tree of calls.

2) (*:1:main:.mpress1)-> (*:1:main:.mpress1)->
(*:1:main:.mpress1) -> (*:1:main:.mpress1)

This second one is the pattern found in almost all of the other packers. We can see that it is quite identical to the previous pattern adopted in the original code. The difference is that it inserted the name of the section where is done unpacking instead of the function itself. This could mostly like mean that they are the same functions of the original code but printed in different ways in the output.

3) (*:1:main:)

The last pattern found in the logs is just a call in the main function. This is probably where the false positive is a real anti-VM function. The reason is that for app laying this technique is enough calling a line of assembly and it is coherent with this pattern. It is also implemented by packers which are considered to use for sure anti dynamic analysis techniques, confuting more the thesis. Despite these instances, and because of the significant number of False Positives, we opted to exclude these findings as they were irrelevant to the research's ultimate results.

6.1.4. Other False Positives

The remaining false positives as the one inserted in Timing, are labeled like this because they are uncertain techniques used during common runs of an everyday program and there were no countermeasures to check if they were employed or not as anti-analysis tools. Instead of adding useless noise to the results of such weak techniques, we preferred to exclude them and trust only the more meaning full ones.

6.2. Highlighted findings

6.2.1. Yoda's Protector

This section is dedicated to the packer **Yoda's prtector** since it necessitated a more thorough investigation to determine the procedures involved. The issue with this packer was that it stopped working right at the start of the program, making it difficult to investigate the remaining logs. The issue was caused by an anti-debugging check done by the packer itself, which tested if the process's parent was "Explorer.exe" at the start of execution. If not, it used the **SuspendThread()** method to suspend the application.
Enumerating all active processes and gathering the associated names was how it discovered the PID of Explorer.exe and the PID of the parent process. Pin.exe was the parent process in our situation since it gathered logs while performing instrumentation, as seen in Fig 6.3.

```
->Process32Next(notepad.exe,10108)
->Process32Next(SearchFilterHost.exe,6416)
->Process32Next(pin.exe,6388)
->Process32Next(pin.exe,1608)
->Process32Next(hello_yp.exe,1168)
->Process32Next(svchost.exe,10084)
->Process32Next(svchost.exe,8860)
->Process32Next(SearchProtocolHost.exe,11536)
->Process32Next(SearchProtocolHost.exe,11536)
->CreateToolHelp32Snapshot(-1018964676, 4515593)
->CreateToolHelp32Snapshot(4, 1608)
->OpenProcess(1608)
->Program launched an exception with code 0xc0000005
```

Listing 6.3: Logs evidencing how yoda's protector is enumerating processes to find the descriptor of the current father's process.

The image depicts how the program sifted through the list of processes until it came across **"pin.exe"** and **"hello yp.exe"**, which are the pin and packed programs, respectively. At this point, the list is stopped, the pin's PID is taken, and the process is opened to obtain further information. It raises an exception and exits the program after this call.

After writing a routine that stopped the list of the processes at the beginning instead of being enumerated, the program could continue the execution, reach the end, and output new interesting results like the function **IsDebuggerPresent()** and **BlockInput()**.

6.2.2. Mew11

Mew 11 gets special notice since it is the only one that uses a one-of-a-kind method that necessitated the creation of a custom rule. Because the headers section is a Read-only region, the solution we're talking about is placing code inside it to avoid writing new code in the solution. Analyzing every jump instruction and examining the area it was jumping in might reveal this strategy.
The rule which spotted this technique was also applied to all the other packers, but it didn't reveal any other positive cases. This means that it could be a technique created and implemented only by this packer.

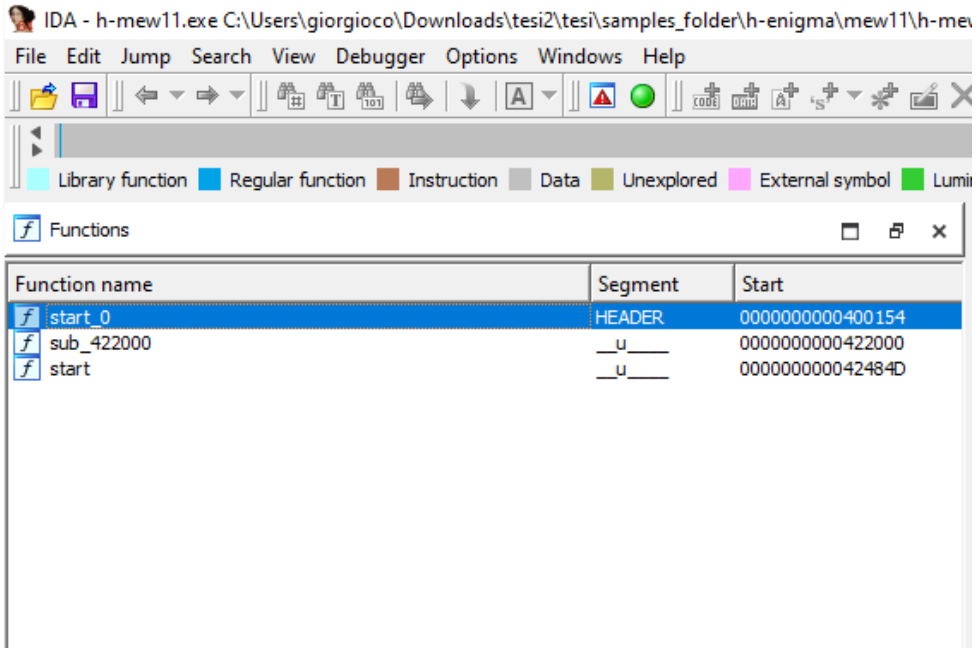

Figure 6.2: Picture taken in IDA pro, it represents the start function pointing to the headers section. This is an additional proof of the correctness of the implemented rule.

6.2.3. Telock

Telock has been included in this list because it displayed unusual behavior when examining logs. The discovery in question was an unusually high number of calls to the **POPF/D - TRAP FLAG instruction**., as it can be seen in Fig 6.4.

```
Instruction: POPF/D - TRAP FLAG SET
Instruction: POPF/D - TRAP FLAG SET
Instruction: POPF/D - TRAP FLAG SET
Instruction: POPF/D - TRAP FLAG SET
Instruction: POPF/D - TRAP FLAG SET
Instruction: POPF/D - TRAP FLAG SET
Instruction: POPF/D - TRAP FLAG SET
Instruction: POPF/D - TRAP FLAG SET
```

Listing 6.4: Part of telock's logs demonstrating that the packer is using interception for counting the instructions.

This evidence was difficult to explain using the instrumentation hooks and other logs, but it may be achievable owing to the study [21], which used dynamic analysis to discover the same approach. This allowed us to not only confirm the findings of the previous article but also get a deeper understanding of how these functions are employed. The trap instruction, as mentioned in the article, is used to count the number of instructions utilized in the program and to verify if there are any extra lines of code. This approach, which is more closely connected to anti-dbi techniques than anti-debugging, is also effective against instrumentation.

6.2.4. NtQuerySystemInformation()

The function **NtQuerySystemInformation()** deserves a mention in this section because it has been used for clearly look into processes and understand if the environment is emulated or it's being performed dynamic analysis. In Fig. 6.5, 6.6, 6.7 can be seen how **Themida**,**Yoda's protector** and **Pelock** are looking for virtual box instances:

```
->NtQuerySystemInformation(PROCESS_INFORMATION) -> VirtualBox.exe
->NtQuerySystemInformation(PROCESS_INFORMATION) -> VBoxSVC.exe
->NtQuerySystemInformation(PROCESS_INFORMATION) -> VBoxSDS.exe
->NtQuerySystemInformation(PROCESS_INFORMATION) -> VirtualBoxVM.exe
->NtQuerySystemInformation(PROCESS_INFORMATION) -> VirtualBoxVM.exe
->NtQuerySystemInformation(PROCESS_INFORMATION) -> VirtualBoxVM.exe
```

Listing 6.5: Part of Themida's logs demonstrating that the packer is looking for Virtual box.

```
->NtQuerySystemInformation(PROCESS_INFORMATION) -> VBoxService.exe
->NtQuerySystemInformation(PROCESS_INFORMATION) -> VBoxTray.exe
->NtQuerySystemInformation(SYSTEM_PROCESS_INFORMATION)
```

Listing 6.6: Part of Yoda's protector logs demonstrating that the packer is looking for Virtual box.

```
->NtQuerySystemInformation(PROCESS_INFORMATION) -> VirtualBox.exe
->NtQuerySystemInformation(PROCESS_INFORMATION) -> VBoxSVC.exe
->NtQuerySystemInformation(PROCESS_INFORMATION) -> VBoxSDS.exe
->NtQuerySystemInformation(PROCESS_INFORMATION) -> VirtualBoxVM.exe
->NtQuerySystemInformation(PROCESS_INFORMATION) -> VirtualBoxVM.exe
->NtQuerySystemInformation(PROCESS_INFORMATION) -> VirtualBoxVM.exe
->NtQuerySystemInformation(SYSTEM_PROCESS_INFORMATION)
```

Listing 6.7: Part of pelock logs demonstrating that the packer is looking for Virtual box.

As it can be seen in the tables 6.6 6.7 6.5, these 3 packers are looking for the presence of virtual box by enumerating and comparing processes utilized by the virtual box

instances. This is clear evidence of anti-VM behavior, and it's been discovered in three separate packers that run comparable checks. For sake of better understanding, we tried to run other processes while **Themida** was running, discovering that it not only looks for virtual machines but also other processes like Procmon (Fig 6.8).

```
->NtQuerySystemInformation(PROCESS_INFORMATION) -> Procmon.exe
->NtQuerySystemInformation(PROCESS_INFORMATION) -> Procmon64.exe
->NtQuerySystemInformation(SYSTEM_PROCESS_INFORMATION)
```

Listing 6.8: Pelock logs evidencing an attempt to find procmon in the processes. .

It may be possible that Themida has a list of interesting processes to look for while searching debugging or virtual machines instances. It could be interesting to run the experiment with different debuggee processes and find out the whole list of the process they are looking for.

6.2.5. NtqueryAttributesFile

One of the most recent discoveries was that **Pelock** searches the disk for the files VBoxOGL.dll and VBoxHook.dll, attempting to access multiple paths. They are common Dynamic Link Libraries used by Virtual Box, and it's fascinating to know that they're utilized by Virtual Box. Pelock implements a list of probable paths where they might be found and enumerates them using the NtqueryAttributesFile function. The next figure (Fig. 6.9) is reported a little sample of the giant pool of paths tested by the program.

```
->NtQueryAttributesFile(?\C:\Program Files\dotnet\VBoxHook.dll)
->NtQueryAttributesFile(\??\C:\Program Files\dotnet\VBoxHook.dll)
->NtQueryAttributesFile(\??\C:\Users\??WindowsApps\VBoxHook.dll)
->NtQueryAttributesFile(\??\C:\Users\??WindowsApps\VBoxHook.dll)
->NtQueryAttributesFile(\??\C:\Users\cocci\.dotnet\tools\VBoxHook.dll)
->NtQueryAttributesFile(\??\C:\Users\cocci\.dotnet\tools\VBoxHook.dll)
->NtQueryAttributesFile(\??\C:\Users\cocci\Downloads\VBoxOGL.dll)
->NtQueryAttributesFile(\??\C:\Users\cocci\Downloads\VBoxOGL.dll)
->NtQueryAttributesFile(\??\C:\Windows\SYSTEM32\VBoxOGL.dll)
```

Listing 6.9: Pelock's logs looking for VBox dlls .

6.2.6. In instruction

This case must be mentioned because it is for sure recognized as an anti-VM technique. The only packer applying an instance of this type of technique is Themida. The VMware virtual machine provides an I/O port communication channel that allows data to be exchanged between the host and guest operating systems. Analysts can get information about I/O ports by using the IN instruction. Furthermore, if analysts perform the IN instruction by entering the value 0x5658 (i.e., VX) into the DX register (i.e., the communication channel), the value comprising the virtual machine's information is placed in the EAX or EBX register. As a result, the IN instruction may be used to detect the virtual machine.

The value checked by themida are **Instruction: IN(0x564d5868, 0x00005658) and Instruction: IN(0x68584d56, 0x77875856)**. Both of the instructions are looking for VMware magic numbers,in particular the endian and endianess values 0x564d5868 and 0x68584d5.

6.2.7. Stalling routines

Packers such as **Enigma Protector, Pelock, PCGuard, Enigma Virtual Machine, Themida, and AsProtect** were shown to have a delaying technique in the logs. The reason behind this is that they didn't utilize the stalling routines just once, for waiting for a single event or for something else in the code. Waiting routines, such as Themida, looked to have entirely contaminated the logs in the samples, with over 19 thousand calls to NtDelayExecution() in a single run. The intriguing thing is that some of these

packers, such as themida, increased the stalling time while executing, whilst others, such as Enigma Protector, kept the same stalling time throughout the execution by repeatedly using the call Sleep(4000) and Sleep(300).

7 | Conclusion and future works

7.0.1. Conclusion

We offered three questions at the start of our study, and now we can evaluate if we were successful in answering them.

RQ1. *Are anti-debugging tactics being used by packers?*

We found that 12 of the 20 packers utilized at least one anti-dynamic analysis approach. This is a gratifying finding that proves that anti-debugging occurs in more than half of a randomly chosen data set of a packer. We should point out that packers are designed for obfuscation and compression, not anti-debugging. Finding such a big number of results might indicate that they are now also used for security purposes.

RQ2. *Is there a correlation between a set of anti-debugging methods and packers?*

During our research, we saw some trends in the categories, but more importantly, we discovered that the most often used anti-debugging strategies (such as IsDebuggerPresent) are basic and old. On the other side, we may show that there is no particular pattern, but that they strive to use a wide range of techniques that are either well-known or unique to each packer.

RQ3. *In our dataset of packers, which techniques are exploited?*

We added a chapter that broke down all of the techniques used by each packer, classifying them and comparing the results among the packers. The findings can be utilized as a foundation for further research as well as more confirmation of packer defenses and protection.

We provided a pool of newly discovered techniques that both verified previous results while also introducing new strategies for each packer.

7.0.2. Future works

We proposed different researches that could be linked using the same environment and approach. The first idea is to perform the same research using packed malware instead of files obfuscated with public packers. The motivation is that viruses usually implement a private and modified version of the public packers and could be interesting to compare how far are from our results or if we can recognize the original packers looking at the logs.

Future research can be used to identify new approaches for the research's most interesting example. Tools like Themida, Enigma, and Obsidium are quite complex and may conceal further techniques and routines. It may be used in conjunction with Instrumentation and Debugging to understand all parts of the program and how it operates.

Another possibility is to look at the packers themselves, rather than the packed file. During our research, we used our instrumentation tool to monitor the packers as they compressed the file. We discovered useful information and outcomes throughout our investigation. The study of packer software is motivated by the possibility that it will aid in the troubleshooting of a packer. We can exactly know which procedures are done during compression and develop better unpackers if we can debug a packer.

Bibliography

[1]

[2] D. Al-Anezi. Generic packing detection using several complexity analysis for accurate malware detection. *International Journal of Advanced Computer Science and Applications*, 5, 01 2014. doi: 10.14569/IJACSA.2014.050102.

[3] M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. 12, 03 2004.

[4] Cisco. Cisco annual internet report. Technical report, 2021. URL `https://www.cisco.com/c/en/us/solutions/executive-perspectives/annual-internet-report/index.html`.

[5] K. Coogan, S. Debray, T. Kaochar, and G. Townsend. Automatic static unpacking of malware binaries. In *2009 16th Working Conference on Reverse Engineering*, pages 167–176, 2009. doi: 10.1109/WCRE.2009.24.

[6] P. Ferrie. The "ultimate" anti-debugging reference. 04 2011.

[7] hex rays. Ida freeware. URL `https://hex-rays.com/ida-free/`.

[8] F. M. M. L. S. O. D. B. G. V. Hojjat Aghakhani, Fabio Gritti, S. B. . S. d. T. . . I. . f. Christopher Kruegel University of California, and ¶davide.balzarotti@eurecom.fr. When malware is packin' heat; limits of machine learning classifiers based on static analysis features. 02 2020. doi: 10.14722/ndss.2020.24310.

[9] M.-J. C. Jong-Wouk Kim1, Jiwon Bang2. Defeating anti-debugging techniques for malware analysis using a debugger. 11 2020. ISSN 2415-6698.

[10] S. KEMP. Digital global overview report, 2021. URL `https://datareportal.com/reports/digital-2021-global-overview-report`.

[11] T. linux programming interface. dnspy, . URL `https://www.intel.com/content/dam/develop/external/us/en/documents/cgo2013-256675.pdf`.

[12] T. linux programming interface. dnspy, . URL `https://github.com/dnSpy/dnSpy`.

[13] T. linux programming interface. objdump(1) — linux manual page, 02 2021. URL https://man7.org/linux/man-pages/man1/objdump.1.html.

[14] C. P. S. T. LTD. Anti debugging tricks, 2020. URL https://anti-debug.checkpoint.com/.

[15] L. Martignoni, M. Christodorescu, and S. Jha. Omniunpack: Fast, generic, and safe unpacking of malware. pages 431 – 441, 01 2008. ISBN 978-0-7695-3060-4. doi: 10.1109/ACSAC.2007.15.

[16] Microsoft. Sysinternals. URL https://docs.microsoft.com/en-us/sysinternals/.

[17] S. Morgan. Ventures, Cyber Defense Magazine, 2019. An optional note.

[18] N. N. M. P. S. B. U. Najmeh Miramirkhani, Mahathi Priya Appini. Spotless sandboxes: Evading malware analysis systems using wear-and-tear artifacts. 12 2017. doi: DOI10.1109/SP.2017.42.

[19] T. Naudi. Acunetix web application vulnerability report. Technical report, 2019. URL https://www.acunetix.com/blog/articles/acunetix-web-application-vulnerability-report-2019.

[20] M. P. M. C. A. C. S. Z. Nicola, Galloro. A systematical and longitudinal study of evasive behaviors in windows malware. 12 2021. doi: https://doi.org/10.1016/j.cose.2021.102550.

[21] M. C. Peter Ferrie, Senior Anti-Virus Researcher. Anti-unpacker tricks. 04 2011.

[22] M. Polino, A. Continella, S. Mariani, S. D'Alessio, L. Fontana, F. Gritti, and S. Zanero. Measuring and defeating anti-instrumentation-equipped malware. pages 73–96, 07 2017. ISBN 978-3-319-60875-4. doi: 10.1007/978-3-319-60876-1_4.

[23] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, and P. G. Bringas. Sok: Deep packer inspection: A longitudinal study of the complexity of run-time packers. In *2015 IEEE Symposium on Security and Privacy*, pages 659–673, 2015. doi: 10.1109/SP.2015.46.

[24] wiki. Peid wiki page. URL https://www.aldeid.com/wiki/PEiD.

[25] J. H. S. YOUNG BI LEE and I. DONG HOON LEE. Bypassing anti-analysis of commercial protector methods using dbi tools. 04 2020. doi: 10.1109/ACCESS.2020.3048848.

List of Figures

List of Tables

List of Symbols

Variable	Description	SI unit
$\boldsymbol{u}$	solid displacement	m
$\boldsymbol{u}_f$	fluid displacement	m

To be written